2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)

FAILMAPPER: Automated Generation of Unit Tests Guided by Failure Scenarios

Ruiqi Dong
Swinburne University of Technology
Melbourne, Australia
rdong@swin.edu.au

Zehang Deng
Swinburne University of Technology
Melbourne, Australia
zehangdeng@swin.edu.au

Xiaogang Zhu
Adelaide University
Adelaide, Australia
xiaogang.zhu@adelaide.edu.au

Xiaoning Du
Monash University
Melbourne, Australia
Xiaoning.Du@monash.edu

Huai Liu
Swinburne University of Technology
Melbourne, Australia
hliu@swin.edu.au

Shaohua Wang*
Central University of Finance and Economics
Beijing, China
davidshwang@ieee.org

Sheng Wen*
Swinburne University of Technology
Melbourne, Australia
swen@swin.edu.au

Yang Xiang
Swinburne University of Technology
Melbourne, Australia
yxiang@swin.edu.au

***Abstract*—The automation of unit test generation has become a critical task for improving the overall efficiency of software development and testing. Many existing techniques attempt to generate a sufficient number of test cases to achieve high code coverage. However, it has been shown that a high coverage does not necessarily guarantee effective bug discovery. A potential enhancement is to guide the unit test generation based on bug properties. However, this solution is challenged by the large number and diversity of bug types, making it difficult to comprehensively summarize bug properties.**

We observe that failures, presented as the results of bugs, manifest in a limited number of scenarios. Therefore, instead of bug properties, in this paper, we propose an innovative framework, named FAILMAPPER, which uses failure scenarios to guide the generation of unit tests. We summarize nine failure scenarios and design the corresponding failure-triggering test strategies. This significantly improves the efficacy of generating test cases towards triggering bugs. To systematically explore possible failure scenarios, FAILMAPPER employs the Monte Carlo Tree Search algorithm to search for the faults that may lead to a failure. Experiments demonstrate that, on 50 known bugs in the Defects4J benchmark, FAILMAPPER can detect many more bugs than five typical unit testing approaches, including EvoSuite, Randoop, CoverUp, HITS, and SymPrompt (40 versus at most 12, out of all 50 bugs). Meanwhile, FAILMAPPER detects 12 out of 20 bugs in the GitBug-Java and Bears-benchmark datasets. We reveal 36 potential issues from 2 Apache projects, and 14 of them have been confirmed as bugs, further demonstrating FAILMAPPER's effectiveness. The experimental results show that our new framework can significantly enhance the overall efficacy of unit testing.

***Index Terms*—Automated Unit Test, Failure Scenario Guided, Software Bug, LLM.**

I. INTRODUCTION

Unit testing is a long-standing and widely used method in software development, where developers write test cases to automatically verify the correctness of individual components [1]. However, most software companies still depend on manually generated unit tests, suffering from the limited capability of thorough testing [2]. This undermines software quality, leading to costly or even incalculable consequences when serious issues occur. As a result, there is increasing interest in automated unit test generation as a means to improve testing efficiency and ensure software reliability by producing effective and focused unit tests.

The fundamental task of automated unit test generation is to produce a pair of input and its expected output, which aims to verify the behavior of units. Existing techniques, including Search-Based Software Testing (SBST) approaches [3]–[5] and recent LLM-driven methods [6]–[8], search the input-output pairs with the focus on maximizing code coverage. However, high code coverage does not necessarily guarantee bug detection [9]. Even with code coverage regularly exceeding 80–90%, these tools frequently fail to detect critical bugs [10]. A potential enhancement on bug discovery is to steer test generation by incorporating bug characteristics, as adopted in other software testing methods [11]. However, unit tests naturally intend to detect various types of bugs, especially logic bugs, which arise because code does not meet business requirements [12]. This diversity challenges the summarization of bug characteristics because each bug type may display its own distinctive root cause [13].

Although bugs can be introduced through a wide variety of ways [14], we observe that code failures tend to manifest in a limited set of recurring scenarios. A failure is a departure of the program or program component behavior from its required behavior [15]. Therefore, our key idea is to *guide the generation of unit tests based on failure scenarios, which significantly reduces the search space of input-output pairs*. To achieve this, we have to overcome the challenge of systematically identifying and triggering potential failures in code without explicit specifications, while minimizing false positives in the generated tests. First, we need to identify potential failure points in the source code in an unbounded specification space. Second, once potential failure points are identified, we need to generate test cases that can effectively trigger these failure

*Corresponding author.

scenarios without clear failure specifications. Third, automated test generation, particularly when using LLMs, suffers from high false-positive rates mainly due to the inaccuracy of assertions used for verifying the correctness of test output.

To address the challenges, we propose FAILMAPPER, the first framework that uses failure scenarios to guide the generation of unit tests. To solve the first challenge, we conducted an empirical study of 854 real-world bugs to identify common failure scenarios, then leveraged static analysis and LLM's semantic understanding to recognize potential instances of these scenarios in source code. For the second challenge, we designed specialized test generation strategies for failure scenarios and employed a *Failure-Aware Monte Carlo Tree Search (MCTS)* algorithm to efficiently explore the space of possible test inputs and assertions . To address the third challenge, we developed a multi-level verification framework that combines pattern-based filtering, static analysis, and LLM-enhanced semantic verification to validate bug candidates.

Our experimental evaluation on Defects4J [16] demonstrates that this failure-guided testing approach significantly outperforms existing methods in detecting real-world bugs. FAILMAPPER is compared with two SBST tools, including Evosuite [3] and Randoop [4], and state-of-the-art LLM-based methods, including Coverup [6], Hits [7], and SymPrompt [8]. We selected the 10 most recent active faults from Cli [17], Codec [18], Csv [19], Math [20], and Compress [21], respectively. Compared with the best performance baseline, FAILMAPPER achieves 233% higher bug detection rate without sacrificing code coverage. Meanwhile, FAILMAPPER reduces the false positive rate by 64.7% compared to the best LLM-based baseline, making it more efficient for practical testing scenarios. The main contributions of this paper are:

1) We summarize nine failure scenarios by conducting a large empirical study on 854 real-world failures to quantify the distribution and characteristics of failures.
2) We propose a failure-guided framework, FAILMAPPER, that effectively triggers bugs via unit test generation. FAILMAPPER reframes unit test generation from maximizing code coverage to targeting failure scenarios.
3) We comprehensively evaluate FAILMAPPER on real-world bugs, providing an extremely higher bug detection rate and lower false positive rates.

II. MOTIVATION

A. The Gap Between Coverage and Bug Detection

Consider the bug in Listing 1, which shows a null pointer issue in the Commons CSV library [22]. This bug manifests when processing header records, where the code calls `trim()` on a potentially null header value (in line 5), resulting in a NullPointerException. When we analyzed the tests generated by state-of-the-art tools for this method, we found a revealing pattern that all generated tests achieve 100% code coverage by passing valid string values and verifying successful assignments. However, none attempt to trigger the failure scenario by passing null values or including null-related assertions to validate the exception handling behavior.

Listing 1 CSV-122: Bug for Null Reference

```
1 for (int i = 0; i < headerRecord.length; i++) {
2     final String header = headerRecord[i];
3     // BUG: Null pointer issue for emptyHeader
4     // trim() cannot deal with a null value
5     final boolean emptyHeader =
      ↪  header.trim().isEmpty();
6     ...
7 }
```

This observation reveals a fundamental limitation that existing test generation approaches optimize for executing code paths successfully, systematically neglecting input and assertions that reveal failure scenarios.

B. Misaligned Search Objectives

The failure neglect is a common disadvantage of existing tools, because they are optimized for what they can measure (successfully examined code regions) rather than what matters for bug detection (failure scenarios) [23], [24]. For example, in Listing 1, it stems from a fundamental misalignment in how current approaches conceptualize test generation. Both SBST and LLM-based methods maximize successful code execution, treating exceptions and failures as undesirable outcomes to be neglected rather than valuable signals to be explored. EvoSuite, an SBST tool, supports multiple coverage criteria in unit test generation [25]. While prior work has explored fault-driven approaches such as exception coverage [26] and mutation-driven generation [27], these techniques still operate within the coverage maximization paradigm. Similarly, LLM-based approaches like CoverUp and HITS focus on covering unexplored code regions with valid inputs, lacking explicit mechanisms to explore failure-inducing scenarios [28].

C. The Key Insight: Failures Follow Systematic Scenarios

A bug is a defect in the source code that represents an incorrect implementation violating the intended behavior, and a failure is the observable manifestation of a bug during execution [15]. While a single bug can manifest through multiple failure scenarios [29], [30], such as a missing null check causing NullPointerException in different contexts, and different bugs can produce similar failures, like various logic errors leading to incorrect outputs. The realization led us to a critical question: instead of trying to understand what code should do to get the expected output and avoid bugs (which requires unbounded specification comprehension), can we directly target how code might fail (which follows finite observations)? To answer this, we conducted an empirical study of 854 real-world bugs from Defects4J. Our analysis revealed a striking finding that *despite software failures being caused by various reasons, they manifest in remarkably consistent scenarios*. As shown in Table I, we identified 12 failure scenarios, and FAILMAPPER covers 9 of them to cover 90.5% of the bugs. This systematic categorization enables us to transform the unit test generation problem from an unbounded specification understanding task to a bounded

TABLE I: Failure Classification

Category	Failure (Subcategories)	Count	Total %
Runtime Exception Failure (REF)	**1.1 Null Reference**	40	4.7
	- Null Reference	40	4.7
	1.2 Index Boundary	36	4.2
	- Index Violations	36	4.2
	1.3 Resource Management	34	3.9
	- Uncaught Exception Propagation	12	1.4
	- Resource Leak Failures	8	0.9
	- Abnormal Termination	14	1.6
	1.4 Concurrent Modification	26	3.1
	- Collection Modification Failures	14	1.6
	- Infinite Loop Failures	9	1.1
	- Control Flow Interruption	3	0.4
	1.5 Race Condition (Uncovered)	6	0.7
	- Race Condition Failures	6	0.7
Logic and Semantic Failures (LSF)	**2.1 Incorrect Behavior**	268	31.4
	- API Contract Violations	261	30.6
	- Parameter Mismatch Failures	7	0.8
	2.2 Logic Assertion	136	15.9
	- Boolean Assertion Failures	72	8.4
	- Boundary Assertion Violations	35	4.1
	- Incomplete Logic Failures	25	2.9
	- Operator Precedence Failures	4	0.5
	2.3 Data Integrity	126	14.8
	- Special Value Mishandling	80	9.4
	- Validation Failures	46	5.4
	2.4 String Processing	69	8.1
	- Encoding Mismatch Failures	41	4.8
	- String Comparison Failures	28	3.3
	2.5 Numeric Computation	35	5.2
	- Type Conversion Failures	35	4.1
	- Arithmetic Failures	9	1.1
	2.6 Configuration Dependent (Uncovered)	14	1.6
	- Argument Order Failures	7	0.8
	- Configuration Load Failures	7	0.8
Other Failures	**3.1 Others (Uncovered)**	64	7.5
	- Regression Test Failures	49	5.7
	- Platform-Specific Failures	15	1.8

matching problem. Therefore, we can directly target these failure scenarios without complex program analysis on specific fault patterns [31].

D. MCTS for Searching

The challenge of systematically exploring possible failure scenarios presents a search problem. We need to (1) efficiently explore a space of potential failure-inducing inputs and assertions, (2) learn which inputs and assertions successfully trigger failures versus those that execute without failures, and (3) balance exploring new failure scenarios with exploiting proven failure results. However, traditional random testing approaches lack the learning capability to focus on promising failure scenarios. Coverage-guided approaches explicitly avoid the failure space that we want to explore. This is where MCTS becomes the natural choice [32]. MCTS can learn from each test execution to distinguish between inputs and assertions that successfully trigger failures and those that do not [33]. Its Upper Confidence Bounds for Tree(UCT) formula naturally balances exploring new failure scenarios with exploiting results that have proven effective. By combining systematic failure scenarios with MCTS's adaptive search capabilities, we can transform test generation from a coverage optimization problem into a failure discovery problem.

III. Methodology

FailMapper introduces a failure-driven test generation framework that shifts the focus from maximizing code coverage to triggering software failures. The core insight driving our approach is that while software bugs vary infinitely, we can generate test cases that expose bugs effectively with the guidance of failures. Our Framework consists of four key components: Failure Scenario Analysis, Failure-Triggering strategies, Failure-Aware MCTS, and Multi-Level Verification. Figure 1 illustrates how these components work together. Starting from the source code, FailMapper analyzes potential failure points, applies targeted strategies through MCTS-guided test generation, and verifies that triggered failures represent actual bugs rather than wrong code behaviors.

FailMapper employs a pipeline that systematically transforms Java source code into failure-triggering unit tests, as shown in Figure 2. The static analysis stage employs a multi-layered parsing strategy to handle diverse Java syntax. For context information, FailMapper operates primarily on code-level artifacts without requiring external documentation. The system extracts method signatures, control flow patterns, data dependencies, and Javadoc comments to understand code semantics. And then, FailMapper collects the FA-MCTS engine's feedback for the current testing iteration and generates a new prompt for the next iteration.

A. Failure Category Analysis

Our failure category analysis begins with an observation that software bugs manifest through a limited set of observable runtime behaviors. Rather than attempting to identify buggy code patterns statically, we focus on the failure scenarios that indicate the presence of bugs. This approach enables a more targeted and efficient bug-triggering process by directly addressing the observable manifestations of software defects.

The development of our failure taxonomy followed a methodology combining theoretical foundations with empirical validation. We first surveyed established bug taxonomies from previous research, including studies on concurrency bug characteristics [34], the analysis of bug patterns in open-source software [35], and the investigation of deep learning bug characteristics [36]. Other studies [37]–[39] also identified failure scenarios such as null pointer dereferences, boundary violations, resource leaks, and type mismatches that have been subjects of software engineering research for decades.

We then examined the prevalence of these failure scenarios in 854 real-world bugs from the Defects4J benchmark [16] for contemporary Java software. For each bug, we analyzed observable failure behavior, the conditions that triggered the failure, and semantic contexts. This analysis confirmed that 90.5% of bugs manifest through nine distinct failure scenarios, which we organized into two main categories.

1) Covered Runtime Exception Failures (REF) - 15.9%: These manifest as uncaught exceptions during execution.

1.1) *Null Reference Failures (4.7%).* This failure scenario occurs when operations are performed on null objects,

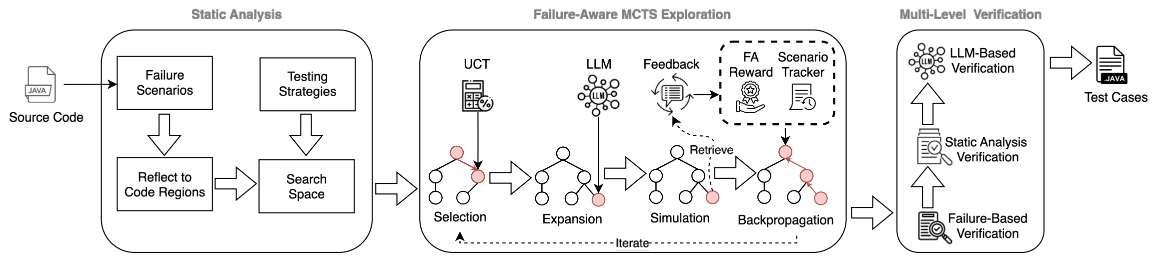


Fig. 1: Overview of the FAILMAPPER framework. The workflow starts with source code analysis to identify potential failure scenarios, then applies failure-triggering test strategies through Failure-Aware MCTS exploration. The generated test cases undergo multi-level verification (scenario-based, static analysis, and LLM-enhanced semantic verification) to distinguish actual bugs from false positives, ultimately producing validated bug-triggering unit tests.

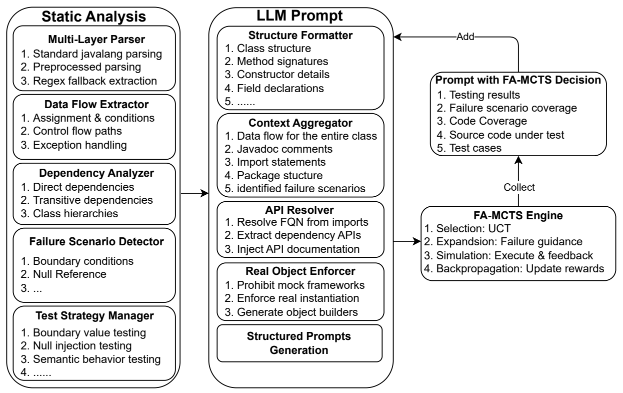

Static Analysis
Multi-Layer Parser
1. Standard javalang parsing
2. Preprocessed parsing
3. Regex fallback extraction
Data Flow Extractor
1. Assignment & conditions
2. Control flow paths
3. Exception handling
Dependency Analyzer
1. Direct dependencies
2. Transitive dependencies
3. Class hierarchies
Failure Scenario Detector
1. Boundary conditions
2. Null Reference
3. ...
Test Strategy Manager
1. Boundary value testing
2. Null injection testing
3. Semantic behavior testing
4. ......
LLM Prompt
Structure Formatter
1. Class structure
2. Method signatures
3. Constructor details
4. Field declarations
5. ......
Context Aggregator
1. Data flow for the entire class
2. Javadoc comments
3. Import statements
4. Package stucture
5. identified failure scenarios
API Resolver
1. Resolve FQN from imports
2. Extract dependency APIs
3. Inject API documentation
Real Object Enforcer
1. Prohibit mock frameworks
2. Enforce real instantiation
3. Generate object builders
Structured Prompts Generation
Add
Prompt with FA-MCTS Decision
1. Testing results
2. Failure scenario coverage
3. Code Coverage
4. Source code under test
5. Test cases
Collect
FA-MCTS Engine
1. Selection: UCT
2. Expandsion: Failure guidance
3. Simulation: Execute & feedback
4. Backpropagation: Update rewards

Fig. 2: The detailed pipeline of FAILMAPPER to construct context.

resulting in NullPointerException. FAILMAPPER identifies potential null reference vulnerabilities by analyzing method calls on variables that might be null, with special attention to parameters and nested property access chains (See Example Listing 1).

1.2) *Index Boundary Failures (4.2%).* They manifest as `ArrayIndexOutOfBoundsException` or similar errors when accessing collections with invalid indices. FAILMAPPER reflects this failure scenario to code regions with operations on array index and string index.

1.3) *Resource Management Failures (3.9%).* They occur when resources are not properly handled, especially under exceptional conditions like swallowed exceptions. These involve improper exception propagation, inadequate cleanup during exceptions, and incorrect return values. Our detection approach focuses on empty catch blocks, overly generic exception handling, and improper resource cleanup.

1.4) *Concurrent Modification Failures (3.1%).* They arise from improper collection modifications during iteration. This failure scenario involves missing breaks, collection modifications during iteration, and improper loop termination conditions. These errors occur when manipulating collections or using iterators incorrectly, often leading to ConcurrentModificationExceptions or infinite loops. FAILMAPPER reflects this scenario to code regions by detecting modified collections during iterations.

2) Covered Logic and Semantic Failures (LSF) - 75.4%: These failures represent violations of business logic or semantic constraints without necessarily throwing exceptions.

2.1) *Incorrect Behavior Failures (31.4%).* The program produces wrong results while executing successfully. These require understanding expected behavior and generating inputs that expose discrepancies (e.g., FIFO vs. LIFO). For this scenario, we leverage LLMs to analyze the unit's logic by providing related contexts, such as Javadoc and method signatures. If LLMs detect incorrect behavior, we ask LLMs to respond to the issue description and corresponding test strategies in a template.

2.2) *Logic Assertion Failures (15.9%).* They include assertion violations and boolean logic errors that cause unexpected program states by misapplying DeMorgan's laws (Negation of a conjunction and disjunction) [40]. This scenario is caused by conditional statements and Boolean expressions.

2.3) *Data Integrity Failures (14.8%).* They involve mishandling of special values (null, empty, negative) or validation logic. To reflect this scenario in code regions, we focus on validation methods and data transformation operations.

2.4) *String Processing Failures (8.1%).* They include encoding issues, incorrect comparisons, and parsing errors. Their systematic nature is evident in recurring code regions, particularly in operations such as parsing, tokenization, and string matching.

2.5) *Numeric Computation Failures (4.1%).* They manifest as overflow, underflow, or precision errors. They exhibit consistent code regions, often arising when mathematical invariants are violated or when numeric values exceed the representational bounds of their data types.

3) Limitations and Excluded Scenarios - 7.5%: While our approach covers 90.5% of observed bugs, certain failure categories are inherently unsuitable for systematic failure-driven testing, shown in Table I with "Uncovered".

3.1) *Race Condition Failures (0.7%).* These require precise thread interleavings and timing control that cannot be reliably reproduced through input generation alone. They need specialized concurrency testing tools [41].

3.2) *Configuration-Dependent Failures (1.8%).* They depend on external configuration files, environment variables, or deployment settings that exist outside the unit testing scope. They require integration or system-level testing.

3.3) *Regression-Specific Failures (5.7%).* They only manifest when comparing behavior across different versions of the software. They require historical context and version-aware testing that goes beyond single-version unit testing.

3.4) *Platform-Specific Failures (1.8%).* These manifest only on specific operating systems, architectures, or runtime environments. Systematic testing would require maintaining multiple execution environments.

The key distinction of our approach is that we do not attempt to statically identify where these failures might occur in the code. Instead, FAILMAPPER generates test cases designed to trigger each failure scenario and observe whether the failure actually manifests by applying corresponding testing strategies. This dynamic approach allows us to find bugs without complex program analysis or specification inference, while acknowledging that certain bug categories require specialized testing approaches beyond our framework's scope.

Note that our categorization, in Table I, is hierarchical with Level 1 representing high-level failure patterns and Level 2 providing implementation-specific variations. We chose to target Level 1 scenarios in FAILMAPPER as they provide the optimal balance between coverage and tractability. While finer-grained categorizations are possible, our experiments, in Table IV, show that Level 1 scenarios are sufficient to achieve significant bug detection improvements.

B. Failure-Triggering Test Strategies

While failure scenarios identify what failures to target, we need systematic strategies for how to trigger these failures. The key insight is that many failure scenarios share common triggering mechanisms. For instance, both null reference failures and data integrity failures often involve edge case testing that can be triggered through similar strategies. By identifying these commonalities, as shown in Table II, we create reusable strategies that work across scenarios. Importantly, our strategies encompass not only input generation but also assertion synthesis to effectively detect both exceptional and semantic failures. Our framework implements nine core testing strategies, each designed to systematically explore different dimensions of potential failures.

TABLE II: Mapping Failure Scenario to Testing Strategies

Failure Scenario	Primary Strategy	Secondary Strategy
Null Reference	Null Injection	Edge Case
Index Boundary	Boundary Value	Extreme Value
Resource Management	Exception Injection	State Mutation
Concurrent Modification	State Mutation	–
Incorrect Behavior	SB Testing	Combination
Logic Assertion	Combination	State Mutation
Data Integrity	Edge Case	Boundary Value
String Processing	Format Variation	Edge Case
Numeric Computation	Extreme Value	Boudnary Value

Note: SB Testing is for Semantic Behavior Testing.

- *Null Injection Testing* serves as our primary strategy for exposing failures related to missing null checks. This strategy asks LLMs to systematically replace object parameters with null values and observes whether the code properly handles these cases. Beyond simple null replacement, the strategy also tests partially null states, such as objects with null fields or collections containing null elements. For assertion generation, this strategy limits LLMs to using null-related and exception-based assertions, expecting either successful null handling or specific exceptions like NullPointerException.
- *Boundary Value Testing* targets the critical transition points in program logic where off-by-one errors and range violations commonly occur. The strategy asks LLMs to generate test inputs at and around natural boundaries such as zero, array boundaries, and collection sizes. For array operations, it systematically tests indices at -1, 0, `length-1`, `length`, and `length+1` positions. The strategy generates both exception assertions for out-of-bounds access and value assertions to verify correct behavior at valid boundaries.
- *Extreme Value Testing* extends boundary testing to the limits of data types and numeric representations. This strategy specifically tests with `Integer.MAX_VALUE`, `Integer.MIN_VALUE`, `Double.POSITIVE_INFINITY`, `Double.NaN`, and other extreme values. Beyond triggering overflow exceptions, this strategy also asks LLMs to generate assertions that verify numeric operations handle extreme values correctly, checking for infinity propagation, `NaN` handling, and overflow behavior in arithmetic operations.
- *Combinatorial Testing* addresses the complexity of boolean logic and control flow decisions. Rather than randomly generating boolean inputs, this strategy asks LLMs to explore combinations of conditions that might reveal logic errors. Crucially, this strategy lets LLMs primarily focus on assertion generation that verifies the logical consistency of outputs. For instance, when testing a method with complex boolean logic, it generates assertions that check logical invariants and validates that equivalent logical expressions produce identical results.
- *State Mutation Testing* specifically targets failures that arise from unexpected state changes during execution. This strategy is designed to modify object states between method calls or during iteration, revealing assumptions about state immutability. The expected assertions are to verify both exceptional behavior like ConcurrentModificationException and semantic correctness by checking that state changes produce expected outcomes or are properly rejected.
- *Exception Injection Testing* focuses on resource management and error handling paths. This strategy induces exceptions at critical points during execution and generates corresponding assertions that verify proper cleanup, error propagation, and recovery behavior. The assertions check not only that exceptions are handled but also that resources are released, partial operations are rolled back, and error states are correctly maintained.
- *Semantic Behavior Testing* leverages LLMs to understand intended behavior and generates assertions that verify semantic correctness. This strategy creates test cases that

Listing 2 CLI-265: Logic fault in Short Option Identification

```java
private boolean isShortOption(String token) {
    if (!token.startsWith("-") || token.length()
    ↪   == 1)
        return false;
    int pos = token.indexOf("=");
    // BUG: Long options incorrectly processed
    // Using substring(1) keeps the second '-'
    // in long options
    String optName = pos == -1 ?
    ↪   token.substring(1)
        : token.substring(1, pos);
    return options.hasShortOption(optName);
    // For input "--f=bar",
    // returns hasShortOption("-f")
    // instead of rejecting
}
```

explore the semantic boundaries of method behavior, with assertions that compare actual results against inferred expectations. Take Listing 2 as an example, a method named `isShortOption` fails to distinguish a long option from short ones. Therefore, based on the Incorrect Behavior failure's instructions in Section III-A1, the LLM generates assertions that verify the method returns true for short options like `-f` and false for long options like `-foo`, validating the semantic contract implied by the method name.

- *Edge Case Testing* ask LLMs to provide comprehensive coverage of special values with assertions that verify both exception handling and correct processing for code under test. When testing with empty strings or collections, this strategy aims to generate assertions that verify whether the code treats empty as a special case, returns appropriate default values, or throws expected exceptions.
- *Format Variation Testing* specifically addresses string processing failures by testing various input formats and generating assertions that verify parsing correctness, encoding consistency, and comparison semantics. The strategy asks LLMs to create assertions that check whether string comparisons use value equality rather than reference equality, whether parsing handles different formats consistently, and whether encoding transformations preserve data integrity.

Each strategy maintains a feedback mechanism that tracks not only its success rate in triggering failures but also the effectiveness of its generated assertions in distinguishing real bugs from expected behavior. This feedback guides the MCTS in allocating computational resources to the most promising strategy combinations for each specific code context.

C. Failure-Aware MCTS

The systematic identification of failure scenarios and their corresponding test strategies presents a new challenge of optimally selecting and applying these strategies to maximize failure triggering. Each code context requires different strategy combinations, and the effectiveness of a strategy can only be determined through actual test execution. This creates a complex sequential decision-making problem where each choice influences subsequent options and outcomes.

Our Failure-Aware Monte Carlo Tree Search (FA-MCTS) extends standard MCTS by incorporating failure scenario knowledge directly into the search process. The key innovation lies in biasing the search toward code regions with high failure potential while maintaining sufficient exploration to discover unexpected bugs. The algorithm uses a modified UCT formula that includes a failure-aware component:

$$UCT(v_i) = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}} + \lambda R(s) \tag{1}$$

Where the UCT formula for node v_i consists of three main components. The first term $\frac{w_i}{n_i}$ represents the exploitation aspect, where w_i denotes the total reward obtained by node i and n_i indicates the number of times node i has been visited. The second term $c\sqrt{\frac{\ln N_i}{n_i}}$ serves as the exploration component, with c being the exploration constant that balances exploration and exploitation, N_i representing the visit count of the parent node, and n_i again being the visit count of node i. The third term $\lambda R(s)$ introduces our logic-aware reward component, where λ controls the weight of bug-specific rewards, $R(s)$. This formula aims to balance exploration-exploitation tradeoffs while incorporating failure considerations.

The FA-MCTS algorithm operates through four iterative phases, each adapted for failure-driven test generation. During the *Selection phase*, the algorithm traverses the tree from the root using the modified UCT formula to identify the most promising node for expansion. This selection process prioritizes nodes representing code regions with identified failure scenarios while maintaining exploration of untested areas. The *Expansion phase* adds new nodes representing unexplored test strategies applicable to the current test state. Here, our failure scenario analysis guides the prioritization of strategies. For instance, when expanding a node representing a method with string operations, Format Variation Testing and Edge Case Testing receive higher priority for expansion. The expansion process considers both the primary and secondary strategies from Table II, creating a child node for an applicable strategy. During the *Simulation phase*, the algorithm evaluates the test strategy's effectiveness through actual test generation and execution. The simulation captures multiple signals from JUnit [42], including whether an exception was thrown, what type of exception occurred, whether assertions passed or failed, and whether the behavior matches expected semantics. This rich feedback distinguishes our approach from coverage-oriented methods that only track successful execution. The *Backpropagation phase* updates the statistics of all nodes along the path from the simulated node back to the root. The update incorporates both rewards and failure-specific information.

The reward function plays a crucial role in guiding the search toward effective bug detection. We design a composite reward function that balances multiple objectives:

$$R(s) = \alpha \cdot R_{cov}(s) + \beta \cdot R_{FSC}(s) + \gamma \cdot R_{REF}(s) + \delta \cdot R_{LSF}(s) \tag{2}$$

Where $R_{cov}(s)$ measures code coverage, $R_{FPC}(s)$ rewards Failure Scenario Coverage (FSC), $R_{REF}(s)$ rewards REF detections and $R_{LSF}(s)$ rewards LSF detection. The weights α, β, γ, and δ control the importance of each component.

The integration of failure scenarios and testing strategies into MCTS provides several key advantages. First, it dramatically reduces the effective search space by focusing on strategies likely to trigger failures rather than exploring all possible test inputs. Second, the learning mechanism allows the algorithm to adapt to project-specific characteristics, discovering which strategies prove most effective for particular code bases and failure types. Third, the probabilistic nature of MCTS ensures robustness against local optima that might trap deterministic search algorithms.

D. Multi-Level Failure Verification

The effectiveness of automated bug detection is often undermined by high false positive rates, particularly when using LLMs for test generation. To address this challenge, FAILMAPPER implements a multi-level verification mechanism that validates fault candidates through progressively sophisticated analysis techniques. This mechanism combines scenario-based filtering, static analysis verification, and LLM-enhanced semantic analysis to distinguish real bugs from spurious test failures.

1) Scenario-Based Verification: The first verification level employs lightweight scenario matching to quickly filter obvious false positives. This stage examines test failures against nine failure-specific scenarios, each associated with characteristic failure signatures. For instance, when a test reports a `NullPointerException`, the verifier checks whether the test explicitly sets null values (indicating intentional null testing) versus encountering unexpected nulls (indicating a potential bug). The failure-based verifier maintains specialized verification strategies for each fault category with calibrated confidence thresholds. For example, `ArrayIndexOutOfBoundsException` scenario achieves 95% confidence when tests explicitly use assertThrows, while boundary condition failures require additional context analysis, yielding 70-80% confidence. This differentiated approach allows rapid filtering of common false-positive scenarios, such as assertion failures due to incorrect test expectations (e.g., expecting null vs. empty string), tests that intentionally trigger exceptions to verify error handling, and boundary tests that expose documented limitations rather than bugs.

2) Static Analysis Verification: The second verification level performs deeper code structure analysis to validate failure candidates against implementation characteristics. For each potential failure, the verifier extracts relevant code patterns and analyzes their properties in relation to the specific failure type. For an example of concurrent-related failures, the verifier examines loop constructs and index calculations, checking for off-by-one patterns like `i <= array.length` instead of `i < array.length`. Another example of null reference faults, FAILMAPPER analyzes the presence and placement of null validation checks. The static analyzer correlates test failures with specific code structures. If a test reports an index error but the source code contains proper boundary validation (e.g., `if (index < 0 || index >= array.length)`), the confidence in the bug decreases.

3) LLM-Enhanced Semantic Verification: The highest verification level performs semantic analysis to understand intended behavior versus actual implementation. We extract semantic relationships between variables, examine API contracts, and analyze parameter constraints through LLM-enhanced code analysis. This level is particularly effective for business logic bugs where the flaw lies in the gap between intended and implemented behavior rather than in syntactic structures. Semantic verification produces the highest-confidence bug validations by incorporating contextual understanding of code purpose.

IV. EVALUATIONS

To evaluate the effectiveness of FAILMAPPER, we conducted a comprehensive series of experiments designed to assess its performance across multiple dimensions, including bug detection capability, false positive rates, code coverage, and computational efficiency. Our evaluation addresses the following research questions:

- **RQ1: Bug Detection Effectiveness.** How effective is FAILMAPPER at detecting bugs compared to state-of-the-art test generation approaches?
- **RQ2: Coverage Influence.** How does FAILMAPPER's focus on failure scenario targeting impact code coverage compared to coverage-driven approaches?
- **RQ3: False Positive (FP) Reduction.** To what extent does FAILMAPPER's multi-level verification framework reduce FPs compared to existing techniques?
- **RQ4: Generalizability Beyond Defects4J.** How well does FAILMAPPER generalize to bugs beyond the Defects4J benchmark?
- **RQ5: MCTS Efficiency Analysis.** How does the number of MCTS iterations affect bug detection, and at which iteration are most real bugs discovered?
- **RQ6: Component Contribution.** What are contributions of the individual components to FAILMAPPER's overall performance?

A. Experimental Setup

1) Bug Benchmark Selection: We evaluated FAILMAPPER on 50 bugs from the Defects4J benchmark [16] with 5 Apache projects (Cli, Codec, Csv, Math, and Compress). For each project, we selected the 10 most recent active bugs related to unit testing. This selection ensures our evaluation covers diverse bug types and coding patterns across different domains. Additionally, to examine the generalizability of FAILMAPPER across diverse domains and datasets, we evaluated FAILMAPPER on 20 bugs from Bears-benchmark [43] and GitBug-Java [44] with 19 different projects.

2) Baseline Comparison: We compared FAILMAPPER against five state-of-the-art test generation tools representing different approaches:

- **EvoSuite** [3]: A search-based testing tool that uses evolutionary algorithms to maximize code coverage.
- **Randoop** [4]: A feedback-directed random testing tool configured with a 20-minute time limit per class.
- **CoverUp** [6]: An LLM-based approach that iteratively generates tests to cover uncovered code segments.
- **HITS** [7]: An LLM-based tool that uses method slicing to generate tests for fine-grained code fragments.
- **SymPrompt** [8]: An LLM-based approach that structures code based on execution paths for test generation.

For the LLM-based baselines, we used their default configurations with 5 LLM requests per method under test, following their original experimental settings.

3) MCTS Configuration and Reward Function: We configured our FA-MCTS with parameters specifically tuned for failure scenario targeting. The key parameters for FA-MCTS include setting i) maximum iterations to 30; ii) exploration constant c to 1.0, balancing exploration versus exploitation; and iii) failure weight λ to 2.0, emphasizing failure-specific rewards. The reward function is designed in Eq. 2 with three hyperparameter: $\alpha = 0.2$, $\beta = 0.3$, $\delta = 0.25$. This emphasizes failure detection over raw coverage, differentiating our approach from traditional coverage-driven methods. All hyperparameters were selected based on a grid search to determine the most effective combination.

4) Implementation Details: All experiments were conducted on identical hardware using AMD Ryzen 9 5950X processors with 64GB RAM running Ubuntu 22.04 LTS. Given the deterministic nature of FAILMAPPER's failure scenario targeting approach, we conducted single runs to ensure fair comparison. As shown in Table III, FAILMAPPER's average testing time on Defects4J was 18.8 minutes per class. Therefore, EvoSuite and Randoop were each allocated 20 minutes per class to match FAILMAPPER's time cost for fair comparisons. EvoSuite was run with its default multi-criteria optimization with eight coverage criteria: Line, Branch, Exception, WeakMutation, Output, Method, MethodNoException, and CBranch [26]. This configuration represents EvoSuite's standard operating mode and has been shown to achieve the best overall results in previous studies [45]. For LLM-based baselines, we used the Claude 3.5 Sonnet model via the Anthropic API [46]. We followed their default configurations of 5 API requests per method, as specified in their original implementations [47]. For coverage measurement, we employed JaCoCo version 0.8.8 [48], measuring instruction coverage at the bytecode level. This code coverage metric counts executed JVM instructions, providing more precise measurement than line or statement coverage while remaining comparable across different code styles and formatting conventions.

B. *RQ1: Bug Detection Effectiveness*

Table IV shows that FAILMAPPER significantly outperforms all baselines, detecting an average of 8/10 bugs compared to 2.4/10 for SymPrompt (best baseline), a 233% improvement. This validates our hypothesis that targeting failure scenarios is more effective than maximizing coverage. For instance, in the Cli project, FAILMAPPER detected all 10 bugs with 84% coverage, while EvoSuite achieved the same coverage but detected 0 bugs. This demonstrates that the quality of test inputs matters more than the quantity of code executed. FAILMAPPER's specialized testing strategies generate inputs specifically designed to trigger failure scenarios such as null references, boundary violations, and resource management issues that are prevalent in command-line parsing applications. For Codec and Compress, FAILMAPPER detected 9 bugs each versus 4 and 2 for the best baselines.

TABLE III: Computational Cost Comparison across Test Generation Approaches on Defects4J

Tool	Time/Class (min)	API Requests	Tokens (k)
CoverUp	31.2	145.3	215.6
HITS	28.4	138.7	198.3
SymPrompt	25.6	132.4	186.7
FAILMAPPER	**18.8**	**40.9**	**182.0**
Efficiency Improvement of FAILMAPPER *vs. Each Baseline*			
vs. CoverUp	40% faster	72% fewer	16% fewer
vs. HITS	34% faster	70% fewer	8% fewer
vs. SymPrompt	27% faster	69% fewer	3% fewer

Interestingly, EvoSuite shows strong performance on the Math project (10 bugs detected), comparable to traditional testing approaches for mathematical software. This can be attributed to the nature of mathematical bugs, which often manifest as arithmetic exceptions or boundary violations that evolutionary algorithms can discover through systematic numerical exploration. However, FAILMAPPER's weaker detection performance on Math (5 bugs) suggests that some failure scenarios in mathematical computations require domain-specific strategies beyond our current nine categories.

The contrast in bug detection rates between FAILMAPPER and LLM-based baselines reveals limitations in current LLM-based test generation. CoverUp, HITS, and SymPrompt achieve similar low detection rates (1.0, 0.6, and 2.4 bugs on average) despite using various and sophisticated prompting strategies. This suggests that without explicit guidance toward failure scenarios, LLMs tend to generate tests that examine code successfully rather than expose failures. Additionally, resource consumption for LLM-based tools was tracked throughout all experiments to ensure fair comparison and practical viability. Table III demonstrates that FAILMAPPER has 27% faster execution and 3% fewer token inputs compared to LLM baselines, while achieving 3.3 times higher bug detection rates.

We manually examined the 10 bugs that FAILMAPPER failed to detect. These bugs primarily fall into two categories: (1) bugs requiring highly specific input combinations that are statistically unlikely to be generated, such as particular floating-point values that trigger precision errors only at specific ranges; and (2) bugs in mathematical computations requiring domain-specific invariants that are difficult to infer without explicit mathematical specifications. For instance, one undetected bug in the Math project required inputs satisfying a

TABLE IV: Experimental Results Comparison on Defects4J with 50 Bugs

Projects	SBST-based Techniques						LLM-based Techniques									Ours		
	EvoSuite			Randoop			CoverUp			HITS			SymPrompt			FAILMAPPER		
	Bugs	FP	Cov.	Bugs	FP	Cov.	Bugs	FP	Cov.	Bugs	FP	Cov.	Bugs	FP	Cov.	Bugs	FP	Cov.
Cli	0	-	**84%**	1	-	76%	2	12	78%	1	15	78%	2	11	82%	**10**	**3**	**84%**
Codec	0	-	**94%**	1	-	82%	1	35	87%	1	36	87%	4	32	75%	**9**	**7**	88%
Csv	1	-	**89%**	3	-	76%	0	51	81%	0	68	84%	2	54	86%	**7**	**13**	79%
Math	**10**	-	**91%**	1	-	81%	1	47	88%	1	95	81%	2	35	82%	5	**20**	83%
Compress	0	-	72%	0	-	63%	1	35	**77%**	0	141	71%	2	36	71%	**9**	**15**	72%
Average	2.2	-	**86%**	1.2	-	75.6%	1.0	36	82%	0.6	71	80%	2.4	34	79%	**8.0**	**12**	81%

Note: Bugs are bug detection rate; FP is false positive rate; Cov. is code coverage; ***Bold*** *means the best performance.*

specific mathematical relationship (e.g., values that cause numerical instability in matrix decomposition) that our general-purpose strategies did not target.

C. *RQ2:* Coverage Influence

A concern when shifting from coverage-driven to failure-driven test generation is the potential impact on code coverage. Table IV shows that FAILMAPPER achieves an average code coverage of 81% across all projects, which is lower than EvoSuite's 86% but comparable to other LLM-based approaches (CoverUp 82%, HITS 80%, SymPrompt 79%). This result reveals important insights about the relationship between code coverage optimization and bug detection effectiveness.

The 5% code coverage gap between FAILMAPPER and EvoSuite reflects a fundamental difference in testing philosophy. Our results demonstrate that this trade-off is worthwhile that despite lower code coverage, FAILMAPPER detects significantly more bugs (40 versus 11 for EvoSuite). The most striking example of this phenomenon occurs in the Cli and Codec projects. On Cli, both FAILMAPPER and EvoSuite achieve 84% code coverage, and FAILMAPPER detects all 10 bugs while EvoSuite detects none. The comparison with other LLM-based approaches is particularly revealing. Despite similar code coverage levels, FAILMAPPER dramatically outperforms CoverUp, HITS, and SymPrompt in bug detection.

FAILMAPPER achieves "meaningful code coverage" by targeting bug-prone regions: error handling, boundary conditions, resource management, validation logic, and so on. This represents a paradigm shift from metrics-driven to risk-based testing. While organizational policies often mandate 80% or higher code coverage [49], our results show that optimizing for raw coverage is counterproductive. FAILMAPPER's balanced approach (81% code coverage with superior bug detection) satisfies code coverage requirements while maximizing bug-finding effectiveness.

D. *RQ3:* False Positive Reduction

False positives refer to incorrect defect reports caused by unreasonable inputs or incorrect expectations, representing a significant challenge in LLM-based bug detection. We manually analyzed all generated cases, failure reports, and reported bug documents on Defects4J to verify false positives. As shown in Table IV, existing LLM-based tools suffer from high false positive rates: CoverUp generates an average of 36 false positives, HITS produces 71, and SymPrompt generates 34 for each project. In contrast, FAILMAPPER achieves an average of only

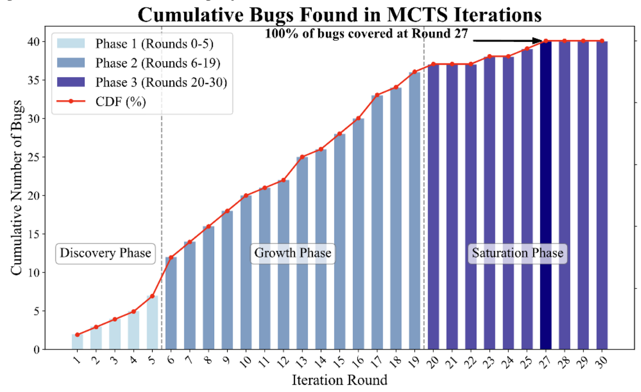


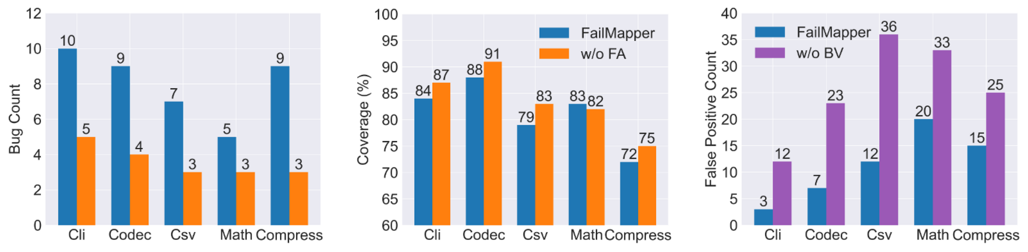
Fig. 3: Cumulative Bugs Found in MCTS Iterations.

12 false positives for 10 classes, representing a 64.7% reduction compared to SymPrompt (the best-performing baseline) and an 83.1% reduction compared to HITS. With traditional LLM-based approaches generating 34 to 71 false positives on average, the manual review burden often outweighs the benefits of automated testing. The reduction of FAILMAPPER to 12 false positives in 10 classes (*average 1.2 for each class*) makes automated bug detection significantly more practical for real-world development workflows.

E. *RQ4:* Generalizability Beyond Defects4J

To evaluate the generalizability of our failure scenarios, we conducted experiments on two independent benchmarks: GitBug-Java [44], containing reproducible Java bugs from 2020-2024, and Bears-benchmark [43], comprising bugs from diverse domains. Data in these benchmarks is totally different from Defects4J for an unbiased assessment of FAILMAPPER's generalizability. Table V presents that FAILMAPPER successfully detected 12 out of 20 bugs from these benchmarks. The detected bugs span all 9 failure scenarios identified in our taxonomy, validating that these patterns represent fundamental failure modes in Java software. The consistent performance across diverse codebases validates software failures manifest through systematic patterns.

FAILMAPPER has identified 36 potential issues in the latest commits of Cli [17] and Math [20]. Among them, 14 have been confirmed as bugs by Apache, including 1 critical bug. So far, 7 of the confirmed bugs have been fixed, and 1 is currently under repair. Among the 22 potential issues that were not accepted, many still represent meaningful findings despite not being classified as bugs. For instance, some issues involve non-public interfaces that only manifest in unit tests rather than actual execution paths. In the Math library, the



(a) Bug Count with MCTS (b) Code Coverage (w/o FA) (c) False Positive (w/o BV)

Fig. 4: Ablation study results showing the impact of key components. (a) Bug detection count comparing complete FAILMAPPER with version without failure awareness (w/o FA), demonstrating significant drops across all projects. (b) Coverage comparison showing increased code coverage when failure awareness is removed, indicating the trade-off between code coverage and bug detection. (c) False positive rates with and without bug verification (w/o BV), highlighting the crucial role of multi-level verification in reducing spurious test failures.

TABLE V: FAILMAPPER's Performance on Independent Benchmarks Demonstrating Failure Generalizability

Benchmark	Project/Bug ID	Detected	Root Cause	Failure Scenario
GitBug-Java	assertj-vavr	✗	API Contract Violations	2.1 Incorrect Behavior
	aws-secretsmanager-jdbc	✓	Input Validation	2.3 Data Integrity
	beanshell	✗	API Contract Violations	2.1 Incorrect Behavior
	cloudsimplus	✗	Parameter Mismatch	2.1 Incorrect Behavior
	ConfigMe	✓	Property Validation	2.3 Data Integrity
	crawler-commons	✓	Character Encoding	2.4 String Processing
	database-engine	✓	Uncaught Exception	1.3 Resource Management
	dataframe-ec	✗	String Encoding	2.4 String Processing
	dotenv-java	✓	Variable Misuse	2.2 Logic Assertion
	JSONata4Java	✓	Type Conversion	2.5 Numeric Computation
Bears	spotify-web-api (246)	✗	Argument Order	2.6 Configuration Dependent
	javapoet (245)	✓	Input Validation	2.3 Data Integrity
	json-ignore (238)	✗	API Contract Violations	2.1 Incorrect Behavior
	cash-count (234)	✓	Arithmetic Error	2.5 Numeric Computation
	oss-parent (232)	✓	Out of Boundary	1.2 Index Boundary
	classgraph (202)	✓	Null Reference	1.1 Null Reference
	aws-encryption (198)	✓	Null Reference	1.1 Null Reference
	pmd-gds (194)	✗	Configuration Load	2.6 Configuration Dependent
	cassandra (192)	✗	API Contract Violations	2.1 Incorrect Behavior
	cassandra (191)	✓	State Management	1.4 Concurrent Modification
Overall Detection Rate		**12/20 (60%)**	Coverage: 9/9 failure scenarios	

`AccurateMathCalc` class is package-private and fails to handle the special case when the input is zero, returning negative infinity, which is mathematically incorrect. However, in real execution flows, the inputs are constrained to the valid range of 1 to 2, thereby preventing such cases from arising. Newly discovered bugs also prove that the failure-aware method used in FAILMAPPER can guide LLMs to uncover new bugs, rather than relying on the LLM's training memory to identify existing ones.

F. *RQ5: MCTS Efficiency Analysis*

Figure 3 presents the cumulative bug discovery across successive MCTS iterations for the 50 bugs in our benchmark. The results reveal several insights regarding the computational efficiency and optimal configuration of our approach. First, we observe that FAILMAPPER successfully detected 40 bugs out of the 50 benchmark bugs (80%), with all 40 bugs being discovered within 27 iterations. This finding motivated our choice of 27 as the optimal iteration parameter for FAILMAPPER, as it represents the point of diminishing returns for additional computational investment.

During the first 5 iterations, bug discovery proceeds gradually as the initial discovery phase, with 7 bugs (17.5% of the total) being identified. This initial phase represents the exploration period where the MCTS algorithm is building its understanding of the program structure and identifying potential failure scenarios. The middle phase, from iteration 6 to 19, shows an acceleration in bug discovery, with 28 additional bugs (70% of the total) being discovered. This represents the most efficient period of the search, where the algorithm leverages insights gained during the exploration phase to target high-probability bug locations. In the final saturation phase, the discovery rate slows, with only 5 more bugs (12.5% of the total) being found.

G. *RQ6: Component Contribution*

This ablation study investigates the individual contributions of FAILMAPPER's key components to its overall performance. We removed failure awareness and bug verification components, respectively, to measure the impact on bug detection, code coverage, and FPRs. Figure 4 presents the results of this analysis.

1) Impact of FA: In this ablation, we only allow MCTS to exploit bugs with a general action, but no failure-scenario guidance or a failure-aware formula. Figure 4a shows that bug detection capability drops significantly across all projects:

from 10 to 5 bugs in Cli (50% reduction), from 9 to 4 bugs in Codec (56% reduction), from 7 to 3 bugs in Csv (57% reduction), from 5 to 3 bugs in Math (40% reduction), and from 9 to 3 bugs in Compress (67% reduction). This substantial degradation demonstrates that failure awareness is a critical factor in FAILMAPPER's bug detection performance. Note that, FAILMAPPER still performs better than other tools shown in Table IV in detecting bugs. This improvement contributes to the exploitation-exploration balance by MCTS, which allows LLM to spend efforts in seeking bugs while sticking to the code coverage convergence.

Additionally, as FA-MCST sometimes repeatedly exploits specific failure scenarios, it will sacrifice some computing resources in exploration. Therefore, as shown in Figure 4b, removing failure awareness components generally leads to higher code coverage percentages.

2) Impact of Bug Verification (BV): Figure 4c demonstrates the crucial role of our multi-level bug verification system in reducing false positives. Without verification, FPRs increase dramatically: from 3 to 12 for Cli (300% increase), from 7 to 23 for Codec (229% increase), from 13 to 36 for Csv (177% increase), from 20 to 33 for Math (65% increase), and from 15 to 25 for Compress (67% increase). This contrast highlights how essential verification is for practical application. Without structured verification, test failures resulting from incorrect expectations rather than actual bugs flood the results. The multi-level verification framework effectively filters out these spurious reports while preserving real bug detections.

V. DISCUSSION

FAILMAPPER represents a fundamental paradigm shift in automated test generation. Traditional approaches, whether SBST or LLM-based, operate under the assumption that maximizing the coverage metrics correlates with bug detection. Our work challenges this assumption by demonstrating that targeting specific failure scenarios is significantly more effective for bug discovery. This shift from "what code should do" (infinite specification space) to "how code might fail" (bounded failure scenarios) transforms test generation from an unbounded search problem to a tractable task.

The success of this approach stems from a key insight validated by our empirical study: while software requirements are infinitely diverse, failure manifestations follow systematic patterns. By identifying and targeting these patterns, FAILMAPPER achieves a 233% improvement in bug detection while maintaining reasonable code coverage. This suggests that decades of focus on coverage metrics may have been optimizing for the wrong objective.

Our research opens several promising directions for future investigation. First, we plan to expand and formalize the failure scenario taxonomy beyond the current nine categories. This includes developing formal specifications for each failure scenario and investigating domain-specific failure patterns in specialized software systems. Second, we aim to conduct more empirical evaluations on a broader range of projects beyond the current benchmark. This expanded evaluation would include projects from different domains, with varying sizes, complexities, and development practices to assess the generalizability of our approach.

VI. RELATED WORK

A. SBST

Within SBST, the primary objective is to navigate this complex search space to identify test cases that best satisfy predefined testing goals. EvoSuite [3] employs evolutionary algorithms to automatically generate test suites that aim to achieve high code coverage. For the dynamically-typed language Python, Pynguin [5] relies on type annotations to infer the expected data types for test inputs to address the unique challenges posed by Python's dynamic nature. Randoop [4] offers an alternative approach to automated unit test generation through feedback-directed random testing by relying on fitness functions and evolutionary algorithms. DynaMOSA [25] is a specific many-objective evolutionary algorithm designed to tackle the test case generation problem, particularly in the context of code coverage testing. However, SBST-generated tests can sometimes lack readability and may not always effectively target complex semantic aspects of the code [45].

B. LLM-Based Test Generation

Recent work leverages LLMs to generate unit tests by understanding code semantics. ChatUnitest [50] resolves the problem of invalid test case generation through a tailored rule-based repair mechanism and by feeding compilation error messages back to LLMs. With the validity of code generation guaranteed, subsequent work is mainly dedicated to explore more code coverage. CoverUp [6] prompts the LLM with under-covered code segments and iteratively refines tests until reaching high code coverage. HITS [7] decomposes complex methods into code slices and asks the LLM to generate tests per slice, dramatically improving branch coverage over template prompts. SymPrompt [8] introduces multi-stage, code-aware prompting so the LLM steps through a method's logic. Compared to SBST tools, LLM-based approaches generate more natural and readable test cases. However, existing studies fail to detect bugs effectively [51], [52].

VII. CONCLUSION

In this paper, we present FAILMAPPER, a novel failure-aware unit testing framework that supports the automated generation of unit tests effective in triggering a wide variety of bugs. In particular, FAILMAPPER bridges the gap between code coverage and bug manifestation, which has not been well addressed by existing testing approaches. Our experimental evaluation demonstrated that FAILMAPPER has substantially outperformed the baseline techniques, including two mainstream SBST methods and three contemporary LLM-based methods and achieving a 233% improvement over the best-performing baseline. Additionally, FAILMAPPER reduced the false positive rate by 64.7% while maintaining comparable code coverage. The ablation studies confirmed the crucial contributions of both the failure-aware components and the bug verification framework to the overall performance.

References

[1] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, 2024.

[2] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 201–211.

[3] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[4] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.

[5] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 168–172.

[6] J. A. Pizzorno and E. D. Berger, "Coverup: Coverage-guided llm-based test generation," *arXiv preprint arXiv:2403.16218*, 2024.

[7] Z. Wang, K. Liu, G. Li, and Z. Jin, "Hits: High-coverage llm-based unit test generation via method slicing," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1258–1268.

[8] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage-guided test generation in regression setting using llm," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 951–971, 2024.

[9] X. Zhu, W. Zhou, Q.-L. Han, W. Ma, S. Wen, and Y. Xiang, "When software security meets large language models: A survey," *IEEE/CAA Journal of Automatica Sinica*, 2024.

[10] F. Molina, A. Gorla, and M. d'Amorim, "Test oracle automation in the era of llms," *ACM Transactions on Software Engineering and Methodology*, 2024.

[11] X. Zhu and M. Böhme, "Regression greybox fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2169–2182.

[12] Amazon Web Services. (2023) What is unit testing? [Online]. Available: https://aws.amazon.com/what-is/unit-testing/

[13] Y. Wang, P. Zhang, M. Sun, Z. Lu, Y. Yang, Y. Tang, J. Qian, Z. Li, and Y. Zhou, "Uncovering bugs in code coverage profilers via control flow constraint solving," *IEEE Transactions on Software Engineering*, vol. 49, no. 11, pp. 4964–4987, 2023.

[14] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?" in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 55–64.

[15] M. Hamill and K. Goseva-Popstojanova, "Common trends in software fault and failure data," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 484–496, 2009.

[16] R. Just, D. Jalali, and M. D. Ernst, "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*, 2014, pp. 437–440.

[17] Apache Commons Cli, "Apache commons cli," https://github.com/apache/commons-cli, 2025, accessed: 2025-05-25.

[18] Apache Commons Codec, "Apache commons codec," https://github.com/apache/commons-codec, 2025, accessed: 2025-05-25.

[19] Apache Software Csv, "Apache commons csv," https://github.com/apache/commons-csv, 2025, accessed: 2025-05-25.

[20] Apache Commons Math, "Apache commons math," https://github.com/apache/commons-math, 2025, accessed: 2025-05-25.

[21] Apache Commons Compress, "Apache commons compress," https://github.com/apache/commons-compress, 2025, accessed: 2025-05-25.

[22] Apache Software Foundation, "[CSV-122] Reported Bug on Apache Commons Csv," https://issues.apache.org/jira/browse/CSV-122, 2014, accessed: 2025-03-16.

[23] Y. Liang, S. Liu, and H. Hu, "Detecting logical bugs of DBMS with coverage-based guidance," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 4309–4326. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/liang

[24] H. Kim, M. O. Ozmen, Z. B. Celik, A. Bianchi, and D. Xu, "Pgpatch: Policy-guided logic bug patching for robotic vehicles," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1826–1844.

[25] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.

[26] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining multiple coverage criteria in search-based unit test generation," in *International Symposium on Search Based Software Engineering*. Springer, 2015, pp. 93–108.

[27] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 147–158.

[28] Y. Tang, Z. Liu, Z. Zhou, and X. Luo, "Chatgpt vs sbst: A comparative assessment of unit test suite generation," *IEEE Transactions on Software Engineering*, 2024.

[29] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 913–923.

[30] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "Not all bugs are the same: Understanding, characterizing, and classifying bug types," *Journal of Systems and Software*, vol. 152, pp. 165–181, 2019.

[31] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[32] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[33] G. M. J.-B. C. Chaslot, "Monte-carlo tree search," 2010.

[34] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008, pp. 329–339.

[35] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical software engineering*, vol. 19, pp. 1665–1705, 2014.

[36] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 510–520.

[37] D. Hovemeyer and W. Pugh, "Finding more null pointer bugs, but not too many," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2007, pp. 9–14.

[38] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 57–72, 2001.

[39] S. Nanz and C. A. Furia, "A comparative study of programming languages in rosetta code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 778–788.

[40] I. M. Copi, C. Cohen, and K. McMahon, *Introduction to logic*. Routledge, 2016.

[41] F. A. Bianchi, A. Margara, and M. Pezzè, "A survey of recent trends in testing concurrent software systems," *IEEE Transactions on Software Engineering*, vol. 44, no. 8, pp. 747–783, 2017.

[42] V. Massol, *JUnit in action*. Citeseer, 2004.

[43] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*, 2019. [Online]. Available: https://arxiv.org/abs/1901.06024

[44] A. Silva, N. Saavedra, and M. Monperrus, "Gitbug-java: A reproducible benchmark of recent java bugs," in *Proceedings of the 21st International Conference on Mining Software Repositories*.

[45] G. Fraser and A. Arcuri, “A large-scale evaluation of automated unit test generation using evosuite,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, pp. 1–42, 2014.

[46] Anthropic Claude, “Anthropic claude api,” https://www.anthropic.com/api, 2023, accessed: 2025-03-18.

[47] ZJU-ACES-ISE, “chatunitest-maven-plugin,” 2023, gitHub repository. [Online]. Available: https://github.com/ZJU-ACES-ISE/chatunitest-maven-plugin

[48] EclEmma Team, *JaCoCo: Java Code Coverage Library, version 0.8.8*, Apr. 2022, release date: April 5, 2022. Distributed under the Eclipse Public License 2.0. [Online]. Available: https://www.jacoco.org/

[49] A. K. Barua, “Test coverage definition - unit testing,” https://learn.microsoft.com/en-us/answers/questions/778016/test-coverage-definition-unit-testing, Mar. 2022, microsoft Q&A. [Online]. Available: https://learn.microsoft.com/en-us/answers/questions/778016/test-coverage-definition-unit-testing

[50] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, “Chatunitest: A framework for llm-based test generation,” in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 572–576.

[51] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, “An empirical evaluation of using large language models for automated unit test generation,” *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, 2023.

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