Fusing LLMs and Genetic Algorithm for High-Quality Unit Test Generation

YIWEN FU, Beihang University, China
XIANG GAO*, Beihang University, China and Hangzhou Innovation Institute of Beihang University, China
BINHANG QI*, National University of Singapore, Singapore and Beihang University, China
YUAN YUAN, Beihang University, China and Hangzhou Innovation Institute of Beihang University, China
HAILONG SUN, Beihang University, China and Hangzhou Innovation Institute of Beihang University, China

Unit testing is essential for early defect detection and software reliability. Existing automated test generation tools, including both search-based and Large Language Model (LLM)-centric approaches, still face significant challenges, including difficulties in producing compilable and executable tests for complex projects, limited coverage with insufficient exploration of control flows and edge cases, and a lack of systematic feedback mechanisms that integrate global search, semantic reasoning, and adaptive refinement. To address these limitations, we present LegaTest, an automated unit test generation framework that synergizes LLM with Genetic Algorithms (GA) through a coordinated Generation–Repair–Optimization process. LegaTest begins with a dual-prompt strategy to generate structurally sound and semantically meaningful test seeds, which are then iteratively refined using execution feedback and coverage guidance. The framework integrates LLM-driven insights with GA operators such as greedy selection, LLM-guided crossover, and semantic-driven mutation. Multi-strategy test repair with hierarchical criteria progressively resolves syntactic and semantic errors, improving both correctness and robustness. On the Defects4J benchmark as well as four Java projects adopted from the experimental setup of ChatUniTest, LegaTest achieves 66.95% line, 54.37% branch, and 74.16% method coverage with 93.3% generation success and 18.7% semantically meaningful assertions, outperforming existing tools by up to 34.7% in coverage, 59.6% in generation success, and 16.5% in assertions, while maintaining concise test suites.

Additional Key Words and Phrases: Automated Unit Testing, Test Generation, Large Language Models, Genetic Algorithms

ACM Reference Format:
Yiwen Fu, Xiang Gao, Binhang Qi, Yuan Yuan, and Hailong Sun. 2026. Fusing LLMs and Genetic Algorithm for High-Quality Unit Test Generation. 1, 1 (June 2026), 36 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Corresponding authors.

Authors' Contact Information: Yiwen Fu, Beihang University, Beijing, China, fuyiwen@buaa.edu.cn; Xiang Gao, Beihang University, Beijing, China and Hangzhou Innovation Institute of Beihang University, Hangzhou, China, xiang_gao@buaa.edu.cn; Binhang Qi, National University of Singapore, Singapore, Singapore and Beihang University, Beijing, China, qibh@nus.edu.sg; Yuan Yuan, Beihang University, Beijing, China and Hangzhou Innovation Institute of Beihang University, Hangzhou, China, yuan21@buaa.edu.cn; Hailong Sun, Beihang University, Beijing, China and Hangzhou Innovation Institute of Beihang University, Hangzhou, China, sunhl@buaa.edu.cn.

ACM XXXX-XXXX/2026/6-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

As software systems continue to grow in complexity, ensuring quality through effective testing becomes increasingly critical [75]. Among various quality assurance practices, unit testing plays a foundational role by enabling early detection and localization of defects [47]. However, manually writing high-quality unit tests remains labor-intensive and error-prone, requiring deep code understanding and significant domain expertise [67]. To reduce this burden, automated unit test generation has attracted growing attention.

Early automated test generation tools, such as EvoSuite [17] and Randoop [48], exemplify traditional approaches that employ a variety of software analysis techniques. These include search-based strategies [17], random test generation [48], model checking [15, 20], and symbolic execution [51], all of which are designed to systematically generate unit tests that achieve high code coverage and strong mutation scores.

More recently, Large Language Models (LLMs) have been widely applied to automated test generation. Early seq2seq-based models, such as AthenaTest [67] and A3Test [1], generate tests directly from code representations and apply lightweight post-processing to improve syntactic correctness. Building on this foundation, a variety of prompt-driven and refinement-based approaches, including ChatTester [78], ChatUniTest [9], TestPilot [58], CoverUp [3], TestART [21], HITS [70], and SymPrompt [55], exploit structured prompts, iterative validation and repair, coverage feedback, and symbolic reasoning to produce semantically meaningful and executable test cases. Together, these methods highlight the potential of LLMs to generate tests with descriptive names, structured inputs, and richer assertions, while effectively incorporating both syntactic and semantic information from the code and its documentation.

Building on the capabilities of LLM-centric approaches, recent hybrid methods further integrate LLMs with search-based software testing (SBST) techniques to leverage the strengths of both paradigms. For instance, CodaMosa [33] combines LLM guidance with search heuristics to prioritize unexplored code regions and optimize coverage, while UTGen [12] generates a high-coverage baseline suite using EvoSuite and subsequently refines it with LLM-based transformations, including rewriting test data, renaming variables, adding comments, and producing descriptive method names, resulting in test cases that are both semantically meaningful and maintainable.

Despite the advances of both traditional and LLM-based approaches, existing automated test generation tools still face fundamental limitations:

Limitation 1 – Test validity and executability. Existing tools often fail to produce compilable and executable tests for complex Java projects. Traditional search-based methods struggle with bytecode instrumentation for deeply nested builders, lambdas, or large methods, while generics, abstract classes, or complex constructors can block compilation [55, 73]. LLM-based approaches frequently lack precise reasoning about class hierarchies, field initialization, generic bounds, and constructor dependencies, leading to syntactically invalid or semantically fragile tests [1, 4, 69, 70]. Consequently, large portions of code remain without runnable tests. For example, in our evaluation, the search-based tool EvoSuite successfully generated tests for only 33.75% of classes, while the LLM-based ChatUniTest succeeded on just 37.97% of classes.

Limitation 2 – Low coverage and inadequate exploration. Even when tests are executable, traditional search-based tools often explore only shallow paths due to local heuristics and premature convergence, generating redundant or overly simplistic test cases that cover limited branches. LLM-based methods, while leveraging semantic understanding, generally lack systematic exploration mechanisms and global path reasoning; they tend to generate tests along straightforward or "obvious" code paths and may miss complex control-flow conditions, deeply nested branches, or rare edge cases [1, 4, 69, 70]. Consequently, neither approach alone achieves sufficient coverage

for complex, real-world systems, limiting the effectiveness of automated testing [8, 21, 57]. For example, in our experiments across 403 benchmark classes, state-of-the-art LLM-based tools[9] achieve only about 38.31% line coverage, 29.65% branch coverage, and 39.50% method coverage on average, while traditional search-based tools EvoSuite reach around 58.66%, 53.41%, and 61.27%, respectively, highlighting the substantial portion of program behaviors that remain untested.

Limitation 3 – Lack of iterative semantic optimization. Existing test generation methods generally lack systematic feedback mechanisms that refine tests based on semantic correctness. Traditional search-based approaches focus on structural coverage metrics and provide little guidance on program semantics, state-dependent behaviors, or protocol compliance. LLM-driven approaches can perform multiple rounds of generation, validation, or repair, yet these iterations are typically local, focusing on refining individual test cases rather than coordinating exploration across the entire input space. As a result, existing tools rarely integrate global search, semantic reasoning, and adaptive refinement into a unified process, limiting their ability to generate test suites that are simultaneously high-coverage, semantically coherent, and robust across complex software systems [55, 71, 73].

Limitation 4 – Weak assertion quality and limited semantic validation. Existing automated test generation methods struggle to produce assertions that go beyond superficial correctness checks. Traditional search-based tools often generate trivial or generic assertions (e.g., constant comparisons or `assertTrue(true)`), which provide little semantic value and fail to validate program behaviors beyond structural coverage. While LLM-based approaches can generate more natural and meaningful assertions, they frequently lack systematic grounding in program states or domain-specific semantics. As a result, they may overfit to syntactic patterns, omit critical property validations, or introduce fragile checks that do not generalize across inputs [30, 74, 80]. Consequently, the generated test suites tend to miss deeper behavioral guarantees, leading to limited fault-detection capability and reduced confidence in software correctness.

While recent hybrid approaches such as CodaMosa [33] and UTGen [12] attempt to combine search-based exploration with LLM-guided generation, they still inherit most of the above limitations rather than resolving them. Specifically, CodaMosa invokes LLMs only when search stagnates but provides no iterative repair: tests that fail due to type errors or semantic mismatches are simply discarded instead of being fixed. Moreover, because CodaMosa does not encode type information into prompts, the LLM often guesses parameter behavior from superficial cues (e.g., method names), resulting in semantically inconsistent tests, runtime failures, or fragile assertions. UTGen, in turn, is constrained by its reliance on EvoSuite: issues such as inability to handle complex constructors or user-defined types persist, and the LLM's refinements may inadvertently alter inputs or execution paths, sometimes lowering coverage compared to raw EvoSuite output. For classes with many parameters or tight coupling, UTGen struggles to enhance semantic validity, and its readability improvements frequently come at the cost of redundant or overly generic comments that add little to assertion quality.

To address the limitations of existing test generation tools, we propose LegaTest, a closed-loop framework that tightly integrates a Genetic Algorithm with LLM guidance. Test generation is formulated as an optimization problem: Genetic algorithm explores the search space efficiently, while the LLM provides semantic reasoning to guide crossover and context-aware mutation, ensuring structural validity and meaningful variations. Specifically, LegaTest generates initial tests via LLM using a structured prompt framework that incorporates comprehensive information about class hierarchies, constructors, field initialization, and domain-specific semantics. This rich contextual guidance enables the generation of deep, semantically meaningful assertions. The initial tests are then refined through a Generation–Repair–Optimization loop. Within this loop, iterative evolution module drives population-based exploration to maintain diversity and quality, while repair module

systematically corrects invalid tests using template-based heuristics and LLM-assisted strategies, ensuring semantic correctness and executability throughout the iterative process.

We first evaluate the initial version of our tool on the widely used Defects4J [28] as well as four Java projects adopted from the experimental setup of ChatUniTest [9], covering 403 complex public classes. Experimental results show that our approach consistently outperforms state-of-the-art baselines, including EvoSuite [17] and ChatUniTest [9]. On average, our method achieves 66.95% line coverage, 54.37% branch coverage, and 74.16% method coverage, compared to 58.66%/53.41%/61.27% for EvoSuite and 38.31%/29.65%/39.50% for ChatUniTest. In terms of generation success, runnable tests are produced for 376 out of 403 classes (93.3%), whereas EvoSuite and ChatUniTest succeed on only 136(33.75%) and 153(37.97%) classes, respectively. Even when executed, the tests remain highly robust: 91.52% of our generated tests run successfully, which is close to EvoSuite's 99.29% and far higher than ChatUniTest. Moreover, our approach generates 16.5% and 3.2% more tests with complex, semantically meaningful assertions than EvoSuite and ChatUniTest, while still maintaining much smaller and more concise test suites.

To further demonstrate the effectiveness of our full framework, we additionally conduct experiments on a selected subset of 100 classes, where we apply an iterative optimization strategy. This extended evaluation complements the benchmark-wide comparison by showing the full potential of our tool when optimization is incorporated. On this subset of 100 classes, the optimized version of our tool substantially improves coverage under both LLM backends. With DeepSeek, average line, branch, and method coverage increase from 62.11% → 79.91%, 49.38% → 67.75%, and 70.61% → 86.11%, respectively, while the average test suite size grows from 13.87 to 26.52. With GPT, the corresponding improvements are from 73.21% → 84.05% (line), 62.74% → 72.90% (branch), and 79.2% → 89.82% (method), with test suite size increasing from 16.15 to 20.98. In comparison, EvoSuite achieves 74.01% line, 68.93% branch, and 78.46% method coverage with an average of 24.69 test cases. ChatUniTest reaches 47.89%/31.82% line, 41.53%/29.22% branch, and 48.27%/32.22% method coverage, with 29.26 and 27.53 tests on average for DeepSeek and GPT, respectively. UTGen achieves 33.03%/41.51% line, 29.52%/36.41% branch, and 39.43%/47.73% method coverage, with 13.26 and 9.17 tests on average. Overall, the iterative optimization consistently improves coverage over the initial version and outperforms these existing tools across both LLM backends, while maintaining compact and practical test suites.

Overall, our method achieves broad generation applicability, markedly higher coverage, and strong execution reliability, while producing more semantically meaningful assertions with superior readability and maintainability.

This paper makes the following key contributions:

- We introduce LegaTest, an approach that combines a genetic algorithm with LLM guidance to generate high-coverage, semantically meaningful, and structurally valid unit tests.
- We propose a structured prompt framework that leverages global behavioral rules and class-specific contextual information to generate highly executable tests for complex classes while producing deep, semantically meaningful assertions, improving syntactic correctness, semantic validity, and overall test expressiveness, providing a robust foundation for subsequent iterative refinement and broadening the applicability of LLM-based test generation to real-world Java projects.
- We evaluate our approach on 403 classes from Defects4J as well as four Java projects adopted from the experimental setup of ChatUniTest and demonstrate substantial improvements in branch coverage, assertion quality, and maintainability compared to existing search-based and LLM-based methods.

Due to space limitations, the tool source code, additional experimental results, and concrete examples are provided in [46].

2 Background

Genetic Algorithms: Swarm intelligence (SI) algorithms are bio-inspired, population-based optimization methods that leverage the collective behaviors of decentralized agents to achieve global optimization [7, 66, 72]. Unlike gradient-based methods, SI algorithms do not require continuity or differentiability of the objective function, which makes them well-suited for high-dimensional, nonlinear, and constrained problems [37, 38]. Among SI-inspired methods, Genetic Algorithms (GAs) were first proposed by Holland in the 1970s [25], inspired by Darwin's theory of natural selection and the mechanisms of classical genetics. GAs simulate the evolutionary process by maintaining a population of candidate solutions encoded as chromosomes, which evolve over generations through biologically inspired operators such as selection, crossover, and mutation [29, 32, 54]. Formally, given a search space Ω and an objective function $f : \Omega \to \mathbb{R}$, GA maintains a population $P^t = \{c_1^t, c_2^t, \ldots, c_N^t\}$ at generation t. The evolutionary cycle can be abstracted as:

$$P^{t+1} = \mathcal{M}\big(\mathcal{C}(\mathcal{S}(P^t, f))\big), \tag{1}$$

where $\mathcal{S}$ denotes selection of parents according to fitness $f(c_i^t)$, $\mathcal{C}$ performs crossover to recombine them, and $\mathcal{M}$ applies mutation to maintain diversity.

Applications of Genetic Algorithms in Software Engineering: Genetic Algorithms (GAs) are population-based, bio-inspired optimization techniques widely used in software engineering [6]. Many tools leverage basic evolutionary operators—selection, crossover, and mutation—to explore complex search spaces. GAs have been applied to automated unit test generation, fuzzing input synthesis, API driver creation and Prompt optimization [13, 14, 35, 60]. Representative examples include EvoSuite [17], which generates JUnit test suites for Java programs, and GA-based fuzzing tools that evolve inputs to exercise different program behaviors. Through iterative evolutionary operations, GAs can produce diverse, structurally valid inputs, improving the effectiveness of automated testing [26].

Engineering Significance of JUnit5: JUnit5's modular design (Platform, Jupiter, Vintage) offers greater flexibility than JUnit4, supporting dynamic tests [19, 62], parameterized execution, and extensibility through annotations like `@ExtendWith` [22, 24]. It integrates well with modern Java frameworks (e.g., Spring), improves readability via AssertJ, and supports features like nested tests and concurrent execution. These capabilities make JUnit5 especially suited for complex testing scenarios in modern CI/CD pipelines [53] and microservice environments [77].

3 Methodology

To address the limitations of existing test generation techniques, we propose LegaTest, an Orchestrated Framework for Automated Unit Test Generation via LLM and Genetic Algorithm Fusion. This framework integrates the natural language understanding capabilities of LLMs with the global search and local optimization strengths of Genetic Algorithms. Through a synergistic design, LegaTest orchestrates the full lifecycle of test generation, from initial generation to repair and iterative optimization. The overall workflow of the framework is illustrated in Figure 1.

3.1 Initial Test Generation

Although LLM-generated tests often provide high readability and reduce manual effort [21], many fail to compile or execute in complex, real-world systems due to hallucinations [27, 81] and insufficient semantic grounding. Moreover, existing LLM-based approaches frequently target individual

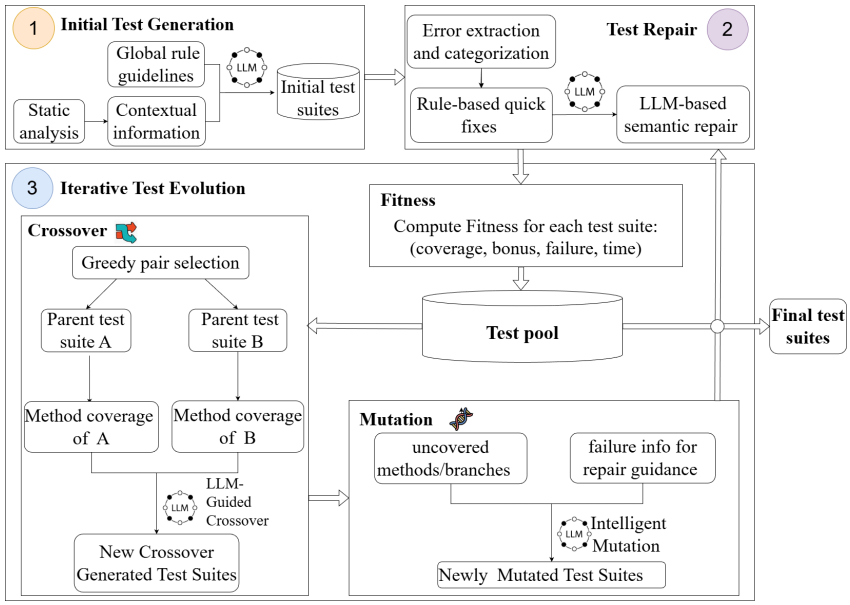


Fig. 1. The overall workflow of LegaTest

Table 1. Prompt Specifications for JUnit 5 Test Generation

(a) System Prompt Specifications

Category	Requirement Description
Core Test Generation Rules	Write full Java test class, JUnit 5 annotations, test only public methods, include edge cases, follow naming convention, each with clear assertions.
Most Critical Rule	ONLY test public methods; NEVER private/protected or reflection.
Test Uniqueness	Each suite must be unique, follow focus, no reuse of methods/scenarios/asserts.
Package/Class Declaration	Use exact package of CUT, never `org.example`, correct `package` declaration.
Annotation Usage	Annotations before methods, `@Test` always, `@BeforeEach` for setup.
Code Quality	Readable, independent tests, comments if needed, proper formatting.
Access Control	Forbidden: private access/reflection. Allowed: via public API only.
Public API Usage	Respect signatures, only use given public methods/fields.
Output Format	Only Java code, start with `package`, include imports + test class.

(b) User Prompt Specifications

Prompt Section	Description
CLASS INFORMATION	Metadata: package, name, Java version, imports.
METHODS SECTION	Public methods with optional call relations.
METHOD CALL HIERARCHY	Relationships for indirect coverage.
FIELD USAGE	Fields with access and visibility.
DEPENDENCIES	Inheritance, interfaces, instantiations.
VARIABLE INFORMATION	Tracks variables and modifications.
CONSTRUCTOR DEPS	Constructor signatures and init usage.
TESTING SCOPE	Strict: public APIs only, no reflection.
TEST FOCUS	Strategy (e.g., boundary, happy path).
CODE	Full Java CUT implementation.
TEST GUIDELINES	High-level test writing rules.
OUTPUT INSTRUCTIONS	Expected output: complete JUnit 5 test class.

methods in isolation, overlooking class-level semantics, object states, and inter-method dependencies. This results in limited test diversity and reduced industrial applicability. To overcome these limitations, we design a structured prompt framework that integrates static analysis-derived semantic contexts with globally enforced rules, explicitly emphasizing readability, maintainability, and expressive assertions to ensure that generated tests are both practical and semantically meaningful.

To enhance reliability and semantic correctness, we incorporate both global rules and code-specific contextual information into the prompt. As shown in Table 1a, the *System Prompt* defines overarching behavioral and structural requirements, such as Core Test Generation Rules, Annotation Usage, and Public API Usage, which apply uniformly across all generated tests. Complementarily, as listed in Table 1b, the *User Prompt* provides class-specific semantic details extracted through static analysis, including CLASS INFORMATION, METHOD CALL HIERARCHY, and DEPENDENCIES. By combining these two components, the prompt embeds both general principles and fine-grained semantics, enabling the generation of executable and type-correct tests even for complex cases such as abstract classes instantiated via subclasses, generics, nested builders, or lambda expressions.

To further improve the diversity and rigor of generated tests, we integrate the *Test Focus Control* mechanism into the prompt design. This mechanism systematically directs the exploration of multiple testing strategies to achieve broader and more fine-grained coverage. Specifically, *Test Focus Control* (i.e. TEST FOCUS in Table 1b) distinguishes different testing emphases for each class under test. Among these, the first group of five strategies is explicitly aligned with well-defined focus methods (e.g., boundary value analysis, exception path validation, dependency injection coverage, and state transition testing), ensuring targeted coverage. In contrast, the latter five strategies encourage large language models to autonomously explore potential scenarios, thereby fostering creativity and uncovering unanticipated cases. The integration of these complementary strategies within the prompt establishes a rich foundation for subsequent crossover operations and increases coverage of complex and rare behaviors.

To prioritize meaningful and well-targeted test suites, we incorporate semantic summarization and method complexity analysis before test generation. Functional intent is inferred from class names, annotations, and identifier semantics, with naming conventions employed as a fallback when documentation is unavailable. Method complexity is evaluated through visibility, logic density, and parameter usage, which provides a principled basis for prioritization. This high-level understanding is then encoded into the User Prompt to guide context-aware test generation.

To ensure readability and maintainability, the prompt enforces explicit coding conventions and structured test patterns. Each test method adopts a Behavior-Driven Development (BDD)–style `@DisplayName`, where the naming format `Verify [functionality] should [expected result] when [condition]` expresses the intended behavior in a human-readable manner. This BDD-style convention makes tests easier to understand by clearly linking functionality, expectation, and context. Test bodies adhere to the Arrange–Act–Assert (AAA) pattern, maintaining separation across its three phases. Test classes inherit the original package structure, ensuring seamless integration into Maven-based projects.

Finally, to strengthen assertion generation, the prompt directs the framework to perform assertion completion. Beyond shallow existence or equality checks, it synthesizes semantic-aware, boundary-driven, and behavior-oriented assertions. By leveraging static analysis and semantic information from the User Prompt, the framework incorporates method signatures, call chains, field usage, constructor dependencies, and edge cases or constants (e.g., `Long.MIN_VALUE`, `Double.NaN`, multi-dimensional arrays) into deep assertions that validate both object behavior and business constraints. Compared with prior approaches, this design yields more precise, expressive, and executable postconditions, thereby increasing both robustness and real-world utility of the generated tests.

All information within the prompt is either pre-defined or automatically collected by LegaTest. Concrete examples illustrating the prompt construction are presented on the project website [46].

3.2 Unit Test Repair

In test generation systems, the compilability and executability of test classes are crucial metrics for quality assessment. To ensure these properties, we propose a test repair mechanism based on

a multi-strategy approach with hierarchical success criteria and phased strategy coordination. It systematically repairs failed test classes at both structural and semantic levels, ensuring syntactic correctness and behavioral consistency.

The repair pipeline follows four phases with gradually relaxed success criteria to maximize recovery of usable tests while maintaining suite quality. In the first phase, all generated tests are executed, and those that compile and pass are accepted immediately. In the second phase, common build errors such as import conflicts or duplicate definitions are addressed through conservative rule-based repair using predefined templates. If rule-based fixes fail, the third phase invokes LLM-based semantic repair, where compilation logs are leveraged to guide the LLM in performing targeted repairs with up to two attempts, focusing on salvaging syntactically valid classes. Finally, the fourth phase discards tests that remain uncompilable, preventing noise from degrading the overall suite.

The key innovation of our approach lies in its *Error Extraction and Categorization Mechanism*, which enhances repair accuracy by supplying LLMs with structured and semantically filtered diagnostic context rather than raw logs. The compilation logs are first normalized into standardized error objects, each mapped to fine-grained categories such as missing symbols, constructor mismatches, unresolved classes or packages, access violations, type errors, or test failures. These categories are hierarchically abstracted into two diagnostic classes—*compilation errors* and *test failures*—which drive distinct repair strategies.

For test failures, assertion traces provide sufficient semantic cues and are used directly. For compilation errors, contextual analysis is performed: self-contained categories like missing imports or duplicate methods are handled without source augmentation, while more complex cases undergo selective source snippet extraction. To ensure token efficiency, a value assessment filters snippets by length, relevance, and code density; only valuable fragments are retained. Depending on error semantics, the mechanism extracts relevant members, reconstructs class signatures, or preserves method headers, while abstracting away method bodies to reduce noise. Extracted errors are then serialized into structured repair prompts that include canonical diagnostic headers, concise error descriptions, and dynamically injected repair strategies. For example:

```
Compilation Error:
- Missing import: 'X'
- Syntax issue: incompatible types (String → int)
- API issue: cannot find method 'someMethod'

Test Failure:
- Method someTest: expected <A> but was <B>
```

This end-to-end mechanism achieves a principled balance between diagnostic precision and token efficiency: it suppresses redundant verbosity while preserving semantically critical cues, thereby enabling LLMs to perform targeted, interpretable, and robust program repair. Unlike prior approaches that rely solely on raw error logs or undifferentiated prompting, our mechanism explicitly categorizes errors and selectively integrates source context, improving both repair hit rate and reliability.

3.3 Iterative Test Evolution

The Iterative Optimization employs evolutionary strategies—selection, crossover, and mutation—to progressively refine the test suite, aiming to improve coverage and quality. The workflow of the Iterative Evolution is outlined in Algorithm 1. Each generation begins with an evaluation against predefined termination conditions. The optimization process terminates upon reaching the

Algorithm 1 Iterative Evolution for Test Suite Refinement

Input: $TestSuite_{\text{init}}$: Initial test suite, $maxGen$: Maximum generations, N: Population size
Output: $TestSuite_{\text{opt}}$: Optimized test suite
$TestSuite \leftarrow TestSuite_{\text{init}}$
$generation \leftarrow 1$
while True **do**
    EvaluateFitnessAndCoverage($TestSuite$)
    if $generation \geq maxGen$ **or** (BranchCoverage ≥ 0.98 **and** MethodCoverage ≥ 0.98) **then**
        break
    end if
    $(parent_1, parent_2) \leftarrow$ SelectMostDiverse($TestSuite$)
    $offspring \leftarrow$ Crossover($parent_1, parent_2$, hybrid_prompting)
    for $i \leftarrow 1$ **to** Length($offspring$) **do**
        if BranchCoverage < 0.95 **then**
            $rate \leftarrow BaseMutationRate \times$ MutationFactor($offspring[i].fitness$)
            if Random$(0, 1) < rate$ **then**
                $offspring[i] \leftarrow$ Mutate($offspring[i]$, hybrid_prompting)
            end if
        end if
    end for
    $best \leftarrow$ GetBestIndividual($TestSuite$)
    if BranchCoverage < 0.95 **then**
        $elite\_mutated \leftarrow$ Mutate($best$, hybrid_prompting)
        $TestSuite \leftarrow TestSuite \cup offspring \cup \{elite\_mutated\}$
    else
        $TestSuite \leftarrow TestSuite \cup offspring$
    end if
    EvaluateFitnessAndCoverage($TestSuite$)
    $TestSuite \leftarrow$ SelectTopN($TestSuite, N$)
    $generation \leftarrow generation + 1$
end while
return $TestSuite$

maximum number of generations, a parameter that can be adjusted to meet different experimental objectives. Otherwise, the algorithm proceeds to the next iteration. In each iteration, the two most diverse tests are selected as parents to maximize variety, and crossover and mutation are guided by LLM to generate new candidates. Among all candidates, individuals are ranked according to their fitness values, and the top-N tests are retained for subsequent cycles. This fitness-driven elitist preservation retains only the best candidates, steering evolution toward semantically valid and coverage-optimized test suites.

3.3.1 Fitness Definition. In evolutionary test generation, the fitness function serves as a quantitative measure of the quality of each individual test suite, determining which candidates are retained for subsequent iterations and thereby driving the selection of high-quality tests for ongoing optimization.

To effectively guide this selection process, the fitness function is designed as a multi-objective [83] evaluator that prioritizes coverage while incorporating failure rate and execution time as auxiliary constraints. Unlike single-objective schemes [52], this design encourages the generation of test suites that not only achieve high coverage but also maintain reliability and efficiency, better aligning with the practical demands of automated testing in real-world scenarios. The fitness score

is computed as:

$$\text{Fitness}(i) = \max\left(0, C(i) + B(i) - P_f(i) - P_t(i)\right) \tag{2}$$

Here, $C(i)$ denotes the base coverage score for individual i, calculated as a weighted sum of line ($C_l(i)$), branch ($C_b(i)$), and method ($C_m(i)$) coverage:

$$C(i) = \alpha_l \cdot C_l(i) + \alpha_b \cdot C_b(i) + \alpha_m \cdot C_m(i) \tag{3}$$

By default: $\alpha_l = 0.5$, $\alpha_b = 0.4$, $\alpha_m = 0.1$.

To reward high-quality test suites, a coverage bonus term $B(i)$ is introduced, which assigns up to $\beta_{\max}$ points for coverage levels beyond γ, with diminishing rewards for lower coverage bands. By default: $\beta_{\max} = 0.3$, $\gamma = 95\%$.

The failure penalty $P_f(i)$ is defined based on the observed failure rate of test case i and capped at $\delta_{\max}$, ensuring that unstable tests are discouraged without excessively penalizing rare failures. When failure information is unavailable, a default penalty of δ_{default} is applied. By default: $\delta_{\max} = 0.05$, $\delta_{\text{default}} = 0.05$. The time penalty $P_t(i)$ accounts for execution cost and is computed as:

$$P_t(i) = \min\left(\tau_{\max}, \frac{t(i)}{\tau_{\text{scale}}}\right), \tag{4}$$

where $t(i)$ is the execution time of test case i in seconds, and penalties are applied only when $t(i) > \tau_{\text{threshold}}$. By default: $\tau_{\max} = 0.02$, $\tau_{\text{scale}} = 10$, $\tau_{\text{threshold}} = 10$.

This fitness formulation ensures non-negativity and balances the competing objectives of maximizing coverage while maintaining robustness and efficiency. It effectively steers the evolutionary process toward selecting tests that are both high-coverage and practically executable.

3.3.2 LLM-Guided Crossover with Greedy Selection. In evolutionary test generation, the effectiveness of crossover highly depends on the quality and diversity of parent test suites. Traditional methods such as Roulette Wheel [36], Tournament [5], and Random Selection [63] often suffer from premature convergence, reduced exploration, and lack of semantic awareness. They typically ignore key distinctions in test suite semantics, coverage, and structural features. To address these limitations, we propose a greedy selection strategy based on a divergence metric, which quantifies differences between test suites in terms of semantics, method coverage, and execution paths. Instead of random pairing, the strategy prioritizes high-divergence pairs for crossover, thereby promoting structural diversity and complementary test behaviors. The selection process is carried out in three stages to ensure diversity while maintaining feasibility: (1) parent pairs are initially selected without repetition, so that each individual participates in at most one pair; (2) if the number of available candidates is insufficient to form all pairs, some individuals may be selected multiple times; (3) in extreme low-sample scenarios, previously selected individuals can be re-paired under carefully controlled constraints to allow crossover to proceed.

We define the divergence between two unit tests $D(a, b)$ as a weighted combination of three dimensions:

$$D(a, b) = \theta_{\text{sem}} \cdot D_{\text{sem}}(a, b) + \theta_{\text{meth}} \cdot D_{\text{meth}}(a, b) + \theta_{\text{path}} \cdot D_{\text{path}}(a, b), \tag{5}$$

where:

D_{sem}: Semantic Divergence, quantifying differences in test method names and associated display names or comments;

D_{meth}: Method Coverage Divergence, comparing the sets of methods each test covers;

D_{path}: Path Coverage Divergence, reflecting differences in the sets of entry(first) method signatures of execution paths covered by each test.

By default, the weights are set to $\theta_{\text{sem}} = 0.4$, $\theta_{\text{meth}} = 0.3$, and $\theta_{\text{path}} = 0.3$, ensuring a balanced contribution from each dimension.

Each component is computed using the Jaccard distance [31]:

$$D_{\text{sem}}(a,b) = 1 - \frac{|T_a \cap T_b|}{|T_a \cup T_b|} \tag{6}$$

where T_a and T_b are token sets extracted from method names and comments.

$$D_{\text{meth}}(a,b) = 1 - \frac{|M_a \cap M_b|}{|M_a \cup M_b|} \tag{7}$$

$$D_{\text{path}}(a,b) = 1 - \frac{|P_a \cap P_b|}{|P_a \cup P_b|} \tag{8}$$

All divergence values are normalized to $[0,1]$, with 1 indicating maximal divergence and 0 indicating full similarity.

While divergence-guided selection ensures high-quality and diverse parent candidates, the crossover operator itself remains a critical bottleneck. Traditional random crossover methods [50] are often ineffective in practice. They struggle to ensure structural and semantic complementarity, frequently resulting in redundant paths, limited coverage gains, and invalid combinations. Additionally, faulty logic in parent tests may be inherited, generating large numbers of invalid offspring and wasting computational resources. To overcome these limitations, we introduce an LLM-guided crossover approach based on structural and semantic analysis. Building on the selected high-divergence parent pairs, the crossover process leverages LLM as both a semantic interpreter and a structural reassembler. This approach begins with multi-dimensional static analysis, extracting information such as covered and uncovered methods as well as test class metadata including imports, field declarations, and setup methods. A coverage complementarity analysis is then performed to identify high-value fusion candidates by quantifying differences between parent test suites, including methods unique to each parent and shared uncovered paths. These structured insights are subsequently utilized to instruct the LLM to perform effective crossover. Guided by such information, the LLM synthesizes a merged test class that integrates the most effective tests from both parents. In addition, it generates new tests for previously uncovered methods. Together, these steps ensure semantic coherence, structural completeness, and enhanced overall coverage.

Crossover Prompt Illustration. Listing 1 presents the core prompt used for LLM-guided crossover. The prompt provides LLM with parent test suites, coverage information, and uncovered code regions, and instructs it to generate new tests by merging parent test suites while maintaining semantic correctness:

```
Crossover Objectives:
1. Preserve all valid tests from both parents.
2. Merge complementary test logic to improve coverage.
3. Generate new test methods to exercise uncovered code regions.
Constraints:
- Ensure semantic coherence.
- Avoid duplicating existing test logic.
```

Listing 1. LLM-Guided Crossover Prompt

3.3.3 Intelligent Mutation. Mutation introduces structural variations into offspring following crossover, facilitating both exploration and convergence in evolutionary test generation. To control mutation effectively, we consider both the global mutation schedule and individual-level adjustments.

A global, exponentially decaying mutation rate [59] is employed:

$$\mu(t) = \mu_{\min} + (\mu_{\max} - \mu_{\min}) \cdot e^{-k \cdot t} \tag{9}$$

where $\mu_{\max} = 0.3$, $\mu_{\min} = 0.1$, $k = 0.1$ and t denotes the iteration number. This schedule encourages stronger mutations in early generations for exploration and reduced mutations in later generations to enhance stability and facilitate convergence.

Each individual's mutation rate is further adjusted based on its fitness. Let $Fitness(i)$ denote the fitness of individual i, then the effective mutation rate μ_i is defined as:

$$\mu_i = \begin{cases} 0.5 \cdot \mu(t), & \text{if } Fitness(i) \geq 0.8, \\ 1.2 \cdot \mu(t), & \text{if } Fitness(i) \leq 0.3, \\ \mu(t), & \text{otherwise.} \end{cases} \tag{10}$$

Specifically, high-fitness individuals adopt reduced rates to preserve structural advantages, while low-fitness individuals increase mutation to promote diversity. Intermediate individuals retain the baseline rate. Notably, elite individuals are always subjected to mutation, ensuring that the most promising solutions are refined further.

The aforementioned adaptive mutation strategy primarily addresses the first core issue of which test to mutate (the Which) by applying varying mutation rate. However, the equally critical question of how to mutate (the How), i.e., generating new and effective test code, has received comparatively less attention and remains inadequately resolved in traditional methods. Traditional methods [23], whether random [18] or rule-based, often lack semantic awareness and generate invalid or irrelevant tests. To address this problem, we propose a semantic-prompt-driven intelligent mutation strategy with an adaptive mutation rate control mechanism based on code context, failure information, existing tests, and logic flow.

Our approach begins by analyzing uncovered code regions, focusing on conditional branches and entry methods. To ensure a focus on meaningful code, the system first applies a rule-based filtering. Specifically, this filtering process retains critical branches such as conditional jumps (including those with keywords like if, goto, or branch), exception handling, and exception construction, while discarding trivial ones like pure assignments, unthrown exceptions, and JVM internal operations. Similarly, methods are filtered to exclude simple getters/setters, internal helpers, and utility routines, preserving instead important public APIs and critical business logic. Given an uncovered method or branch, LegaTest perform mutation by first identifying tests of those methods. These test exemplars serve as concrete templates, guiding the LLM to adapt and extend them for generating new tests that exercise the missing branches. Concurrently, failing test cases are extracted, and the LLM is provided with the names of the failing methods, the type of failure, and detailed error messages. This enables the generation of mutations that simultaneously extend code coverage and repair existing test errors.

By combining adaptive, fitness-based mutation rate control with semantic guidance that leverages existing tests and failure information, the proposed strategy preserves effective solutions while generating high-quality, targeted test cases that improve coverage, repair failing tests, and strengthen the overall test suite.

Mutation Prompt Illustration. Listing 2 presents the core prompt used for LLM-guided mutation. The prompt provides LLM with a test suite, coverage gaps, and failure information, and instructs it to generate or repair tests to improve branch and path coverage:

```
Mutation Objectives:
1. Identify uncovered branches or paths in the code under test.
2. Repair any failing tests using failure diagnostics.
3. Generate new tests to cover previously uncovered branches.
Constraints:
- Preserve the intent of existing tests.
- Ensure semantic correctness and structural validity.
```

Listing 2. LLM-Guided Mutation Prompt

The following concrete example illustrates LLM-guided crossover and semantic, branch-aware mutation through a worked, code-level case on a `DiscountPolicy` class with conditional logic and boundary constraints.

```java
public class DiscountPolicy {
    public int compute(int price, boolean premium, boolean holiday) {
        if (price <= 0) throw new IllegalArgumentException("Invalid price");
        // [Logic A] Premium users get 15, others get 5
        int discount = premium ? 15 : 5;
        // [Logic B] Holiday bonus adds 10 (Condition: price > 100)
        if (holiday && price > 100) discount += 10;
        // [Boundary] Maximum discount is capped at 20
        return Math.min(discount, 20);
    }
}
```

Target Class Under Test (Business Logic & Boundary)

Phase 1: Divergence-Guided Parent Selection. Two parents are selected based on divergence. Parent A covers basic/error paths, while Parent B covers premium features. The combination path (15 + 10) is missing.

```java
@Test
@DisplayName("Invalid price inputs should throw IllegalArgumentException")
void testInvalid() {
    DiscountPolicy p = new DiscountPolicy();
    assertThrows(IllegalArgumentException.class,
        () -> p.compute(0, false, false));
}
@Test
@DisplayName("Standard user receives default discount of 5")
void testStandard() {
    DiscountPolicy p = new DiscountPolicy();
    assertEquals(5, p.compute(50, false, false));
}
```

Parent A (Basic & Error)

```java
@Test
@DisplayName("Premium user receives higher base discount of 15")
void testPremium() {
    DiscountPolicy p = new DiscountPolicy();
    assertEquals(15, p.compute(80, true, false));
}

@Test
@DisplayName("Holiday bonus applies when price exceeds threshold")
void testHoliday() {
    DiscountPolicy p = new DiscountPolicy();
    assertEquals(15, p.compute(120, false, true));
}
```

Parent B (Premium & Holiday)

Phase 2: LLM-Guided Crossover (Fusion). The LLM merges the logic but initially misses the hidden boundary cap (predicting 25 instead of 20).

```
# Input:
- Parent A (Basic Logic)
- Parent B (Premium Logic)
# Uncovered Path Analysis:
- Condition 1: premium == true
- Condition 2: holiday == true
- Condition 3: price > 100
- Goal: Verify combined discount logic.

# Fusion Instructions:
1. Merge Parent A and B logic.
2. Synthesize NEW test for Uncovered
   Path (Premium + Holiday).
3. Predict assertion values based on
   logic flow.
```

Crossover Context (LLM Prompt Inputs)

```
class DiscountTest {
    // ... Merged tests from A & B ...
    @Test
    @DisplayName("Synthesized path combining
         Premium and Holiday logic")
    void testPremiumHoliday() {
        // Logic Fusion:
        // Premium(15) + Holiday(10) = 25
        // LLM Prediction Error:
        // Misses Math.min(x, 20) cap
        assertEquals(25,
            new DiscountPolicy()
            .compute(150, true, true));
    }
}
```

Offspring (Intermediate Result)

Phase 3: Coverage-Driven Mutation. The offspring executes and fails (`expected: <25> but was: <20>`). Mutation repairs the failure and targets the uncovered boundary.

```
# Execution Feedback:
- Test: testPremiumHoliday
- Status: FAILED
- Msg: expected <25> but was <20>



# Coverage Gap Analysis:
- Method: Math.min(discount, 20)
- Gap: Boundary Saturation not taken.


# Mutation Objectives:
1. REPAIR failing assertion using
   feedback from execution.
2. GENERATE new test for
   saturation boundary.
3. ENSURE all branches and paths
   are fully covered.
```

Mutation Context (Gap & Failure)

```
class DiscountTest {
    // ... Other tests ...
    @Test // 1. REPAIRED based on feedback
    @DisplayName("Premium and Holiday
         combination should cap at 20")
    void testPremiumHoliday() {
        assertEquals(20,
            new DiscountPolicy()
            .compute(150, true, true));
    }
    @Test // 2. NEW BOUNDARY TEST
    @DisplayName("Saturation boundary
         explicitly validates max cap")
    void testSaturationBoundary() {
        assertEquals(20,
            new DiscountPolicy()
            .compute(300, true, true));
    }
}
```

Final Mutated Individual

4 Experiment

In this section, we systematically evaluate the performance of LegaTest by answering the following research questions.

- **RQ1:** Are the generated tests syntactically correct, compilable, and executable?
- **RQ2:** How adequate and effective are the generated test suites in terms of coverage?
- **RQ3:** How does the Iterative Evolution influence the quality and coverage of test suites?
- **RQ4:** Can the generated tests produce deep, semantically rich assertions beyond shallow ones, and to what extent does this enhance test quality?
- **RQ5:** What is the time and computational cost of the proposed approach compared to baseline techniques?

Benchmark and Class Extraction: We select Defects4J v2.0.1 [28] as our experimental benchmark due to its wide adoption and well-curated real-world Java bugs. To ensure representativeness and experimental feasibility, we use five projects—Math_5f, Lang_6f, Collections_25f, Compress_6f, and Csv_16f—covering diverse project sizes and defect types such as numerical errors, concurrency issues, null pointer exceptions, and parsing bugs. The project and version selection prioritizes

high citation frequency, defect diversity, Java 8 compatibility (EvoSuit only support Java 8), and non-overlapping functionality. Besides Defects4J, we additionally evaluate our approach on four Java projects—Commons-Cli, Commons-Csv, Ecommerce-Microservice, and Binance-Connector-originally used in the ChatUniTest study[9]. This supplementary benchmark is included to enable a direct comparison with ChatUniTest, one of our baselines, under identical experimental conditions.

Building on these projects, we construct our final benchmark subjects by applying a rigorous class extraction process to ensure that retained classes exhibit meaningful behavioral complexity rather than merely serving as utility or data-holding structures. Specifically, we focus on Java source files under the `src/main` directory, excluding paths containing `utils` or `enums`. Moreover, classes with names containing patterns such as `Utils`, `Helper`, `Test`, `DTO`, `VO`, `POJO`, `Entity`, `Bean`, `Model`, `Adapter`, `Factory`, or `Builder` are discarded, as they typically represent lightweight or auxiliary components. In addition, since LegaTest focus on test generation for complex classes, each class must satisfy the following conditions: (1) it contains at least two complex methods; (2) it includes at least three functionally meaningful methods; and (3) the proportion of getter/setter methods does not exceed 50%. Through this multi-stage filtering, we finally obtain 403 classes that capture semantically meaningful, behaviorally complex modules, providing a reliable and representative benchmark foundation for evaluating LegaTest.

Baseline and Configuration: To evaluate the effectiveness of LegaTest, we compare it against two baselines on the full benchmark: EvoSuite [17], a representative traditional SBST tool, and ChatUniTest [9], a recently proposed LLM-based test generation approach. EvoSuite serves as a strong baseline from conventional search-based testing, while ChatUniTest reflects the potential of state-of-the-art LLM-driven techniques. We note that ChatTester [78] follows a similar LLM-based paradigm as ChatUniTest, but empirical studies[70] have shown that its performance is generally inferior to ChatUniTest. To avoid redundant comparisons with methods of the same family, we select ChatUniTest as the representative and stronger LLM-based baseline.

Moreover, UTGen [12], a hybrid approach that combines EvoSuite-based test generation with LLM-driven refinement, is evaluated only on a randomly selected subset of 100 classes. Due to its substantially higher runtime and resource consumption, UTGen is not evaluated on the full benchmark. This design reflects practical computational constraints and is explicitly reported for transparency.

To ensure fair comparison, all tools are evaluated under consistent settings. For each run, LLM-based tools perform one generation round followed by up to two repair rounds if applicable. LegaTest, generates 10 test suites per class. For the baselines, we follow the default configuration of ChatUniTest to generate 10 test methods per focal method. UTGen is also configured to generate 10 test suites per class. EvoSuite is evaluated using its latest stable release (v1.0.6), which supports Java 8 and JUnit 4; all default settings are used, and it is also configured to generate 10 test suites per class. Besides, all experiments are conducted in a unified environment using project-specific Java versions (Java 8 or Java 11 as required), Maven, JUnit, and JaCoCo. Line, method, and branch coverage and pass rate are computed using the same evaluation scripts to ensure consistency across tools.

Implementation: LegaTest is implemented as a Python-based orchestration framework tailored for Maven-based Java projects, aiming to automatically generate high-quality unit tests that improve both diversity and effectiveness. The system leverages a dual-prompt strategy for initial test generation, which guides large language models to produce tests with deep assertions, high readability, and semantic alignment to the class under test (CUT). To further enhance coverage, the iterative optimization module integrates information across previously generated tests, complementing their strengths to exercise additional execution paths. Unlike existing approaches,

LegaTest can effectively generate tests not only for complex classes but also for abstract classes that carry practical testing value.

From a technical perspective, the framework incorporates both lightweight and fine-grained static analysis. Javalang is used to extract high-level structural information such as package declarations, imports, and complete method signatures (including names, parameters, and return types), providing an architectural overview of the CUT. Complementarily, Tree-sitter enables robust parsing at the method level to capture semantic elements such as invocations, variable references, constructor calls, and exception handling blocks, which together characterize the CUT's behavioral flow and external dependencies.

For LLM integration, we employ DeepSeek-Chat V3, an open-source chat-based model trained on a large-scale multi-language code corpus. It is selected for its strong reasoning capabilities and proficiency in generating code, while also providing an accessible and reproducible research setting. All model interactions are performed through its official API using default configurations. To ensure fairness, the baseline method is also evaluated using DeepSeek-Chat V3 under the same setup. In addition, we also evaluate our approach using GPT-5-mini on a randomly selected subset of 100 classes. This subset evaluation includes both test generation and iterative evolution. Due to the higher cost and resource consumption associated with GPT-based models, GPT-5-mini is not used for the full-benchmark evaluation. All baselines involved in this subset experiment are evaluated under the same model setting to ensure fairness.

Finally, LegaTest incorporates a lightweight repair module to ensure executability of generated tests by automatically handling common issues such as import conflicts, duplicate definitions based on compiler and runtime feedback. The complete system is orchestrated to follow a full lifecycle: initial test generation, iterative optimization, repair, and final compilation and execution. The implementation, along with datasets and evaluation scripts, will be released on GitHub to facilitate replication and future research.

4.1 Correctness and Executability of Generated Tests

This research question investigates whether the generated test suites are syntactically valid, compilable, and executable at runtime. We compare our approach (LegaTest) with two representative baselines: the LLM-based ChatUniTest and the search-based tool EvoSuite. The evaluation focuses on key correctness and executability metrics: (i) Successful Generation measures the proportion of classes for which at least one valid test case is generated. (ii) Assertion Errors measures the proportion of generated test cases containing failing assertions. (iii) Runtime Failures measures the proportion of test cases that fail during execution due to runtime exceptions, including NullPointerException, IndexOutOfBoundsException, ClassCastException, and other runtime errors. (iv) Successful Executions measures the proportion of test cases that run successfully without assertion or runtime failures.

Table 2 reports the results across nine benchmark projects and the aggregated totals. Overall, LegaTest consistently achieves the highest test generation capability, successfully producing tests for 376 out of 403 classes (93.3%), whereas ChatUniTest and EvoSuite succeed on only 153 (37.97%) and 136 (33.75%) classes, respectively. This demonstrates a clear advantage of our LLM-guided generation strategy in terms of breadth of applicability, as it can reliably cover almost all classes in the benchmark. Such generation capability is crucial, since only classes for which tests can be successfully generated can later benefit from repair and optimization.

Regarding execution reliability, EvoSuite achieves very low runtime failure rates (0.35%) and the highest successful execution ratio (99.29%). This is expected, as EvoSuite leverages well-engineered search-based heuristics tightly integrated with the Java ecosystem. However, this comes at the cost of limited generation success: it fails to generate tests for the majority of benchmark classes

Table 2. Experimental Results Comparison

Project Name	Method	Successful Generation	Assert Errors	Runtime Failure	Successful Executions	# Test Cases
Math_5f(162)	LegaTest (Initial)	**142 (87.65%)**	**1057 (7.07%)**	477 (3.19%)	**13426 (89.74%)**	14960
	ChatUniTest	22 (13.58%)	368 (14.32%)	**38 (1.48%)**	2164 (84.20%)	2570
	Evosuite	–	–	–	–	–
Collections_25f(136)	LegaTest (Initial)	**132 (97.06%)**	612 (3.96%)	326 (2.11%)	14530 (93.93%)	15468
	ChatUniTest	49 (36.03%)	1581 (9.14%)	**51 (0.30%)**	15659 (90.56%)	17291
	Evosuite	76 (55.88%)	**40 (0.49%)**	33 (0.41%)	**8056 (99.10%)**	8129
Lang_6f(38)	LegaTest (Initial)	**38 (100%)**	277 (5.09%)	47 (0.86%)	5116 (94.05%)	5440
	ChatUniTest	**38 (100%)**	1159 (8.39%)	37 (0.27%)	12623 (91.34%)	13819
	Evosuite	35 (92.11%)	**42 (0.45%)**	**2 (0.02%)**	**9253 (99.53%)**	9297
Compress_6f(15)	LegaTest (Initial)	**15 (100%)**	73 (5.97%)	162 (13.24%)	988 (80.79%)	1223
	ChatUniTest	14 (93.33%)	152 (8.12%)	93 (4.97%)	1626 (86.91%)	1871
	Evosuite	**15 (100%)**	**2 (0.08%)**	**4 (0.15%)**	**2581 (99.77%)**	2587
Csv_16f(2)	LegaTest (Initial)	**2 (100%)**	43 (10.51%)	14 (3.42%)	352 (86.07%)	409
	ChatUniTest	**2 (100%)**	30 (7.20%)	27 (6.47%)	360 (86.33%)	417
	Evosuite	**2 (100%)**	**0**	**0**	**506 (100%)**	506
Commons_Cli(9)	LegaTest (Initial)	**9 (100%)**	64 (5.48%)	51 (4.37%)	1063 (90.15%)	1178
	ChatUniTest	6(66.67%)	177 (14.32%)	**6 (0.49%)**	1053 (85.19%)	1236
	Evosuite	8 (88.89%)	**0**	41 (1.64%)	**2459 (98.36%)**	2500
Commons_Csv(2)	LegaTest (Initial)	**2 (100%)**	**48 (13.26%)**	**16 (4.42%)**	**298 (82.32%)**	362
	ChatUniTest	**2 (100%)**	152 (23.31%)	36 (5.52%)	464 (71.17%)	652
	Evosuite	–	–	–	–	–
Ecommerce(15)	LegaTest (Initial)	14(93.33%)	**17 (2.65%)**	165 (25.74%)	**459 (71.61%)**	641
	ChatUniTest	**15 (100%)**	161 (10.87%)	**353 (23.84%)**	967 (65.29%)	1481
	Evosuite	–	–	–	–	–
Binance(24)	LegaTest (Initial)	**22(91.67%)**	**89 (3.94%)**	**17 (0.75%)**	**2153 (95.31%)**	2259
	ChatUniTest	5 (20.83%)	67 (9.11%)	18 (2.45%)	650 (88.44%)	735
	Evosuite	–	–	–	–	–
Total	LegaTest (Initial)	**376 (93.30%)**	2280 (5.44%)	1275 (3.04%)	38385 (91.52%)	41940
	ChatUniTest	153 (37.97%)	3847 (9.60%)	659 (1.64%)	35566 (88.76%)	40072
	Evosuite	136 (33.75%)	**84 (0.36%)**	**80 (0.35%)**	**22855 (99.29%)**	23019

(only 33.75% coverage). In contrast, LegaTest achieves a much broader generation scope while still maintaining strong executability (91.52% successful executions), demonstrating that the generated tests are not only broadly applicable but also robust in practice. Compared to ChatUniTest, LegaTest yields both significantly higher generation success (93.3% vs. 37.97%) and lower error rates.

It is important to emphasize that the results reported here correspond to the initial version of LegaTest, evaluated across all 403 benchmark classes. Presenting these results establishes a comprehensive baseline of raw generation capability across the benchmark, which is critical for subsequent analysis: broad generation coverage is a prerequisite for any optimization, as only successfully generated tests can be further refined and enhanced. As shown in Table 2, LegaTest achieves the highest generation success across the full benchmark while maintaining robust executability and low error rates, demonstrating its overall superiority in automated test generation compared to the baselines.

Summary: LegaTest successfully generates tests for 376/403 classes (93.3%), significantly outperforming ChatUniTest (153/403, 37.97%) and EvoSuite (136/403, 33.75%). Execution success of generated tests is 91.52%, demonstrating both broad applicability and high robustness. Compared to the baselines, LegaTest achieves higher generation coverage and lower runtime failures, establishing its superiority in correctness and executability.

Table 3. Coverage Results of Full Benchmark Classes (Average Values)

Project Name	Method	Line Coverage (%)	Branch Coverage (%)	Method Coverage (%)	Avg. # Test Cases
Math_5f	LegaTest (Initial)	**57.28**	**47.59**	**64.74**	**11.17**
	ChatUniTest	4.99	3.85	6.08	11.69
	EvoSuite	–	–	–	–
Collections_25f	LegaTest (Initial)	**76.96**	**65.72**	**81.94**	**12.54**
	ChatUniTest	25.23	24.31	25.75	35.29
	EvoSuite	44.50	40.14	45.65	13.64
Lang_6f	LegaTest (Initial)	64.23	51.37	72.73	**14.50**
	ChatUniTest	60.47	53.35	58.80	36.37
	EvoSuite	**78.69**	**72.74**	**81.20**	30.39
Compress_6f	LegaTest (Initial)	69.50	48.68	**82.48**	**9.74**
	ChatUniTest	37.35	30.33	45.78	13.39
	EvoSuite	**70.65**	**68.02**	80.80	17.25
Csv_16f	LegaTest (Initial)	67.14	43.97	86.17	**13.20**
	ChatUniTest	66.87	50.64	62.95	20.85
	EvoSuite	**93.54**	**87.43**	**100.00**	25.30
Commons_Cli	LegaTest (Initial)	71.12	48.19	**77.11**	**13.59**
	ChatUniTest	45.21	25.91	44.35	28.35
	EvoSuite	**74.27**	**59.05**	76.20	31.97
Commons_Csv	LegaTest (Initial)	66.86	35.87	**80.72**	**18.10**
	ChatUniTest	**68.48**	**49.54**	67.27	32.6
	EvoSuite	–	–	–	–
Ecommerce	LegaTest (Initial)	71.49	–	77.98	**7.34**
	ChatUniTest	**72.21**	–	**78.84**	9.87
	EvoSuite	–	–	–	–
Binance	LegaTest (Initial)	**67.54**	**44.18**	**79.82**	**11.9**
	ChatUniTest	13.39	1.67	13.01	24.49
	EvoSuite	–	–	–	–
Total	LegaTest (Initial)	**66.95**	**54.37**	**74.16**	**11.94**
	ChatUniTest	38.31	29.65	39.50	26.82
	EvoSuite	58.66	53.41	61.27	19.60

4.2 Adequacy and Effectiveness of Test Suites

This research question investigates the adequacy and effectiveness of the generated test suites in terms of code coverage. Specifically, we evaluate line, branch, and method coverage, as well as the overall test suite size. For the initial assessment, we consider the raw test suites produced by LegaTest and compare them against state-of-the-art LLM-based tools (ChatUniTest) and the traditional search-based tool EvoSuite. By measuring coverage across all 403 benchmark classes, we provide a comprehensive view of the effectiveness of raw test generation, as summarized in Table 3. Considering that our benchmark contains 403 test classes in total, performing iterative optimization on all of them would incur excessive computational costs and is unnecessary for addressing the research question. Instead, we optimized about 100 test classes ($\approx$ 25% of the benchmark) following a balanced selection strategy. Specifically, when constructing the experimental dataset, we prioritized classes for which all tools successfully generated tests, and, whenever feasible, ensured that almost every package within each repository contributed at least one selected class. For packages with multiple candidate classes, we randomly sampled representative instances to cover diverse functionalities and structural characteristics. This approach enables broad coverage of testing scenarios while remaining computationally feasible. By striking a balance between efficiency and representativeness, the evaluation provides reliable evidence of the effectiveness of the genetic algorithm–based iterative optimization strategy without exhaustively optimizing all 403 classes in the benchmark.

Table 3 reports results on all 403 classes using the initial version of LegaTest, while Table 4 presents results on the selected 100 classes, where the iterative optimization strategy is applied

Table 4. Coverage Results of Selected 100 Classes with Iterative Optimization (Average Values). For LLM-based tools, results are reported under different LLM backends. EvoSuite is LLM-agnostic, thus its results are independent of the LLM choice.

Project Name	Method	Line Coverage (%)		Branch Coverage (%)		Method Coverage (%)		Avg. # Test Cases	
		DeepSeek	GPT	DeepSeek	GPT	DeepSeek	GPT	DeepSeek	GPT
Math_5f	LegaTest (Initial)	43.07	59.53	29.77	46.48	52.29	69.16	**13.75**	20.54
	LegaTest	**70.15**	**75.02**	**51.0**	**59.8**	**75.18**	**81.3**	30.75	21.63
	ChatUniTest	8.00	11.38	8.77	8.76	8.20	10.76	16.52	**19.28**
	EvoSuite	–	–	–	–	–	–	–	–
	UTGen	–	–	–	–	–	–	–	–
Collections_25f	LegaTest (Initial)	67.72	78.31	62.78	74.59	72.16	80.6	15.11	16.67
	LegaTest	**81.05**	**87.91**	**75.47**	**83.53**	**84.31**	**89.33**	26.14	19.96
	ChatUniTest	49.77	34.94	48.04	32.75	51.08	35.89	38.70	32.08
	EvoSuite	65.56	–	60.59	–	69.23	–	18.30	–
	UTGen	46.19	30.67	39.40	26.31	52.43	34.95	**11.39**	**9.87**
Lang_6f	LegaTest (Initial)	66.35	78.66	53.67	68.83	73.79	83.69	**14.58**	17.92
	LegaTest	**83.94**	**89.86**	71.81	**78.94**	**90.72**	**96.06**	26.1	24.07
	ChatUniTest	60.48	28.31	53.66	28.03	57.98	27.23	37.24	33.12
	EvoSuite	80.81	–	**74.71**	–	83.39	–	30.39	–
	UTGen	15.52	48.62	15.24	42.13	17.16	51.89	15.80	**7.91**
Compress_6f	LegaTest (Initial)	58.69	60.12	42.84	50.45	75.91	71.04	11.8	10.86
	LegaTest	**80.19**	**67.61**	**70.6**	58.23	**91.39**	**78.14**	29.15	15.69
	ChatUniTest	34.91	46.87	27.85	43.06	42.97	50.78	13.25	19.25
	EvoSuite	66.65	–	65.04	–	76.36	–	17.38	–
	UTGen	47.55	40.47	41.34	37.54	65.97	57.6	**8.77**	**9.83**
Csv_16f	LegaTest (Initial)	67.14	83.38	43.97	62.01	86.17	92.93	**13.20**	**13.35**
	LegaTest	89.10	93.16	68.43	74.14	99.62	99.58	39.05	21.3
	ChatUniTest	66.87	62.45	50.64	46.03	62.95	64.21	20.85	15.65
	EvoSuite	**93.54**	–	**87.43**	–	**100.00**	–	25.30	–
	UTGen	90.09	48.08	86.52	43.13	96.53	50	22.42	19
Commons_Cli	LegaTest (Initial)	66.43	68.71	48.39	47.93	77.72	78.69	**14.17**	**14.64**
	LegaTest	**82.60**	**84.29**	67.79	70.13	**89.81**	**92.03**	22.23	22.5
	ChatUniTest	66.04	55.52	49.98	42.39	61.16	52.56	27.67	21.39
	EvoSuite	81.65	–	**71.53**	–	87.70	–	41.77	–
	UTGen	–	–	–	–	–	–	–	–
Commons_Csv	LegaTest (Initial)	66.86	64.23	35.87	36.53	80.72	71.63	**18.10**	**14.28**
	LegaTest	**87.12**	**82.55**	**55.49**	**53.57**	**97.83**	**88.47**	28.80	23.01
	ChatUniTest	68.48	34.13	49.54	25.00	67.27	39.20	32.6	26.25
	EvoSuite	–	–	–	–	–	–	–	–
	UTGen	–	–	–	–	–	–	–	–
Ecommerce	LegaTest (Initial)	88.73	83.6	–	–	89.22	84.18	**7.39**	**8.35**
	LegaTest	**100**	**100**	–	–	**100**	**100**	14.93	13.91
	ChatUniTest	82.39	7.69	–	–	83.93	8.92	9.9	11.70
	EvoSuite	–	–	–	–	–	–	–	–
	UTGen	–	–	–	–	–	–	–	–
Binance	LegaTest (Initial)	14.34	73.96	7.47	46.63	16.82	81.59	9.71	**12.97**
	LegaTest	**21.88**	**75.72**	**10.82**	**49.6**	**23.06**	**88.47**	12.21	19.23
	ChatUniTest	9.58	28.15	8.57	0	8.56	27.11	**6.33**	24.70
	EvoSuite	–	–	–	–	–	–	–	–
	UTGen	–	–	–	–	–	–	–	–
Total	LegaTest (Initial)	62.11	73.21	49.38	62.74	70.61	79.2	13.87	16.15
	LegaTest	**79.91**	**84.05**	**67.75**	**72.90**	**86.11**	**89.82**	26.52	20.98
	ChatUniTest	47.89	31.82	41.53	29.22	48.27	32.22	29.26	27.53
	EvoSuite	74.01	–	68.93	–	78.46	–	24.69	–
	UTGen	33.03	41.51	29.52	36.41	39.43	47.73	**13.26**	**9.17**

to obtain the final version of the tool. This design ensures that the baseline evaluation covers the entire benchmark, while the optimization-focused analysis provides a deeper view of the tool's full capabilities. The "Total" row reports weighted averages across all benchmark classes, where each project's coverage is weighted by its number of successfully covered classes. This avoids smaller projects disproportionately influencing the overall results. For *Ecommerce-Microservice*, branch coverage is reported as "–" because the selected classes do not contain explicit branch constructs. These classes mainly implement thin service-layer logic that delegates functionality to

external components without introducing conditional or loop-based control flow. As a result, branch coverage is not applicable for this project and is excluded from the weighted average reported in the "Total" row. For *Math_5f*, EvoSuite results are unavailable because this project heavily relies on complex numerical routines (e.g., special functions such as Beta, Gamma, and Erf), extensive use of abstract classes and generic types, and large static initializations. These characteristics make it difficult for EvoSuite's search-based strategy to synthesize meaningful inputs: randomly generated values often violate strict mathematical domains, abstract or generic classes cannot be directly instantiated, and heavy static initialization frequently leads to timeouts during analysis. As a result, EvoSuite was unable to produce valid test suites for the majority of classes in this project. For the additional projects adopted from the ChatUniTest benchmark, EvoSuite also fails to generate valid test suites due to practical incompatibilities with project environments and dependencies. For *Commons-Csv*, the failure is caused by dependency incompatibility with the H2 database library, which is distributed as a multi-release JAR and compiled with newer Java versions. EvoSuite v1.0.6 relies on an outdated bytecode analysis library and cannot correctly parse such class files, leading to bytecode analysis errors during test generation. For *Ecommerce-Microservice* and *Binance-Connector*, EvoSuite fails because these projects require Java 11 or higher, whereas EvoSuite v1.0.6 only supports Java 8 bytecode, preventing successful class analysis and instrumentation.

Since UTGen builds upon EvoSuite by first generating tests and then refining them using LLM-based techniques, it inherits EvoSuite's limitations. Consequently, UTGen also fails to generate test suites for *Commons-Csv*, *Ecommerce-Microservice*, and *Binance-Connector*. Moreover, UTGen incurs substantial computational overhead during refinement. To ensure experimental feasibility, we impose a 30-minute timeout per class. Under this constraint, although EvoSuite can generate initial tests for *Commons-Cli*, UTGen fails to complete refinement within the time limit, resulting in unsuccessful runs for this project.

Overall, the results show that even the initial version of LegaTest already achieves an excellent balance between coverage and efficiency. On projects such as *Lang_6f*, *Compress_6f*, and *Csv_16f*, EvoSuite does obtain higher raw coverage, but only by generating substantially larger test suites. In contrast, LegaTest consistently delivers far smaller test suites—often less than half the size of EvoSuite's and ChatUniTest's—while still maintaining competitive or superior coverage. Notably, in *Math_5f* and *Collections_25f*, LegaTest achieves the highest coverage across all three metrics with the fewest test cases, underscoring both its effectiveness and efficiency. Even in settings where EvoSuite yields higher absolute coverage (e.g., *Csv_16f*), LegaTest still achieves strong method coverage (86.17%) with nearly half the number of test cases (13.20 vs. 25.30), highlighting its practicality for concise yet effective test generation.

When aggregated across the full 403-class benchmark, the initial version of LegaTest already attains the highest overall averages: line coverage (66.95%), branch coverage (54.37%), and method coverage (74.16%). These results significantly outperform ChatUniTest (38.31%, 29.65%, 39.5%) and also surpass EvoSuite (58.66%, 53.41%, 61.27%). Importantly, this superior coverage is reached with the smallest test suites on average (11.94 cases), which is 39% fewer tests than EvoSuite (19.60) and 55% fewer than ChatUniTest (26.82). Thus, even without optimization, LegaTest demonstrates clear advantages in both effectiveness and efficiency.

To assess the impact of the iterative optimization component—which completes the design of LegaTest —Table 4 reports results on the selected 100 classes under two LLM backends, namely DeepSeek and GPT. As a supplementary reference, we additionally include a raw zero-shot GPT-5-mini baseline on the selected subset. Since our main comparisons focus on task-specific baselines, we report its detailed results in Appendix A. The comparison between the initial and optimized versions reveals consistent and substantial improvements across all projects for both backends. For instance, in *Compress_6f*, line coverage increases from 58.69% to 80.19% under DeepSeek and from

60.12% to 67.61% under GPT, and method coverage from 75.91% to 91.39% and from 71.04% to 78.14%, respectively, while the average number of test cases grows from 11.8 to 29.15 (DeepSeek) and from 10.86 to 15.69 (GPT). Similarly, in *Lang_6f*, method coverage improves from 73.79% to 90.72% under DeepSeek and from 83.69% to 96.06% under GPT, outperforming ChatUniTest (57.98%/27.23%), EvoSuite (83.39%) and UTGen (17.16%/51.89%). In *Csv_16f*, LegaTest achieves 99.62% method coverage under DeepSeek and 99.58% under GPT, very close to EvoSuite's 100.00%.

Aggregated over the 100 optimized classes, the full version of LegaTest improves average line coverage from 62.11% to 79.91% under DeepSeek and from 73.21% to 84.05% under GPT, branch coverage from 49.38% to 67.75% and from 62.74% to 72.9%, respectively, and method coverage from 70.61% to 86.11% and from 79.2% to 89.82%. Compared with UTGen, which produces the smallest test suites on average (13.26 cases under DeepSeek and 9.17 under GPT), LegaTest delivers substantially higher coverage across all metrics. Specifically, UTGen attains only 33.03%/41.51% line coverage, 29.52%/36.41% branch coverage, and 39.43%/47.73% method coverage under DeepSeek/GPT, whereas the optimized LegaTest nearly doubles these values while maintaining a practical test suite size. Although the test suite size grows moderately (13.87 $\rightarrow$ 26.52 for DeepSeek and 16.15 $\rightarrow$ 20.98 for GPT), it remains smaller than ChatUniTest (29.26/27.53) and competitive with EvoSuite (24.69). These results confirm that the genetic algorithm–based iterative optimization strategy is a crucial enhancement: it not only boosts coverage substantially but also preserves the compactness and practicality of the generated suites across different LLM backends.

Statistical Significance Analysis. To further assess whether the observed improvements are statistically significant, we perform paired statistical tests on the same set of 100 selected classes under a unified evaluation protocol.

First, we analyze whether LegaTest significantly outperforms the baselines in successful class-level result generation. Each class is treated as a binary outcome indicating whether the tool successfully generates class-level coverage results, and McNemar's exact test is applied. McNemar's test is a non-parametric test for paired nominal data that evaluates whether two methods differ significantly on the same instances [44].

Table 5 summarizes the results. Here, each discordant pair indicates a class for which only one of the two compared tools successfully produces class-level results. Under DeepSeek, the discordant pair counts are 16 / 2 against ChatUniTest and 58 / 1 against UTGen, yielding exact p-values of 1.31×10^{-3} and 2.08×10^{-16}, respectively. Under GPT, the discordant pair counts are 42 / 0 against ChatUniTest and 53 / 0 against UTGen, with exact p-values of 4.55×10^{-13} and 2.22×10^{-16}. These results indicate that LegaTest significantly outperforms both baselines in successful class-level result generation.

Second, we examine whether LegaTest achieves significantly higher per-class coverage values. For each class, we use its coverage score (i.e., line, branch, and method coverage), assigning zero when a tool fails to generate effective tests. We then apply paired t-tests, which assess whether the mean difference between two paired samples is significantly different from zero [64].

Table 6 reports the results. Under both DeepSeek and GPT backends, LegaTest achieves significantly higher line, branch, and method coverage than both baselines, with all p-values below 0.001.

Overall, these results consistently demonstrate that the improvements of LegaTest are statistically significant across different backends and evaluation metrics, and are highly unlikely to be due to random variation.

Table 5. Statistical significance analysis using McNemar's exact test on successful class-level result generation.

Backend	Comparison	Discordant Pairs (Ours / Baseline)	p-value
DeepSeek	vs ChatUniTest	16 / 2	1.31×10^{-3}
DeepSeek	vs UTGen	58 / 1	2.08×10^{-16}
GPT	vs ChatUniTest	42 / 0	4.55×10^{-13}
GPT	vs UTGen	53 / 0	2.22×10^{-16}

Table 6. Paired t-test results (p-values) on per-class coverage values.

Backend	Comparison	Line	Branch	Method
DeepSeek	vs ChatUniTest	2.29×10^{-8}	4.65×10^{-6}	3.25×10^{-10}
DeepSeek	vs UTGen	1.65×10^{-22}	4.91×10^{-17}	7.24×10^{-21}
GPT	vs ChatUniTest	5.74×10^{-27}	3.13×10^{-22}	4.82×10^{-30}
GPT	vs UTGen	8.25×10^{-25}	9.26×10^{-20}	5.25×10^{-23}

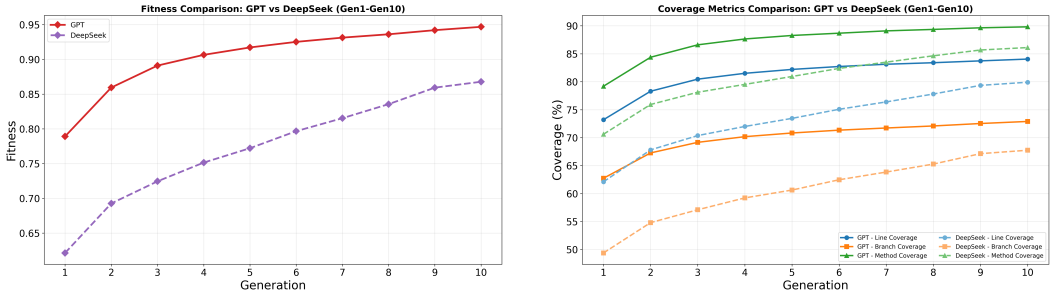
Summary: Across the 403 benchmark classes, LegaTest achieves 66.95% line coverage, 54.37% branch coverage, and 74.16% method coverage, surpassing ChatUniTest (38.31%, 29.65%, 39.50%) and EvoSuite (58.66%, 53.41%, 61.27%). Average test suite size is 11.94, 39% smaller than EvoSuite (19.6) and 55% smaller than ChatUniTest (26.82). After iterative optimization on 100 classes, LegaTest improves line/branch/method coverage from 62.11/49.38/70.61% → 79.91/67.75/86.11% with DeepSeek, and from 73.21/62.74/79.2% → 84.05/72.90/89.82% with GPT, with corresponding test suite sizes increasing from 13.87 → 26.52 and from 16.15 → 20.98. These results indicate that LegaTest is effective across different LLM backends.

4.3 Impact of Iterative Evolution

We examine the role of iterative evolution in enhancing the quality and adequacy of automatically generated test suites using the 100 classes introduced in the previous section, under two LLM backends: DeepSeek and GPT. The experimental setup of Iterative Evolution is as follows: the population size N is set to 10, and the maximum number of generations (*maxGen*) is 10. These settings are fixed across all runs. The evolution of each class terminates when either the maximum number of generations is reached or both branch and method coverage exceed 98%. Additionally, when branch coverage reaches 95%, mutation operations are skipped, since the primary purpose of mutation is to explore and cover previously uncovered branches, and further mutations beyond this point yield limited additional coverage. This configuration ensures a balance between computational efficiency and effective evolutionary refinement.

In particular, we investigate how successive generations influence test performance with respect to line, branch, and method coverage, as well as the overall fitness score, for both LLM backends. By systematically analyzing the evolutionary trends across generations, we aim to evaluate the effectiveness of the proposed genetic algorithm–based iterative optimization strategy and demonstrate its capacity to incrementally refine test quality through continuous iterative improvement.

The experimental results are presented in Figures 2a and 2b. Figure 2a illustrates the evolution of fitness across generations for both DeepSeek and GPT. For both backends, fitness shows a clear and consistent upward trend from Gen1 to Gen10, indicating steady improvement through iterative evolution. With DeepSeek, the fitness increases from 0.62 at Gen1 to 0.87 at Gen10, while GPT exhibits a similar trend, improving from 0.79 to 0.95. In both cases, the most notable improvements occur in the early generations (Gen1–Gen4), followed by more gradual but consistent growth in later stages. This pattern indicates that iterative optimization based on genetic algorithms effectively



(a) Evolution of Fitness (b) Coverage Trends

Fig. 2. Comparison of Fitness and Coverage Trends Across Generations

enhances the adaptability of the generated test suites, with evolutionary refinement maintaining its positive impact throughout the process.

Figure 2b reports the trends of line, branch, and method coverage over the same generations, again for both LLM backends. All three coverage metrics demonstrate monotonic improvement as evolution proceeds. Under DeepSeek, line coverage increases from 62.11% at Gen1 to 79.91% at Gen10, branch coverage from 49.38% to 67.75%, and method coverage from 70.61% to 86.11%. Under GPT, the corresponding improvements are from 73.21% to 84.05% (line), 62.74% to 72.90% (branch), and 79.20% to 89.82% (method). These results suggest that iterative evolution not only boosts overall fitness but also systematically improves coverage at multiple levels of granularity.

In summary, the findings confirm the effectiveness of the proposed genetic algorithm–based iterative optimization strategy across different LLM backends. The progressive improvements in both fitness and coverage metrics across generations validate the ability of iterative evolution to enhance the quality and adequacy of test suites in a sustained manner. Compared to the baseline performance at Gen1, the test suites at Gen10 achieve substantial relative improvements under DeepSeek, with line, branch and method coverage increasing by 28.7%, 37.2%, and 22%, respectively, while overall fitness improves by 39.7%. Under the GPT backend, similar upward trends are observed, with relative improvements of 14.8% in line coverage, 16.2% in branch coverage, 13.4% in method coverage, and 20% in overall fitness. These results highlight the sustained benefits of iterative evolution and indicate that its effectiveness is robust across different LLM backends.

Summary: Iterative evolution over 10 generations increases line, branch, and method coverage by 28.7%, 37.2%, and 22%, respectively, under the DeepSeek backend, and by 14.8%, 16.2%, and 13.4% under the GPT backend, compared to Gen1; overall fitness improves by 39.7% and 20%, respectively. These results confirm that the genetic algorithm–based refinement improves the quality and adequacy of test suites, and that the effectiveness of our approach is largely independent of the underlying LLM. This continuous improvement validates the synergy between the global exploration of crossover and the local refinement of mutation, confirming that the complete evolutionary strategy is essential for achieving high coverage.

4.4 Assertion Quality and Expressiveness

To evaluate the quality of generated assertions, we introduce a three-level taxonomy of assertion complexity:

- **Level 1 (Basic assertions)** covers simple truth-value checks or direct equality comparisons, such as `assertTrue(result)` or `assertEquals(x, 5)`, representing the most superficial level of validation.
- **Level 2 (Property assertions)** involves verifying object properties, collection states, or relational expressions, such as `.size()`, `.isEmpty()`, `.contains()`, or comparisons like `count != 0` and `value >= threshold`.
- **Level 3 (Complex assertions)** captures deeper semantic validations, including two main aspects: (i) structural depth, such as method call chains with a depth greater than two (e.g., `obj.getChild().getParent().getName()`) or use of `assertThrows` for exception handling, and (ii) domain-specific semantic relevance, indicated by the presence of keywords reflecting meaningful program states or critical properties (e.g., *status*, *role*, *permission*, *token*, *password*, *secure*, *order*, *exception*, *error*). These keywords are chosen based on common software domain practices, where such terms often correspond to important business logic, security checks, or critical program invariants. While not exhaustive, their presence serves as a practical proxy for semantic depth, helping to distinguish assertions that go beyond superficial value checks.

In addition to this categorical classification, we assess two complementary metrics: average method-call depth, which reflects the structural complexity of assertions, and average object count, which measures the number of distinct semantic entities referenced in assertions. Together, these metrics provide a quantitative perspective on the richness and expressiveness of the generated tests.

To ensure fairness, we selected the same set of test classes for all tools, such that ChatUniTest, EvoSuite, and both the initial and optimized versions of LegaTest were evaluated on an identical basis. UTGen is not included in this assertion quality analysis. As reported elsewhere in the paper, UTGen is evaluated only on a randomly selected subset of 100 classes due to its substantially higher runtime and resource consumption. Including UTGen here would require restricting all tools to the intersection of this subset, resulting in a significantly reduced sample size and undermining the robustness and statistical reliability of the comparison. Therefore, we exclude UTGen from this analysis to ensure fairness and stability of the results. Figure 3 presents the results as a normalized stacked bar chart combined with bubble visualization: the stacked bars represent the distribution of assertions across Levels 1–3, while the bubble size encodes the average object count and the color encodes the average method-call depth. This design highlights both the categorical distribution of assertion types and the structural-semantic complexity achieved by different tools. The results show that LegaTest consistently outperforms the baselines in producing deeper and more expressive assertions. EvoSuite's assertions are dominated by basic checks (52.7% at Level 1), with only 2.2% falling into Level 3, indicating shallow validation. ChatUniTest performs slightly better, but still relies heavily on basic assertions (51.6% Level 1) with only 15.5% at Level 3. In contrast, the initial version of LegaTest already demonstrates a substantial advantage, achieving 26.7% Level 3 assertions, the highest among all tools, alongside the greatest average method-call depth (1.49) and object count (3.28). After iterative optimization, LegaTest maintains a balanced distribution with 46.1% Level 1, 35.2% Level 2, and 18.7% Level 3 assertions, reflecting a deliberate trade-off: iterative evolution aims to enhance coverage, reduce runtime failures, and improve overall test stability while preserving the semantic richness of assertions. Highly complex Level 3 checks remain valuable for deep semantic validation, but the algorithm retains a mix of more robust Level 1 and Level 2 assertions to ensure that tests execute reliably across diverse inputs. This adjustment improves the practical executability and reliability of the test suites, while still maintaining meaningful Level 3 validations that contribute to the expressiveness and fault-detection capability of the generated tests. while still surpassing EvoSuite (0.84/2.28) and ChatUniTest (1.14/2.91) in average depth (1.25) and object richness (3.14). These results indicate that LegaTest not only generates a larger proportion of

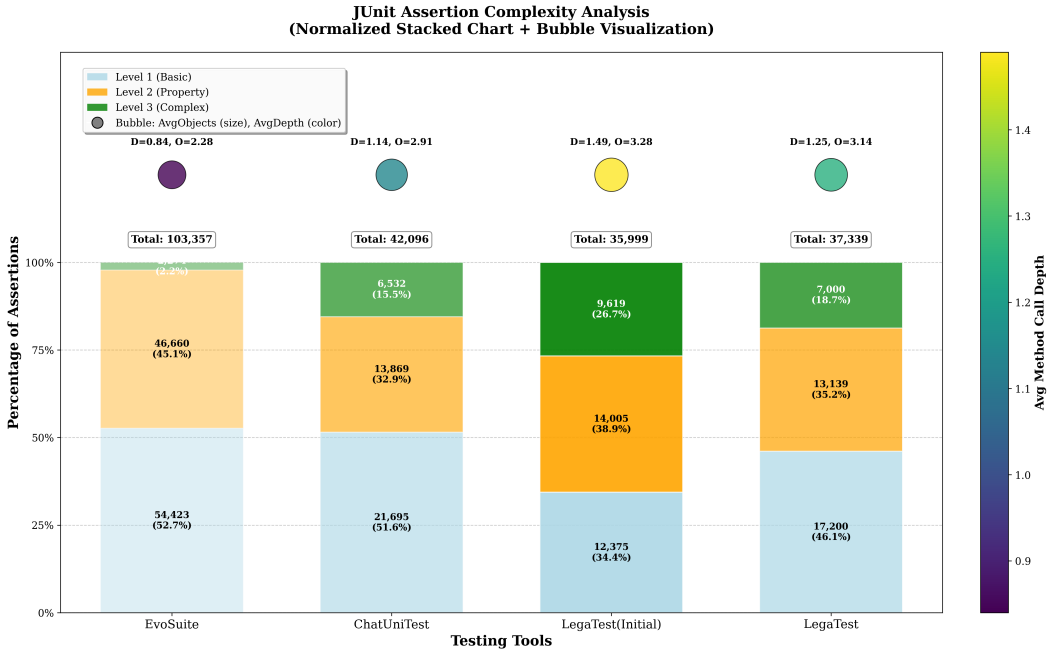


Fig. 3. Assertion Complexity Levels and Structural Metrics Comparison

semantically rich, deep assertions, but also sustains a balanced and meaningful assertion distribution after repair and optimization. Overall, the advantage of LegaTest in assertion quality directly translates into stronger semantic expressiveness and higher reliability of the generated test suites.

Mutation Testing Analysis. To further validate the effectiveness of the generated assertions beyond syntactic correctness and structural complexity, we conduct a mutation testing analysis using PiTest[11] to ensure consistency with the assertion quality analysis above, this experiment is performed on the same set of complex classes. The mutation scope is explicitly configured to target only these classes, allowing us to directly verify whether the structurally complex assertions (Levels 2 and 3) translate into actual fault-detection capability.

The analysis generated a total of 5,401 mutants across the target classes. The test suites generated by LegaTest successfully detected (covered) 3,277 mutants, out of which 2,518 were killed. Consequently, the tool achieved an overall Mutation Score of 46.62% and a notably high overall Test Strength of 76.84%.

The divergence between these two metrics provides important context regarding the tool's performance. The Mutation Score (46.62%) is constrained by the intrinsic difficulty of the selected test targets. Since we specifically evaluated semantically complex classes, achieving full path coverage is inherently challenging; many mutants reside in deep execution paths that are difficult to reach within standard execution time limits, leading to coverage gaps or timeouts during mutation analysis. However, the critical metric for evaluating assertion quality is Test Strength, which measures the ratio of killed mutants strictly within the covered code paths ($killed/(killed + survived)$). A Test Strength of 76.84% demonstrates that once a code path is exercised, the accompanying assertions are highly effective at constraining program behavior. This confirms that the high proportion of

Level 3 assertions in LegaTest is not merely ornamental; they impose rigorous semantic checks that successfully distinguish between correct and mutated behaviors on covered paths.

Summary: LegaTest produces 18.7% Level 3 (complex) assertions after optimization, outperforming ChatUniTest (15.5%) and EvoSuite (2.2%). Average method-call depth is 1.25 and average object count is 3.14, exceeding the baselines(0.84/2.28 and 1.14/2.91). These results demonstrate that LegaTest generates semantically richer and structurally deeper assertions. This structural advantage is validated by mutation testing on the same dataset, where LegaTest achieves a Test Strength of 76.84%. While the Mutation Score (46.62%) reflects the challenge of fully covering these complex classes, the high Test Strength confirms that the generated assertions impose effective semantic constraints, ensuring strong fault-detection capability wherever coverage is achieved.

4.5 Performance and Computational Efficiency

This section evaluates the computational efficiency of LegaTest in terms of runtime and token consumption, and compares it with representative baseline tools, including EvoSuite, ChatUniTest, and UTGen. We first present a detailed analysis of LegaTest's internal cost across different stages, followed by an aggregate comparison with baseline approaches.

Efficiency of LegaTest. We analyze the computational cost of LegaTest by decomposing its workflow into two main phases: (i) the *initial generation* phase, which constructs the first population of test suites from source code, and (ii) the *iterative evolution* phase, which applies crossover/mutation-based generation to improve coverage.

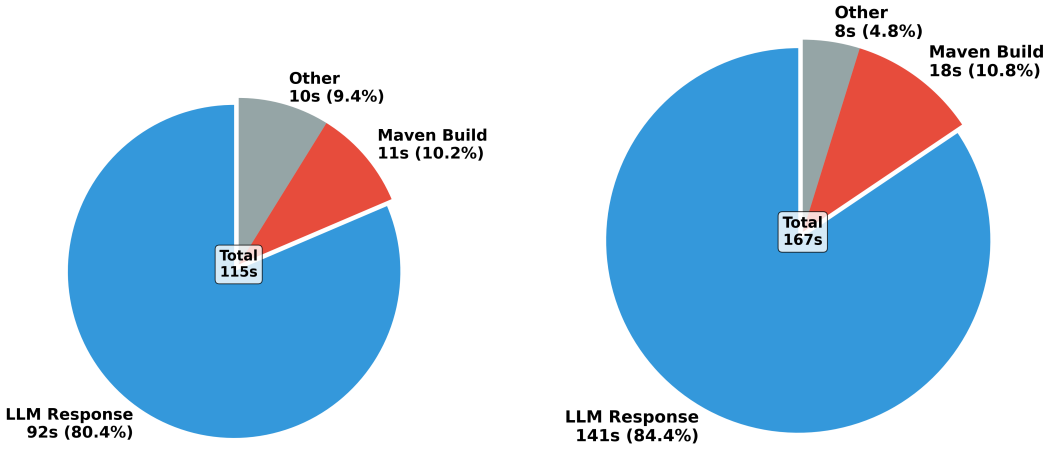
Figure 4 illustrates the runtime distribution and time breakdown for these two phases. During the initial generation phase, generating one test suite takes around 115 seconds on average, with LLM inference dominating the cost (approximately 80.4%), followed by Maven compilation (10.2%) and other overheads (9.4%), including file I/O, test parsing, and intermediate result processing. In the iterative evolution phase, the average runtime increases to around 167 seconds per evolved test suite, primarily due to more complex LLM-driven evolution optimization.

Figure 5 reports the corresponding token consumption for the two phases, further decomposed into input and output tokens. For GPT-5-mini, reasoning-related tokens are included in the output token counts. For the initial generation phase, each test suite consumes around 13.8K tokens on average, including roughly 8.8K input tokens and 5K output tokens, where generation accounts for approximately 72% of the total token usage and repair for 28%. In contrast, the iterative evolution phase consumes around 21.2K tokens per test suite on average, with a more balanced distribution between crossover/mutation (51.2%) and repair (48.8%), reflecting the increased complexity of repairing newly synthesized tests.

Overall, the computational cost of LegaTest is dominated by LLM inference. At the test-suite level, initial generation requires around 115 seconds and 13.8K tokens per suite, while iterative evolution requires around 167 seconds and 21.2K tokens per evolved suite. Since LegaTest adopts a population-based evolutionary strategy, the total cost at the class level depends on the configured population size and number of generations.

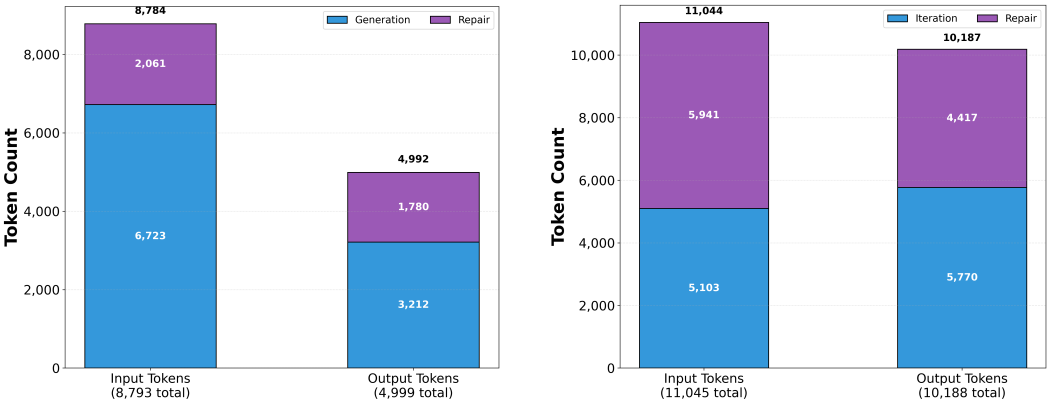
Comparison with Baselines. We further compare LegaTest with baseline tools at an aggregate level, focusing on average runtime per class and total token consumption per class. Figure 6 summarizes the comparison.

EvoSuite operates under a default 60-second time budget per class and does not involve LLM calls, resulting in the lowest runtime and zero token consumption. ChatUniTest exhibits a similar average



(a) Initial generation phase: runtime breakdown (b) Iterative evolution phase: runtime breakdown

Fig. 4. Time consumption of LegaTest across different phases.

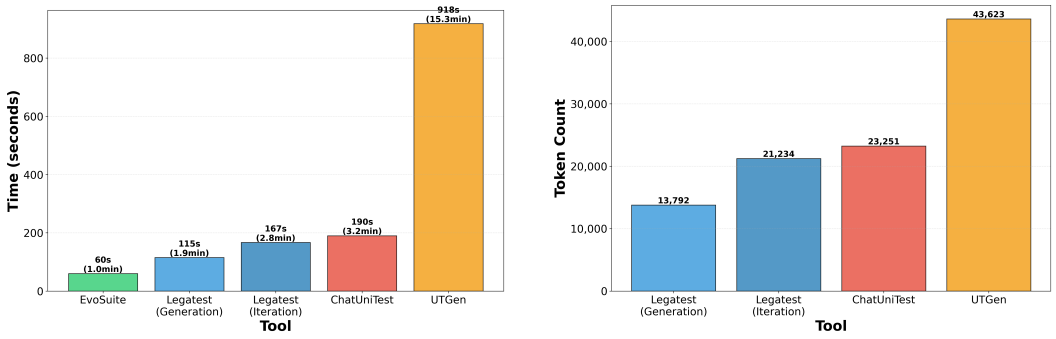


(a) Initial generation phase: token consumption (b) Iterative evolution phase: token consumption

Fig. 5. Token consumption of LegaTest, decomposed into input and output tokens.

runtime to LegaTest (around 190.2 seconds per class) with comparable token usage (approximately 23K tokens per class). In contrast, UTGen incurs substantially higher computational cost, requiring 918 seconds (15.3 minutes) per test suite and consuming 43.6K tokens per suite on average. This overhead is mainly attributed to its recursive retry mechanism, strict multi-stage validation, and fine-grained per-method processing.

In summary, LegaTest achieves a balanced trade-off between computational efficiency and test generation capability. Compared to traditional SBST tools, it introduces additional LLM-related cost, while remaining significantly more efficient than refinement-heavy LLM-based approaches such as UTGen. This makes LegaTest practical for iterative, coverage-driven unit test generation.



(a) Average runtime per class

(b) Average token consumption per class

Fig. 6. Efficiency comparison between LegaTest and baseline tools.

Summary: LegaTest incurs moderate LLM-driven overhead while maintaining practical efficiency. Initial generation costs around 115 seconds and 13.8K tokens per test suite, and iterative evolution costs around 167 seconds and 21.2K tokens per evolved suite. Compared with baselines, LegaTest is substantially more efficient than refinement-heavy approaches such as UTGen, while providing richer test generation than fixed-budget SBST tools like EvoSuite.

5 Threats to Validity

External Validity. The main threats to external validity relate to the generalizability of our results. Our evaluation covers multiple Java projects, including five from Defects4J and four additional real-world projects adopted from prior work. The Defects4J subset was selected to balance representativeness, diversity of defect types, and experimental feasibility. Specifically, exclusions were driven by two factors: (1) Baseline Stability and Benchmarking Standards: We distinguished between projects presenting *algorithmic challenges* versus those with *systemic environmental incompatibility*. We excluded projects where the baseline tool (EvoSuite) exhibits systemic failures due to architectural conflicts (e.g., *Mockito*'s bytecode manipulation or *Chart*'s GUI dependencies) or excessive scale (e.g., *Closure*). Conversely, we retained projects like *Math*—despite baseline difficulties—to serve as algorithmic benchmarks, aligning with standard practices in automated test generation literature; and (2) Domain Redundancy: We prioritized distinct defect types over repetitive domains (e.g., excluding redundant XML/JSON parsers like *Jackson* or *Gson*) to optimize the computational budget of the iterative LLM-GA process. Similarly, while evaluating across multiple programming languages would further strengthen generalizability, our current implementation of LegaTest, the baseline tools, and available benchmarks are all Java-based. Although the overall workflow of LegaTest is language-agnostic in design, the results may not directly generalize to other programming languages or arbitrary Java projects. Extending the approach to broader contexts remains a direction for future work.

To mitigate the threat of model dependency, we additionally evaluated LegaTest using a multi-model setup, incorporating a different LLM backend (i.e., gpt-5-mini). Experiments on a representative subset of 100 classes show consistent trends across models, indicating that the effectiveness of LegaTest is largely independent of the specific choice of LLM.

Internal Validity. Threats to internal validity concern the experimental implementation and causal relationships. The framework incorporates stochastic components, such as LLM-based

generation and Genetic Algorithm (GA)–driven optimization. To mitigate nondeterministic results, we performed repeated runs and quality checks.

Regarding the evolutionary components, crossover and mutation serve distinct but complementary roles within LegaTest: crossover drives global exploration by combining test structures, while mutation is explicitly designed for *local exploitation*—fine-tuning elite individuals to reach specific uncovered branches. We did not conduct a separate ablation study on mutation because, consistent with evolutionary computation theory, this operator is fundamental for preventing premature convergence. Removing mutation would structurally compromise the framework's ability to refine tests; therefore, its inclusion is justified by its intrinsic theoretical role in sustaining the iterative optimization process described in RQ3.

Another potential issue is the computational cost comparison. While we provide detailed measurements of runtime and token usage, the experiments do not enforce a unified wall-clock time budget across tools. This design choice is motivated by the fundamentally different execution models of SBST-based tools and LLM-based approaches. Strict time limits appropriate for SBST might interrupt semantically complete LLM generation steps, introducing reproducibility issues. Future work may explore standardized time-budget evaluations to further strengthen fairness.

Construct Validity. Threats to construct validity relate to the suitability of our evaluation metrics and experimental design.

First, regarding the use of "fixed" versions: We evaluated LegaTest on the fixed versions of Defects4J projects, treating them as the ground truth. This design explicitly models a regression testing scenario[49, 76], where the objective is to capture the stable behavior of the current code to prevent future regressions. We acknowledge that this assumption implies LegaTest may generate tests that enforce existing bugs as correct behavior (if they are present in the fixed version). However, prioritizing the creation of a high-coverage "safety net" for software evolution over latent bug discovery is a standard trade-off in regression test generation research[16, 49]. Therefore, our results reflect the tool's effectiveness in generating robust regression suites rather than its capability to discover new faults in existing code.

Second, regarding assertion quality, the optimization process may slightly reduce the proportion of highly complex Level 3 assertions. This reflects a deliberate design trade-off: maintaining a balanced mix of Level 1 (basic) and Level 2 (property) assertions improves test robustness and executability during evolution. Importantly, the generated test suites still preserve a significant portion of meaningful Level 3 checks, ensuring that semantic expressiveness is not sacrificed for stability.

Third, to validate that these assertions effectively constrain program behavior, we employed mutation testing. A potential threat lies in the use of a subset of complex classes (identical to those used in the assertion analysis) and the resulting Mutation Score (46.62%). This score is constrained by the intrinsic complexity of the target classes, which leads to unavoidable timeouts and coverage gaps during mutation analysis. To mitigate this threat and ensure a valid assessment of assertion quality, we rely on Test Strength (76.84%) as the primary indicator. By measuring the kill ratio strictly within covered paths, Test Strength isolates the effectiveness of the assertions from coverage limitations, confirming that LegaTest generates assertions capable of detecting actual faults when the code is exercised.

6 Related works

6.1 Search Based Software Testing (SBST)

Search-based software testing (SBST) employs metaheuristic algorithms to automatically generate test cases, reducing manual effort and uncovering boundary values and exceptional inputs. A

variety of tools embody this approach: EvoSuite [17], one of the most widely adopted tools, employs evolutionary algorithms to generate JUnit test suites aimed at maximizing coverage. Randoop [48] applies a feedback-directed random strategy, incrementally constructing sequences from previously executed, non-failing statements. Sapienz [43], developed at Meta, applies multi-objective SBST to large-scale Android app testing, optimizing for crash detection, coverage, and test efficiency. Pynguin [41] is a search-based unit test generation framework for Python that leverages evolutionary algorithms and supports dynamically typed code. Stoat [65] performs stochastic model-based testing for Android apps by combining dynamic analysis, static analysis, and Gibbs sampling to guide UI exploration.

Despite significant progress, SBST still faces notable limitations. Existing methods often generate boundary-oriented tests with trivial assertions, require costly regeneration after minor code changes, and struggle in dynamically typed contexts due to reliance on type inference [42]. Complex constructs—such as nested builders, lambda expressions, generics, or abstract classes—frequently cause compilation or instrumentation failures, while the stochastic nature of search yields tests with low readability and maintainability. To address these issues, our tool LegaTest integrates semantic cues and uncovered code paths to guide test generation. It produces richer assertions and systematically covers challenging scenarios, including generics, abstract classes, and nested lambdas. By enforcing BDD-style `@DisplayName` conventions and the Arrange-Act-Assert pattern, it enhances readability and maintainability, ultimately delivering more accurate and practical test cases than conventional SBST techniques.

6.2 LLM-Based Unit Test Generation

Recently, large language models (LLMs) have also been applied to unit test generation, and numerous studies and tools show their strong potential in automatically producing effective test cases.

6.2.1 Pure LLM-Based Approaches. Most studies focus on leveraging LLMs as the primary engine for test generation, optimizing performance through prompt engineering, feedback loops, or fine-tuning. ChatTester [78] leverages structured prompts to guide LLMs in producing unit tests, showing that prompt engineering can significantly improve quality over naive generation. TestPilot [58] targets npm packages in JavaScript/TypeScript, using LLMs along with function signatures, implementations, and documentation examples to generate practical unit tests. ChatUniTest [9] introduces a refinement loop where LLMs iteratively improve generated tests by analyzing coverage feedback. CoverUp [3] combines static analysis with LLM prompting to specifically target uncovered code regions, improving coverage in difficult-to-reach areas. TestART [21] integrates LLM-driven test generation with automated repair techniques, enabling generated tests to adapt when code evolves. HITS [70] employs hierarchical prompting strategies to systematically generate diverse test cases, enhancing coverage and fault detection. SymPrompt [55] augments LLM-based generation with symbolic execution, steering the model toward semantically meaningful test inputs. AUTOE2E [2] leverages LLMs to automatically generate feature-driven end-to-end (E2E) test cases for web applications by inferring potential features and translating them into executable test scenarios. IntUT [45] guides LLM-based test generation using explicit test intentions, including test inputs, mock behaviors, and expected results, to systematically produce unit tests. Fine-tuned LLMs for unit testing [61] apply fine-tuning techniques to LLMs to generate unit tests across different tasks and benchmarks. STRUT [39] guides LLMs to produce structured test cases, which are then transformed into executable unit tests.

6.2.2 Hybrid Approaches Combining LLMs with Traditional Techniques. To overcome the limitations of pure LLM generation, researchers have integrated LLMs with search-based software testing (SBST), fuzzing, retrieval augmentation, and advanced static analysis. CodaMosa [33] combines LLM

guidance with search heuristics to prioritize unexplored code regions and guide test generation. UTGen [12] generates a baseline test suite using EvoSuite and then applies LLM-based transformations to refine it. TestSpark [56] fuses EvoSuite-based search and an IDE-LLM feedback loop to synthesize compile-valid Java unit tests inside IntelliJ IDEA. RefTest [82] introduces reference-based retrieval augmentation for unit test generation by retrieving existing tests of related methods under the Given-When-Then structure, thereby improving the correctness, completeness, and maintainability of generated tests. LLAMAFUZZ [79] adapts large language models as structure-aware mutators for grey-box fuzzing by fine-tuning LLMs on effective fuzzing seeds, enabling the generation of format-valid yet semantically diverse inputs that significantly improve coverage and bug discovery when combined with AFL++. TitanFuzz [13] treats large language models as zero-shot fuzzing engines, using prompt-based program synthesis and multi-site code mutation to automatically generate and evolve deep learning test programs, achieving substantially higher API and code coverage without model fine-tuning. AwTest-LLM [68] leverages static code analysis to enrich LLM prompts with structural and dependency information, enabling the generation of compile-valid C++ unit tests for large-scale autonomous driving systems and improving build success and coverage over naive prompting. Test4Py [40] enhances LLM-based Python test generation with call graph-guided parameter summarization, behavior-guided type inference, type-aware prompting, and adaptive repair to generate semantically valid tests for dynamically typed programs. PALM [10] couples MIR-level CFG path analysis with LLM prompting: it extracts minimal condition chains for each execution path, embeds them with contextual dependencies into per-path prompts, and iteratively repairs compilation errors to yield high-coverage Rust unit tests without fine-tuning. PROBE [34] extends LLM-based test generation to property-based testing by combining contextual grounding, cross-function semantic planning, and adversarial refinement to synthesize stronger properties and expose semantic loopholes in generated tests.

Limitations of Existing Works. While existing LLM-based test generation tools employ various strategies, including prompt-driven generation, baseline suite refinement, or integration with SBST—they face notable limitations. For instance, TestPilot does not adapt prompts based on type information, and UTGen depends on EvoSuite to generate a baseline suite, which cannot handle many complex classes. More generally, most approaches lack tight integration between search-based strategies and LLM guidance, depend heavily on external heuristics, and fail to fully exploit semantic and type information from prior tests. In contrast, LegaTest addresses these shortcomings through a closed-loop framework that tightly couples a genetic algorithm with LLM guidance, formulating test generation as an optimization problem: the genetic algorithm efficiently explores the search space, while the LLM provides semantic reasoning to guide crossover and context-aware mutation, producing structurally valid, semantically meaningful, and diverse test cases.

7 Conclusion

This paper presents LegaTest, a framework that integrates large language models (LLMs) with genetic algorithms through a coordinated Generation–Repair-Optimization process to enhance the quality, semantic soundness, expressiveness of assertions, and maintainability of unit test generation. We introduce a dual-control prompting mechanism that combines structural analysis with semantic context to guide LLMs in producing accurate, logically complete, and semantically meaningful tests. In the optimization phase, LLMs are fused with genetic algorithms in a fitness-driven evolutionary process augmented by runtime feedback. This enables broader exploration, reduces redundancy, enhances structural and semantic coherence through LLM-guided crossover, and supports fine-grained, goal-directed mutation with adaptive control. In the repair phase, a

hybrid mechanism blends rule-based and LLM-driven strategies under hierarchical success criteria to incrementally resolve structural and logical errors.

By coordinating LLMs and genetic algorithms during optimization, LegaTest shifts unit test generation from static prompts to a feedback-driven evolutionary process, improving accuracy, diversity, and adaptability. LLMs provide semantic reasoning, while iterative evolution ensures effective global optimization. Practical design features such as BDD-style naming, AAA structuring, and semantically rich assertions further improve engineering usability. Experiments on five open-source projects demonstrate that LegaTest achieves higher branch coverage than state-of-the-art baselines, while also improving readability, correctness, maintainability, and the semantic validation power of generated assertions.

Acknowledgments

This work was supported by the National Key R&D Program of China under Grant No. 2024YFB4506200.

References

[1] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2024. A3test: Assertion-augmented automated test case generation. *Information and Software Technology* 176 (2024), 107565.

[2] Parsa Alian, Noor Nashid, Mobina Shahbandeh, Taha Shabani, and Ali Mesbah. 2024. Feature-Driven End-To-End Test Generation. *arXiv preprint arXiv:2408.01894* (2024).

[3] Juan Altmayer Pizzorno and Emery D Berger. 2024. CoverUp: Coverage-Guided LLM-Based Test Generation. *arXiv e-prints* (2024), arXiv–2403.

[4] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. 2024. Unit test generation using generative AI: A comparative performance analysis of autogeneration tools. In *Proceedings of the 1st International Workshop on Large Language Models for Code*. 54–61.

[5] Tobias Blickle. 2000. Tournament selection. *Evolutionary computation* 1, 181-186 (2000), 188.

[6] Dimitrios Stamatios Bouras, Sergey Mechtaev, and Justyna Petke. 2025. Llm-Assisted Crossover in Genetic Improvement of Software. In *2025 IEEE/ACM International Workshop on Genetic Improvement (GI)*. IEEE, 19–26.

[7] Amrita Chakraborty and Arpan Kumar Kar. 2017. Swarm intelligence: A review of algorithms. *Nature-inspired computing and optimization* (2017), 475–494.

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[9] 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.

[10] Bei Chu, Yang Feng, Kui Liu, Hange Shi, Zifan Nan, Zhaoqiang Guo, and Baowen Xu. 2025. Boosting Rust Unit Test Coverage through Hybrid Program Analysis and Large Language Models. *arXiv preprint arXiv:2506.09002* (2025).

[11] Henry Coles. 2026. PITest: Real World Mutation Testing. http://pitest.org/. Accessed: 2026-01-12.

[12] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. 2024. Leveraging large language models for enhancing the understandability of generated unit tests. *arXiv preprint arXiv:2408.11710* (2024).

[13] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.

[14] Sumit Dhaiya, Brij Kishore Pandey, Sri Bhargav Krishna Adusumilli, and Rajiv Avacharmal. 2021. Optimizing API Security in FinTech Through Genetic Algorithm based Machine Learning Model. *International Journal of Computer Network and Information Security* 13, 3 (2021), 24.

[15] Eduard P Enoiu, Adnan Čaušević, Thomas J Ostrand, Elaine J Weyuker, Daniel Sundmark, and Paul Pettersson. 2016. Automated test generation using model checking: an industrial evaluation. *International Journal on Software Tools for Technology Transfer* 18, 3 (2016), 335–353.

[16] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary generation of whole test suites. In *2011 11th International Conference on Quality Software*. IEEE, 31–40.

[17] Gordon Fraser and Andrea Arcuri. 2011. 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*. 416–419.

[18] Gordon Fraser and Andreas Zeller. 2010. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th international symposium on Software testing and analysis*. 147–158.

[19] Boni Garcia. 2017. *Mastering Software Testing with JUnit 5: Comprehensive guide to develop high quality Java applications*. Packt Publishing Ltd.

[20] Angelo Gargantini and Constance Heitmeyer. 1999. Using model checking to generate tests from requirements specifications. *ACM SIGSOFT Software Engineering Notes* 24, 6 (1999), 146–162.

[21] Siqi Gu, Chunrong Fang, Quanjun Zhang, Fangyuan Tian, and Zhenyu Chen. 2024. Testart: Improving llm-based unit test via co-evolution of automated generation and repair iteration. *arXiv e-prints* (2024), arXiv–2408.

[22] Shekhar Gulati and Rahul Sharma. 2017. JUnit 5 Extension Model. In *Java Unit Testing with JUnit 5: Test Driven Development with JUnit 5*. Springer, 121–137.

[23] Hirohide Haga and Akihisa Suehiro. 2012. Automatic test case generation based on genetic algorithm and mutation analysis. In *2012 IEEE International Conference on Control System, Computing and Engineering*. IEEE, 119–123.

[24] Yong He, Kayvon Fatahalian, and Theresa Foley. 2018. Slang: language mechanisms for extensible real-time shading systems. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–13.

[25] John H Holland. 1992. Genetic algorithms. *Scientific american* 267, 1 (1992), 66–73.

[26] Haoya Hu. 2024. System Parameter Optimization based on Genetic Algorithm. *International Journal of Mechanical and Electrical Engineering* 2, 3 (2024), 11–16.

[27] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems* 43, 2 (2025), 1–55.

[28] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[29] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. 2021. A review on genetic algorithm: past, present, and future. *Multimedia tools and applications* 80, 5 (2021), 8091–8126.

[30] Shaker Mahmud Khandaker, Fitsum Kifetew, Davide Prandi, and Angelo Susi. 2025. AugmenTest: Enhancing Tests with LLM-Driven Oracles. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 279–289.

[31] Sven Kosub. 2019. A note on the triangle inequality for the Jaccard distance. *Pattern Recognition Letters* 120 (2019), 36–38.

[32] Annu Lambora, Kunal Gupta, and Kriti Chopra. 2019. Genetic algorithm-A literature review. In *2019 international conference on machine learning, big data, cloud and parallel computing (COMITCon)*. IEEE, 380–384.

[33] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. 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, 919–931.

[34] Xiao Li, Runlin Liu, Zhe Zhang, Xiang Gao, and Hailong Sun. 2026. Beyond Superficial Tests: Adversarial Refinement for Reliable Property-Based Testing. (2026).

[35] Jiang Lin and Weiyu Dong. 2024. SPAFuzz: Web Security Fuzz Testing Tool Based on Fuzz Testing and Genetic Algorithm. In *2024 4th International Conference on Consumer Electronics and Computer Engineering (ICCECE)*. IEEE, 190–195.

[36] Adam Lipowski and Dorota Lipowska. 2012. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications* 391, 6 (2012), 2193–2196.

[37] Jingsen Liu, Yiwen Fu, Yu Li, Lin Sun, and Huan Zhou. 2024. An effective theoretical and experimental analysis method for the improved slime mould algorithm. *Expert Systems with Applications* 247 (2024), 123299.

[38] Jingsen Liu, Yiwen Fu, Yu Li, and Huan Zhou. 2023. A novel improved slime mould algorithm for engineering design. *Soft Computing* 27, 17 (2023), 12181–12210.

[39] Jinwei Liu, Chao Li, Rui Chen, Shaofeng Li, Bin Gu, and Mengfei Yang. 2025. STRUT: Structured Seed Case Guided Unit Test Generation for C Programs using LLMs. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 2113–2135.

[40] Runlin Liu, Zhe Zhang, Yunge Hu, Yuhang Lin, Xiang Gao, and Hailong Sun. 2025. Type-aware LLM-based Regression Test Generation for Python Programs. *arXiv preprint arXiv:2503.14000* (2025).

[41] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 168–172.

[42] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An Empirical Study of Automated Unit Test Generation for Python. CoRR abs/2111.05003 (2021). *arXiv preprint arXiv:2111.05003* (2021).

[43] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th international symposium on software testing and analysis*. 94–105.

[44] Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika* 12, 2 (1947), 153–157.
[45] Zifan Nan, Zhaoqiang Guo, Kui Liu, and Xin Xia. 2025. Test intention guided llm-based unit test generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 779–779.
[46] Authors of LegaTest. 2025. Fusing LLMs and Genetic Algorithm for High-Quality Unit Test Generation. https://github.com/wenwen-666/LegaTest.
[47] Michael Olan. 2003. Unit testing: test early, test often. *Journal of Computing Sciences in Colleges* 19, 2 (2003), 319–328.
[48] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.
[49] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 75–84.
[50] G Pavai and TV Geetha. 2016. A survey on crossover operators. *ACM Computing Surveys (CSUR)* 49, 4 (2016), 1–43.
[51] Corina S Păsăreanu, Peter C Mehlitz, David H Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proceedings of the 2008 international symposium on Software testing and analysis*. 15–26.
[52] Iman Rahimi, Amir H Gandomi, Fang Chen, and Efren Mezura-Montes. 2023. A review on constraint handling techniques for population-based algorithms: from single-objective to multi-objective optimization. *Archives of Computational Methods in Engineering* 30, 3 (2023), 2181–2209.
[53] Thorsten Rangnau, Remco v Buijtenen, Frank Fransen, and Fatih Turkmen. 2020. Continuous security testing: A case study on integrating dynamic security testing tools in ci/cd pipelines. In *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, 145–154.
[54] Devdeep Ray and Srinivasan Seshan. 2022. CC-fuzz: genetic algorithm-based fuzzing for stress testing congestion control algorithms. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. 31–37.
[55] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 951–971.
[56] Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. 2024. Testspark: Intellij idea's ultimate test generation companion. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 30–34.
[57] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527* (2023).
[58] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* 50, 1 (2023), 85–105.
[59] Nicol N Schraudolph and Richard K Belew. 1992. Dynamic parameter encoding for genetic algorithms. *Machine learning* 9, 1 (1992), 9–21.
[60] Xavier Sécheresse, Antoine Villedieu de Torcy, et al. 2025. GAAPO: Genetic Algorithmic Applied to Prompt Optimization. *arXiv preprint arXiv:2504.07157* (2025).
[61] Ye Shang, Quanjun Zhang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. 2025. A large-scale empirical study on fine-tuning large language models for unit testing. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1678–1700.
[62] Elvys Soares, Márcio Ribeiro, Rohit Gheyi, Guilherme Amaral, and André Santos. 2022. Refactoring test smells with junit 5: Why should developers keep up-to-date? *IEEE Transactions on Software Engineering* 49, 3 (2022), 1152–1170.
[63] Peter Stone. 2009. The logic of random selection. *Political theory* 37, 3 (2009), 375–397.
[64] Student. 1908. The probable error of a mean. *Biometrika* (1908), 1–25.
[65] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 245–256.
[66] Jun Tang, Gang Liu, and Qingtao Pan. 2021. A review on representative swarm intelligence algorithms for solving optimization problems: Applications and trends. *IEEE/CAA Journal of Automatica Sinica* 8, 10 (2021), 1627–1643.
[67] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).
[68] Wenhan Wang, Xuan Xie, Yuheng Huang, Renzhi Wang, An Ran Chen, and Lei Ma. 2025. Fine-grained Testing for Autonomous Driving Software: a Study on Autoware with LLM-driven Unit Testing. *arXiv preprint arXiv:2501.09866* (2025).
[69] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[70] 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.
[71] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1398–1409.
[72] Feng Yang, Pengxiang Wang, Yizhai Zhang, Litao Zheng, and Jianchun Lu. 2017. Survey of swarm intelligence optimization algorithms. In *2017 IEEE International Conference on Unmanned Systems (ICUS)*. IEEE, 544–549.
[73] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Hu. 2024. Harnessing the power of llms in practice: A survey on chatgpt and beyond. *ACM Transactions on Knowledge Discovery from Data* 18, 6 (2024), 1–32.
[74] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, et al. 2024. On the evaluation of large language models in unit test generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1607–1619.
[75] Qian Yang, J Jenny Li, and David Weiss. 2006. A survey of coverage based testing tools. In *Proceedings of the 2006 international workshop on Automation of software test*. 99–103.
[76] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability* 22, 2 (2012), 67–120.
[77] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. 2021. Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In *Proceedings of the Web Conference 2021*. 3087–3098.
[78] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207* (2023).
[79] Hongxiang Zhang, Yuyang Rong, Yifeng He, and Hao Chen. 2024. Llamafuzz: Large language model enhanced greybox fuzzing. *arXiv preprint arXiv:2406.07714* (2024).
[80] Quanjun Zhang, Weifeng Sun, Chunrong Fang, Bowen Yu, Hongyan Li, Meng Yan, Jianyi Zhou, and Zhenyu Chen. 2025. Exploring automated assertion generation via large language models. *ACM Transactions on Software Engineering and Methodology* 34, 3 (2025), 1–25.
[81] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.
[82] Zhe Zhang, Xingyu Liu, Yuanzhang Lin, Xiang Gao, Hailong Sun, and Yuan Yuan. 2025. Reference-Based Retrieval-Augmented Unit Test Generation. *ACM Transactions on Software Engineering and Methodology* (2025).
[83] Yong Zheng and David Xuejun Wang. 2022. A survey of recommender systems with multi-objective optimization. *Neurocomputing* 474 (2022), 141–153.

A Raw Zero-Shot GPT-5-mini Baseline on the Selected Subset

To complement the evaluation on the selected subset under the GPT-based setting, we additionally report a raw zero-shot GPT-5-mini baseline. We report line coverage, branch coverage, method coverage, and the average number of test cases.

Table 7 presents the results of the raw zero-shot GPT-5-mini baseline, together with the GPT-based results of LegaTest (Initial), the full version of LegaTest, ChatUniTest, UTGen, and EvoSuite on the same selected subset.

Table 7 reports the additional raw zero-shot GPT-5-mini baseline on the selected subset under the GPT-based setting. The results show that raw zero-shot GPT-5-mini is a meaningful baseline: it outperforms some existing baselines on several projects, indicating that a strong general-purpose LLM can already provide non-trivial test generation capability on this subset. However, it still remains clearly below both LegaTest (Initial) and the full version of LegaTest. In the "Total" row, raw zero-shot GPT-5-mini achieves 49.71% line coverage, 43.15% branch coverage, and 55.11% method coverage, compared with 73.21%, 62.74%, and 79.20% for LegaTest (Initial), and 84.05%, 72.90%, and 89.82% for the full version of LegaTest. These results suggest that the improvements of LegaTest cannot be explained solely by using a strong GPT backend in a raw zero-shot manner, but instead come from the proposed structured generation and iterative optimization design.

Table 7. Coverage Results on the Selected Subset under the GPT-based Setting, with an Additional Raw Zero-Shot GPT-5-mini Baseline. EvoSuite is LLM-agnostic; its results are therefore identical to those reported in the main text.

Project Name	Method	Line Coverage (%)	Branch Coverage (%)	Method Coverage (%)	Avg. # Test Cases
Math_5f	Raw Zero-Shot GPT-5-mini	44.00	35.29	49.73	**8.18**
	LegaTest (Initial)	59.53	46.48	69.16	20.54
	LegaTest	**75.02**	**59.80**	**81.30**	21.63
	ChatUniTest	11.38	8.76	10.76	19.28
	EvoSuite	–	–	–	–
	UTGen	–	–	–	–
Collections_25f	Raw Zero-Shot GPT-5-mini	71.77	69.64	72.94	**6.89**
	LegaTest (Initial)	78.31	74.59	80.60	16.67
	LegaTest	**87.91**	**83.53**	**89.33**	19.96
	ChatUniTest	34.94	32.75	35.89	32.08
	EvoSuite	65.56	60.59	69.23	18.30
	UTGen	30.67	26.31	34.95	9.87
Lang_6f	Raw Zero-Shot GPT-5-mini	47.73	41.76	54.64	9.59
	LegaTest (Initial)	78.66	68.83	83.69	17.92
	LegaTest	**89.86**	**78.94**	**96.06**	24.07
	ChatUniTest	28.31	28.03	27.23	33.12
	EvoSuite	80.81	74.71	83.39	30.39
	UTGen	48.62	42.13	51.89	**7.91**
Compress_6f	Raw Zero-Shot GPT-5-mini	32.92	26.87	43.11	**5.76**
	LegaTest (Initial)	60.12	50.45	71.04	10.86
	LegaTest	**67.61**	**58.23**	**78.14**	15.69
	ChatUniTest	46.87	43.06	50.78	19.25
	EvoSuite	66.65	65.04	76.36	17.38
	UTGen	40.47	37.54	57.60	9.83
Csv_16f	Raw Zero-Shot GPT-5-mini	3.85	0	3.85	**3**
	LegaTest (Initial)	83.38	62.01	92.93	13.35
	LegaTest	93.16	74.14	99.58	21.30
	ChatUniTest	62.45	46.03	64.21	15.65
	EvoSuite	**93.54**	**87.43**	**100.00**	25.30
	UTGen	48.08	43.13	50.00	19.00
Commons_Cli	Raw Zero-Shot GPT-5-mini	–	–	–	–
	LegaTest (Initial)	68.71	47.93	78.69	**14.64**
	LegaTest	**84.29**	70.13	**92.03**	22.50
	ChatUniTest	55.52	42.39	52.56	21.39
	EvoSuite	81.65	**71.53**	87.70	41.77
	UTGen	–	–	–	–
Commons_Csv	Raw Zero-Shot GPT-5-mini	29.13	16.32	34.09	**4**
	LegaTest (Initial)	64.23	36.53	71.63	14.28
	LegaTest	**82.55**	**53.57**	**88.47**	23.01
	ChatUniTest	34.13	25.00	39.20	26.25
	EvoSuite	–	–	–	–
	UTGen	–	–	–	–
Ecommerce	Raw Zero-Shot GPT-5-mini	86.21	–	86.67	**5**
	LegaTest (Initial)	83.60	–	84.18	8.35
	LegaTest	**100.00**	–	**100.00**	13.91
	ChatUniTest	7.69	–	8.92	11.70
	EvoSuite	–	–	–	–
	UTGen	–	–	–	–
Binance	Raw Zero-Shot GPT-5-mini	10.38	5.27	14.60	**3.9**
	LegaTest (Initial)	73.96	46.63	81.59	12.97
	LegaTest	**75.72**	**49.60**	**88.47**	19.23
	ChatUniTest	28.15	0.00	27.11	24.70
	EvoSuite	–	–	–	–
	UTGen	–	–	–	–
Total	Raw Zero-Shot GPT-5-mini	49.71	43.15	55.11	**7.60**
	LegaTest (Initial)	73.21	62.74	79.20	16.15
	LegaTest	**84.05**	**72.90**	**89.82**	20.98
	ChatUniTest	31.82	29.22	32.22	27.53
	EvoSuite	74.01	68.93	78.46	24.69
	UTGen	41.51	36.41	47.73	9.17