LLMs vs. Classical Tools for Unit Test Generation

An Exploratory Head to Head on Defects4J

捉虫特工队

{liujp2025, shenxx2025, zhaoyh32025, zhaozy12025}@shanghaitech.edu.cn

2025 年 12 月 19 日

Catalog

Catalog

Unit Testing

Unit testing, a.k.a. component or module testing, is a form of software testing which describes tests that are run at the unit-level to contrast testing at the integration or system level.

It is critical for software reliability but is labor-intensive and time-consuming, and developers struggle to maintain adequate coverage manually.

Limitations of Traditional Tools

Tools like EvoSuite and Randoop excel at code coverage but fail in quality:

1. Lack of Readability: They generate "spaghetti code" with random variable names, making it hard to maintain.
2. The Oracle Problem : They detect crashes but cannot verify functional correctness (business logic).

Large Language Model

A large language model (LLM) is a language model trained with self-supervised machine learning on a vast amount of text, designed for natural language processing tasks, especially language generation. The largest and most capable LLMs are generative pre-trained transformers (GPTs) and provide the core capabilities of modern chatbots. LLMs can be fine-tuned for specific tasks or guided by prompt engineering. These models acquire predictive power regarding syntax, semantics, and ontologies inherent in human language corpora, but they also inherit inaccuracies and biases present in the data they are trained on.

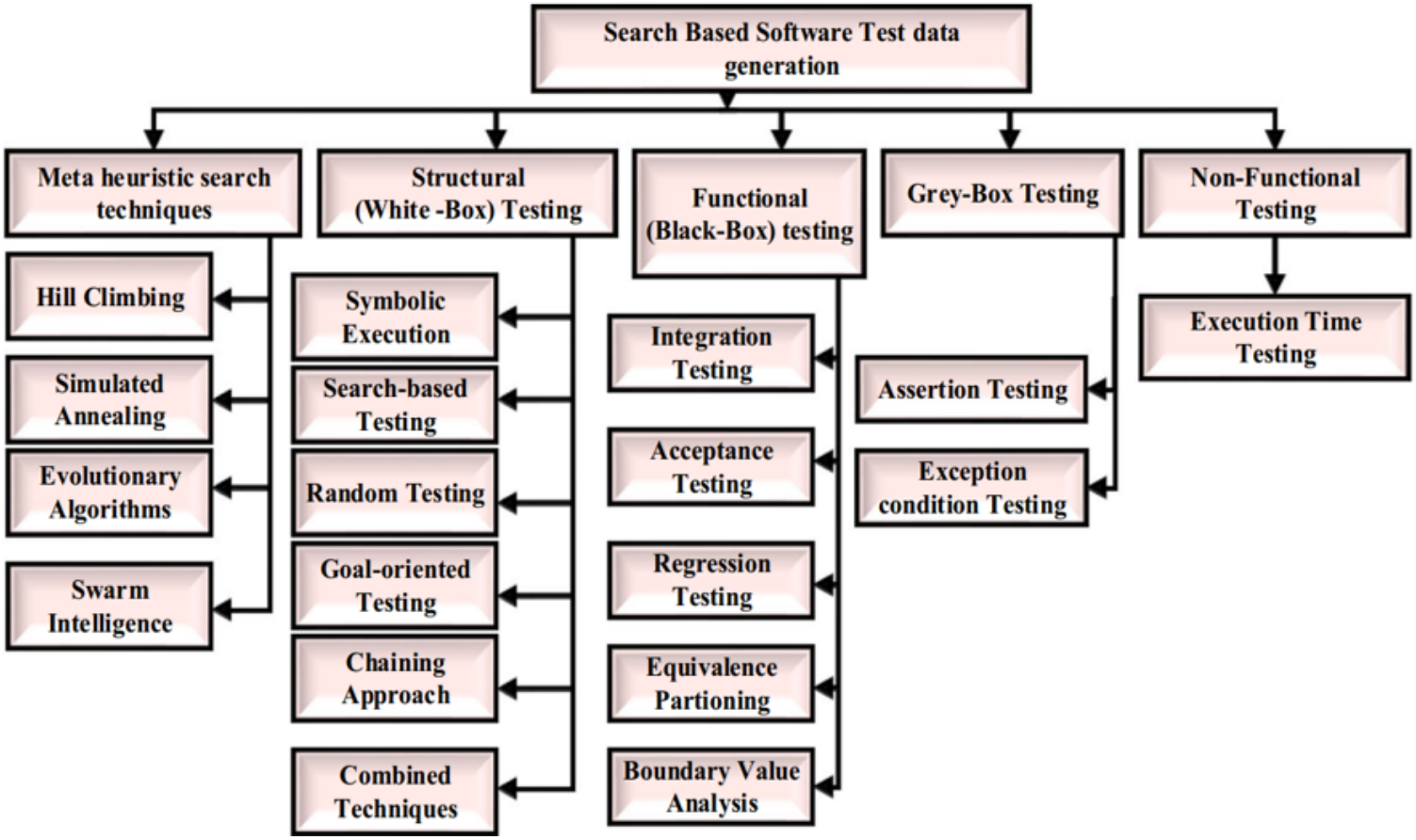
The Advantage of LLMs

1. Semantic Understanding: LLMs can analyze context to verify business logic, not just execution paths.
2. Human-Like Readability: They generate code with standard naming and structure, making it easier to debug and maintain.

Catalog

SBST

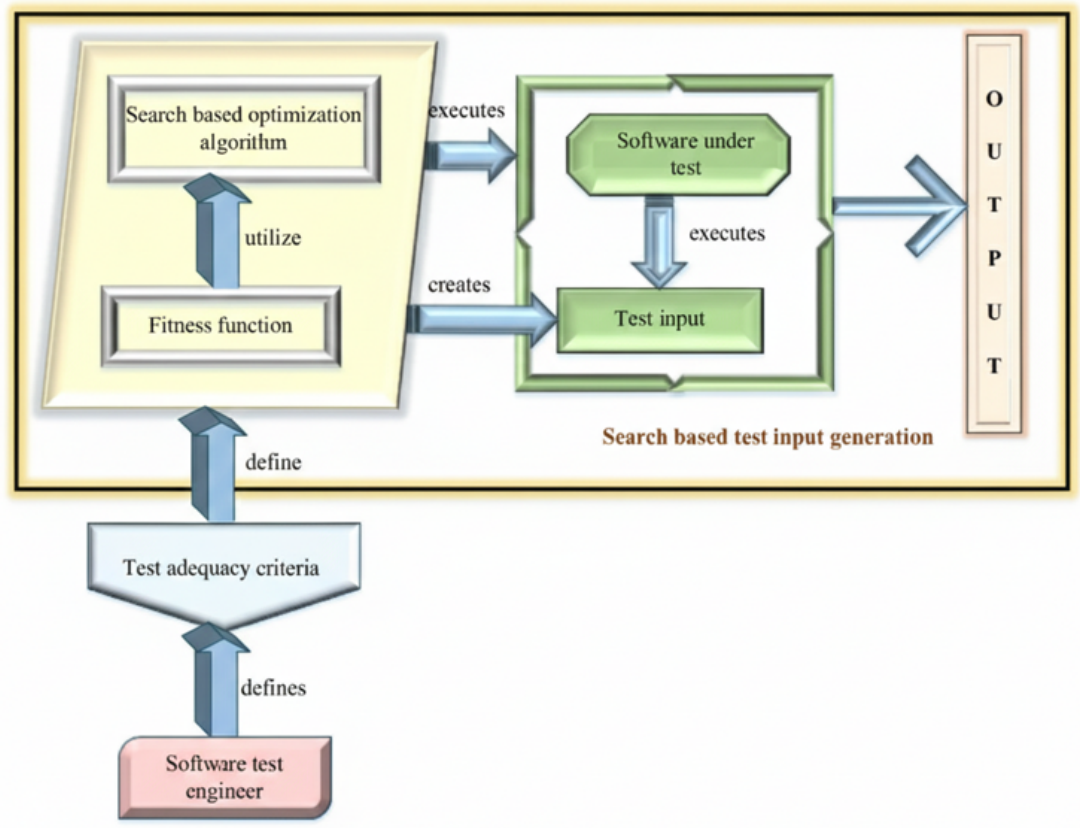
Search-based software testing (SBST), which has been widely employed, is a technique that employs search algorithms such as genetic algorithms and simulated annealing to create test cases.



SBST

Core Principles of SBST transforms software testing problems into search optimisation problems:

* Test objectives (such as code coverage) serve as optimisation targets.
* Test inputs constitute solutions within the search space.
* Search algorithms automatically identify optimal test cases.



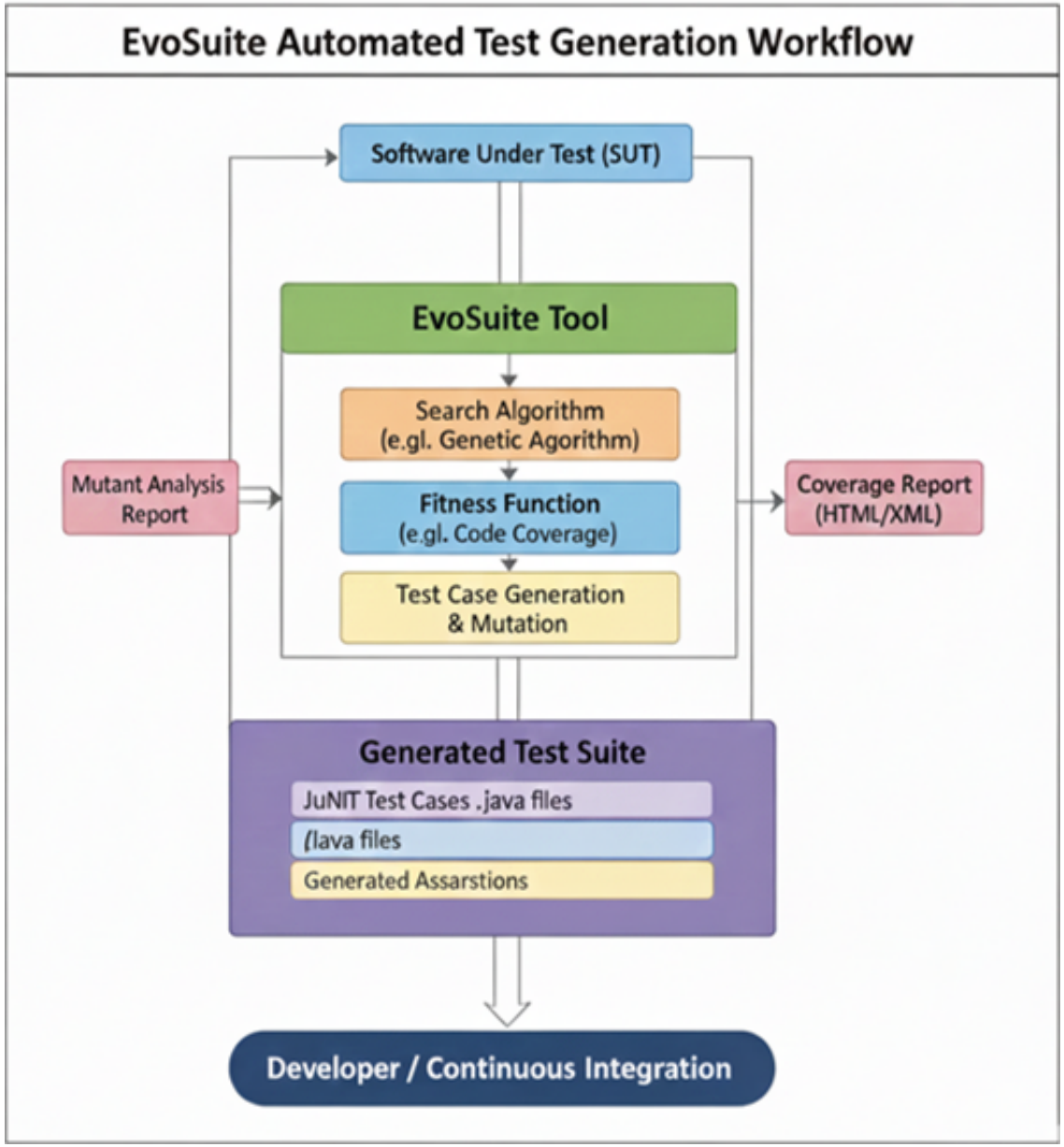
Evosuite

Input: Compiled Java bytecode (.class files or JAR packages).
Instrumentation: Insert probes into the bytecode to collect coverage data.
Population initialization: Generate random test sequences.
Evolutionary cycle:

* Execution and evaluation: Run tests and compute fitness metrics such as branch coverage.
* Genetic operations: Select superior individuals, perform crossover and mutation to generate a new population.

Test generation: Convert sequences from the final population into JUnit test code and attempt to generate assertions.
Output deliverable: A JUnit test class

Internal Processing Flow of Evosuite



Randoop

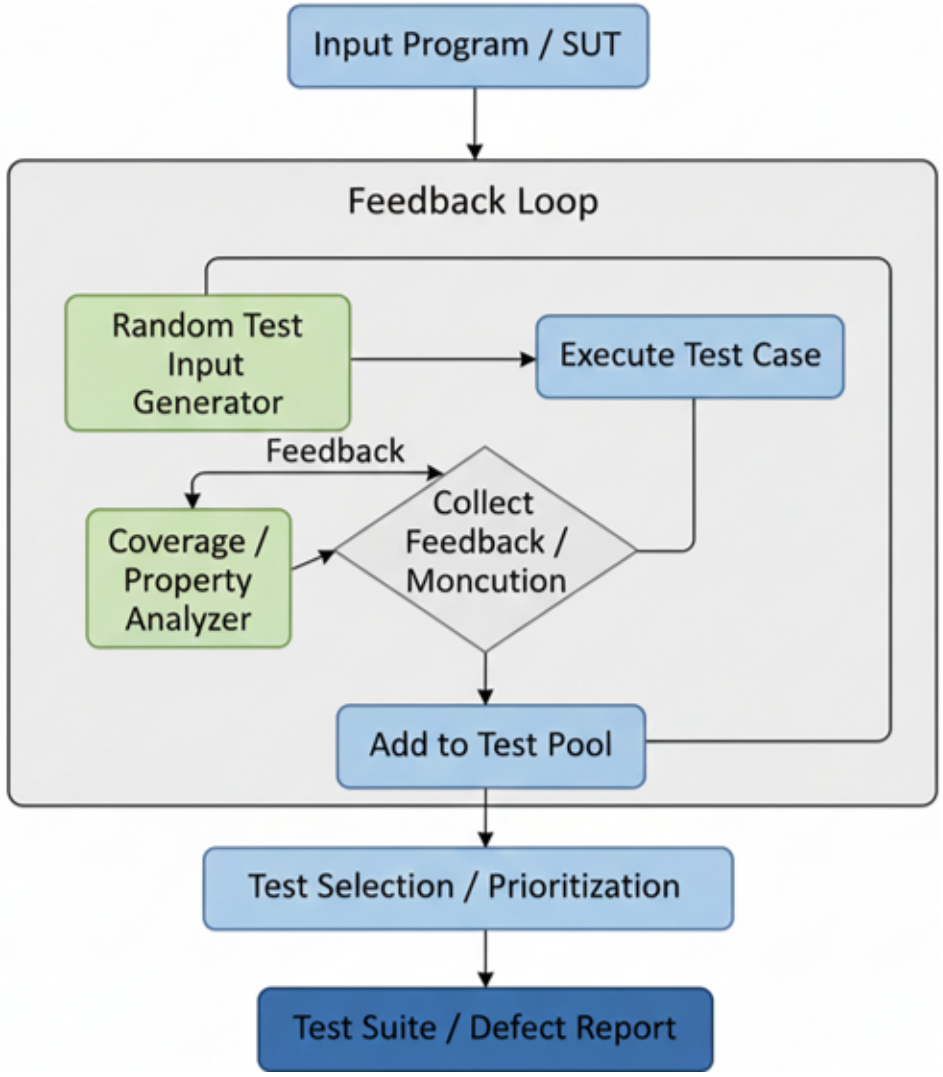
Randoop is based on ‘feedback-driven random testing’. Unlike EvoSuite, it lacks a global optimisation objective, instead operating through the iterative process — Randomly combine available methods/constructors into a new call sequence.

If the sequence triggers an uncaught exception → Save it as an error-revealing test.
If the sequence executes normally without producing duplicate behaviour → Save it as a regression test.
Otherwise, discard.

Randoop

Component Collection: Analyse bytecode to list all testable methods and constructors.
Random Extension: Create new sequences by randomly selecting methods based on existing objects and values.
Execution and Classification: Execute sequences and classify them based on feedback (normal/exception).
Assertion Generation: For normal sequences, append simple object equality (equals) or non-null (null) assertions at the end.
Output Product: A JUnit test class

Randoop


Input Program / SUT
Feedback Loop
Random Test Input Generator
Execute Test Case
Feedback
Coverage / Property Analyzer
Collect Feedback / Moncution
Add to Test Pool
Test Selection / Prioritization
Test Suite / Defect Report

Catalog

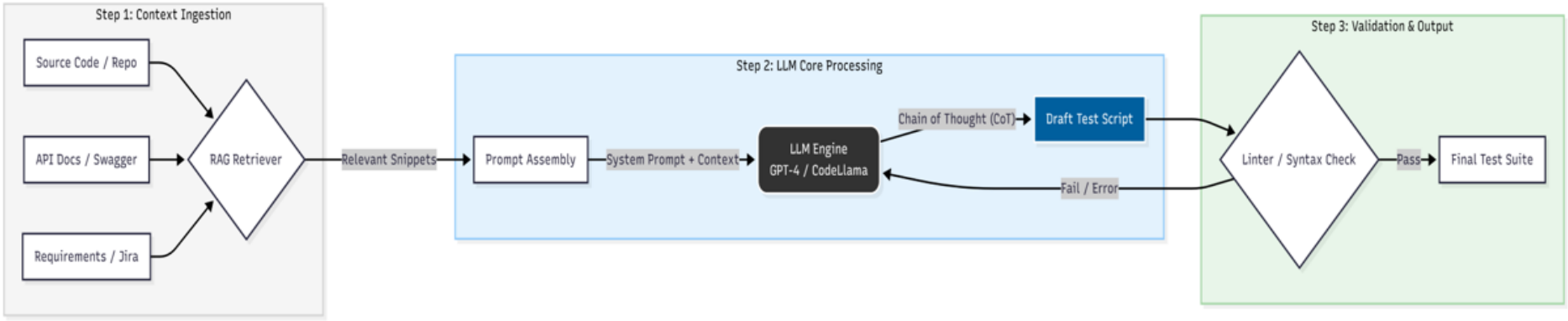
How LLMs Redefine Testing

LLMs introduce a fundamental shift from rule-based automation to cognitive, context-aware generation.

This isn't just about writing scripts faster; it's about enabling the machine to understand intent, infer behavior, and create meaningful validation strategies.

1. From Script Execution to Semantic Understanding.
2. The Power of In-Context Learning and Generation.
3. Transformation: Adaptive Test Strategy Generation.

Step 1 Context Ingestion (The Knowledge Base)
RAG Mechanism: A Vector Retriever filters noise, ensuring the LLM understands the business intent behind the code, not just the syntax.
Step 2 Core Processing (The Reasoning Engine)
Prompt Assembly: Dynamically combines relevant code snippets with system instructions. Chain of Thought (CoT): The LLM breaks down complex logic step-by-step to generate test scenarios that cover edge cases and business logic.
Step 3 Closed-Loop Validation (The Quality Gate)
Deterministic Checks: A Linter acts as a strict gatekeeper for syntax and compilation errors. Self-Correction Mechanism: Crucially, if validation fails, the error log is fed back to the LLM. The system iteratively rewrites the script until it passes.

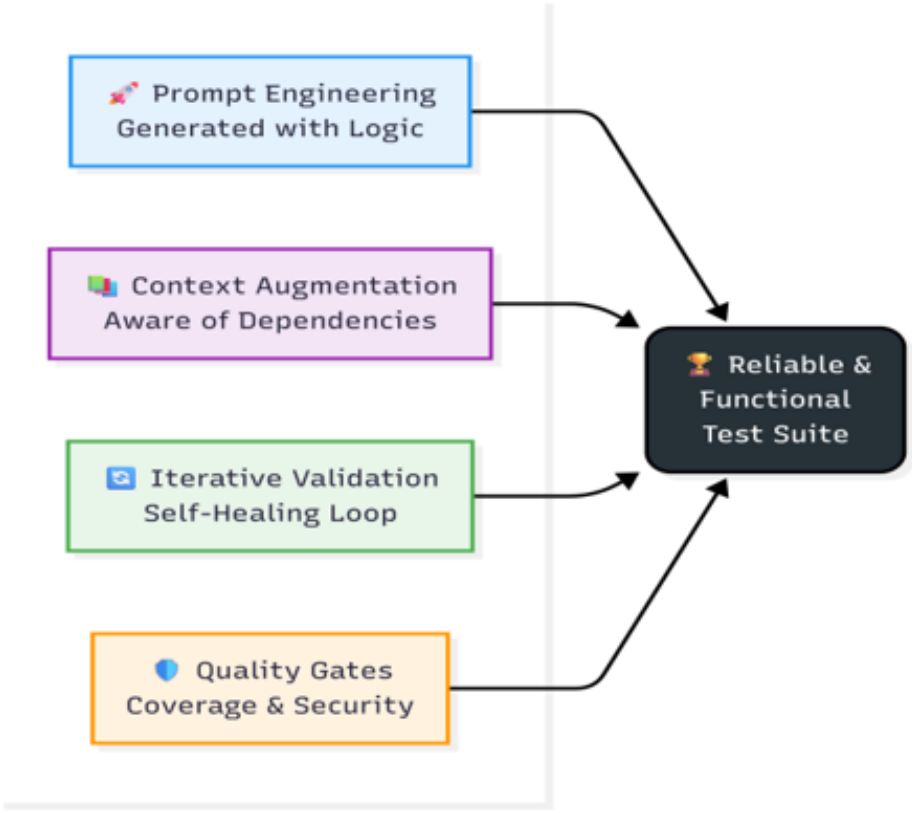


Prompt Engineering (89% impact on quality): Structured instructions that guide LLM generation

Context Augmentation: Supplementing LLM knowledge with specific codebase information

Iterative Validation: Compile-execute-feedback loops that ensure functional tests

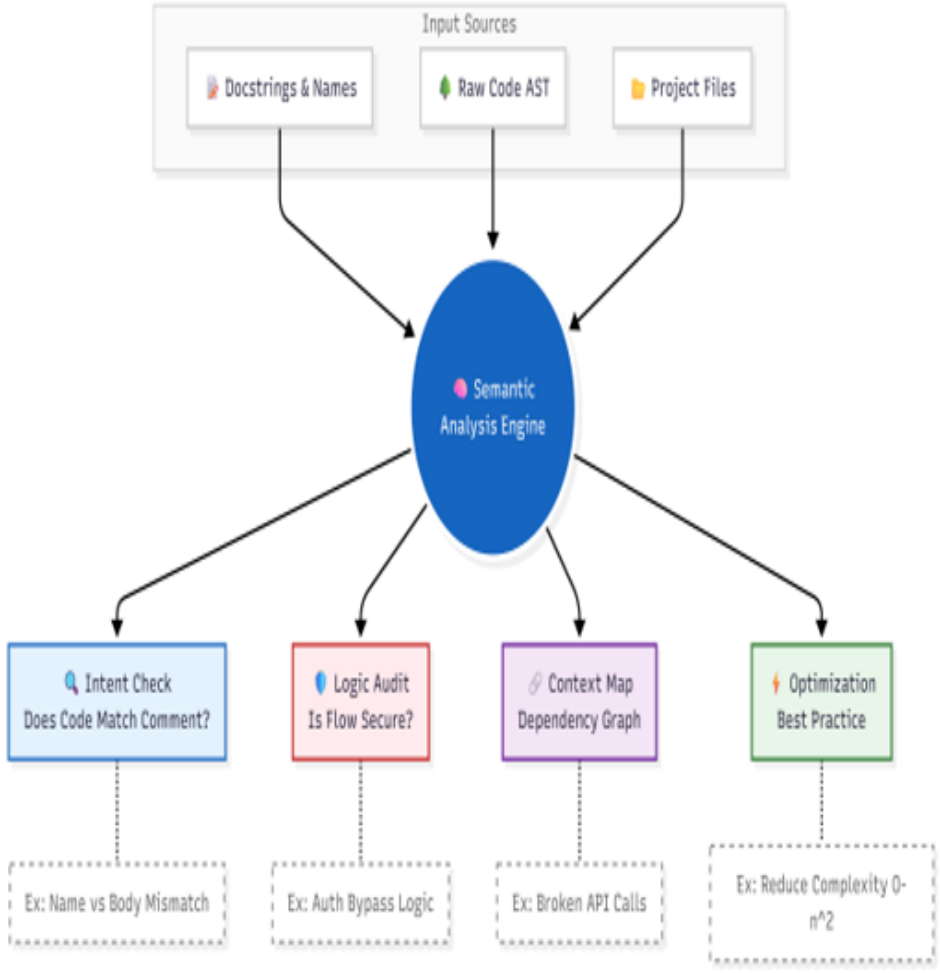
Quality Gates: Automated assessment of generated test coverage and effectiveness



Semantic Understanding in Code Inspection
Moving Beyond Syntax: Analyzing Logic, Intent, and Context

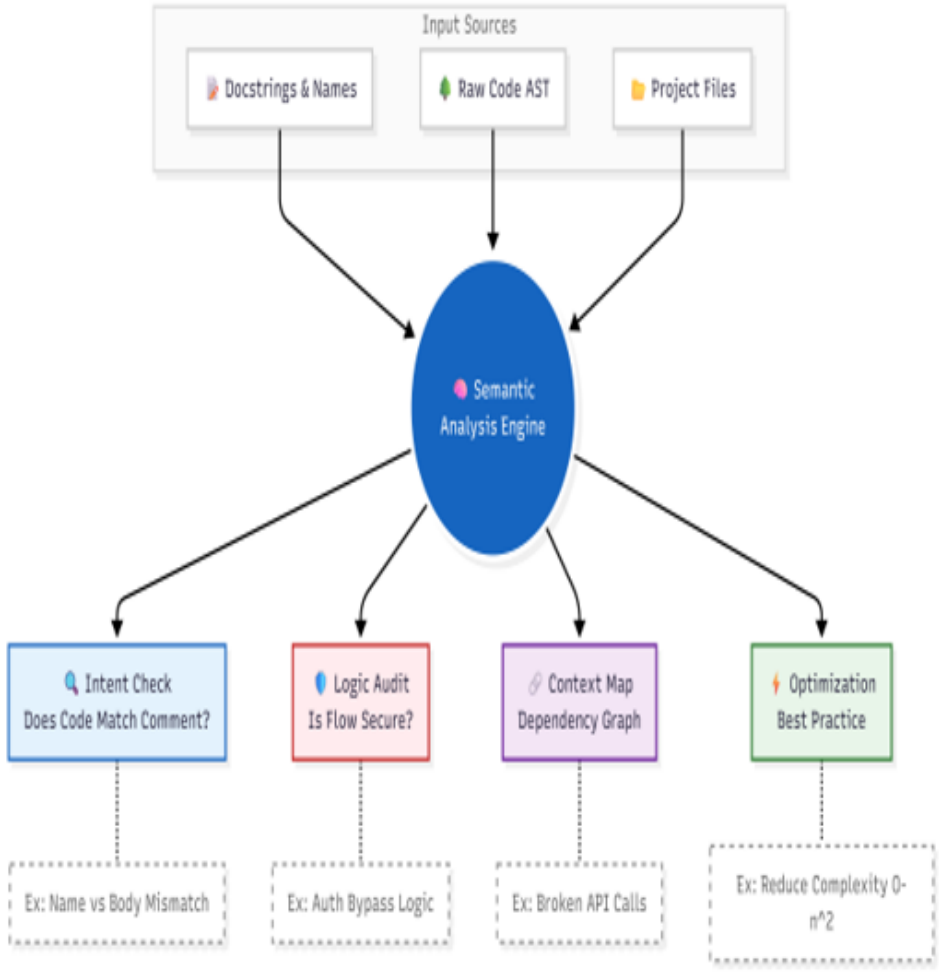
1. **Intent-Implementation Alignment**
   Concept: The LLM analyzes function names, docstrings, and comments to determine the developer's intent, then verifies if the actual code logic matches it.

   Capability: Detects semantic contradictions (e.g., a function named calculate_discount that contains no subtraction logic).



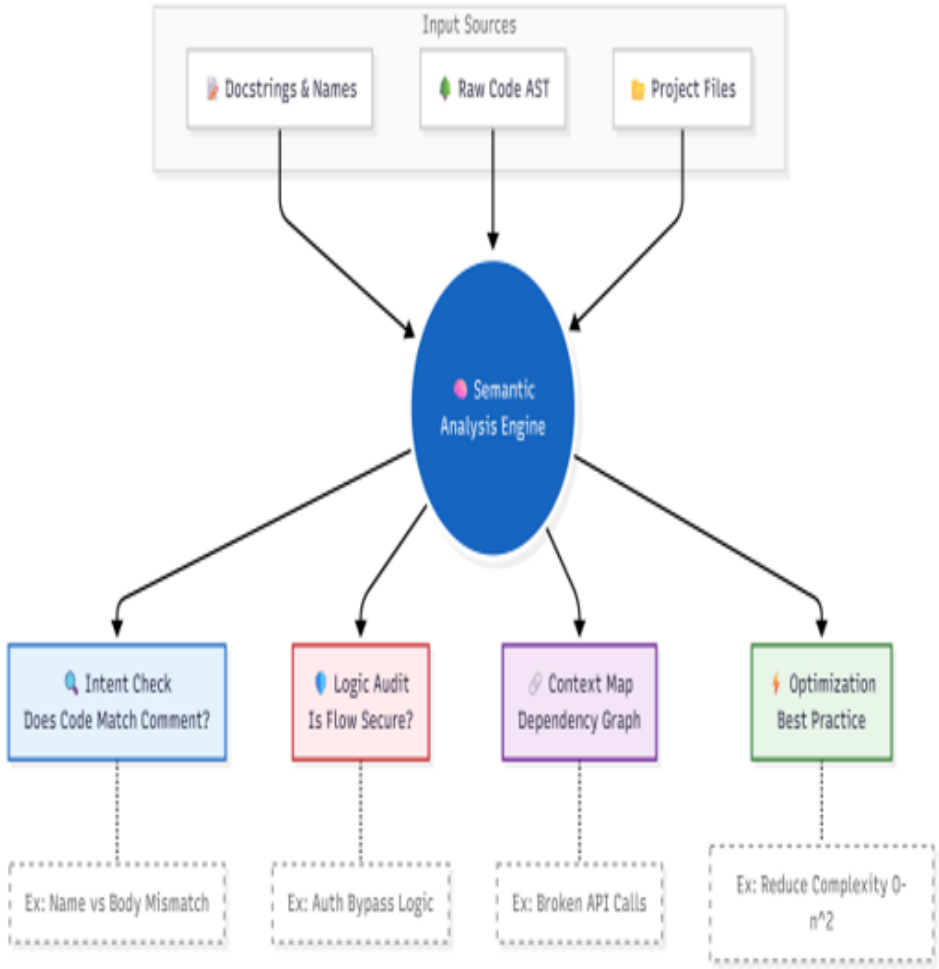
2. Deep Logic & Vulnerability Detection
   Concept: Identifies complex bugs that compilers and standard linters miss. It reasons through execution paths like a human auditor.

   Capability: Spots race conditions, infinite loops in edge cases, and business logic flaws (e.g., "If user_id is null, the auth check is skipped").



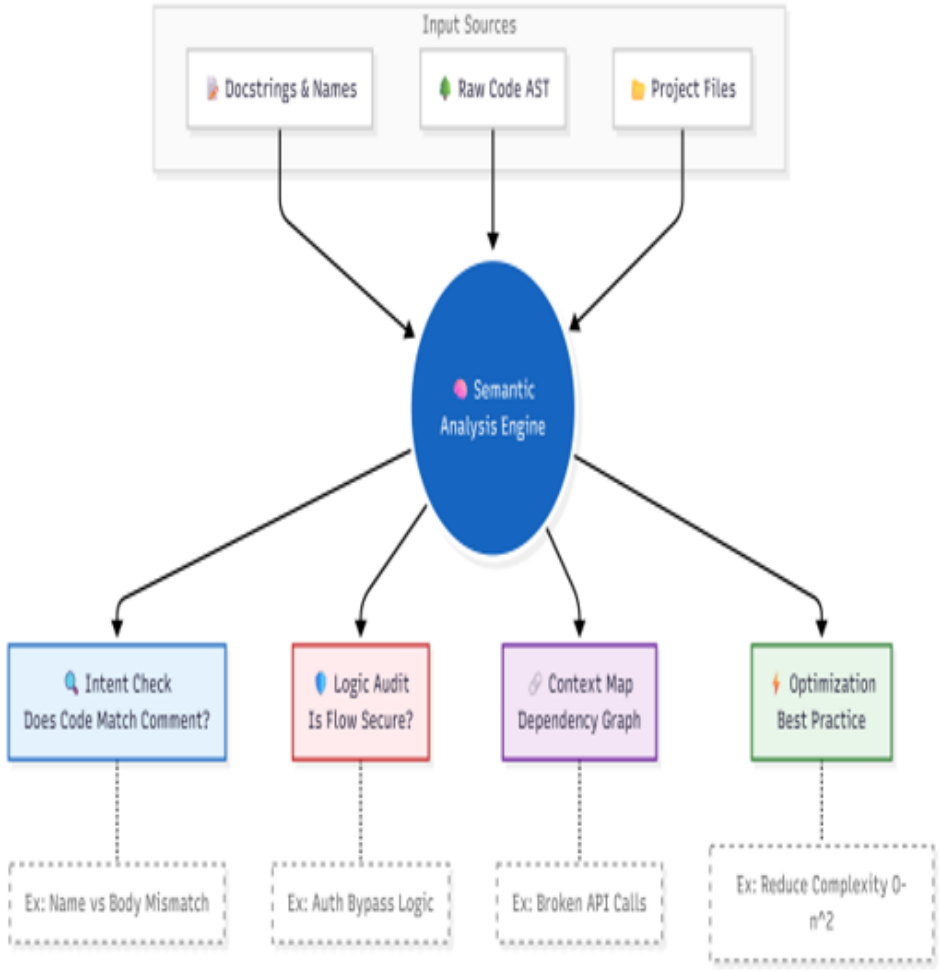
3. Cross-Module Context Awareness
   Concept: Leverages RAG (Retrieval-Augmented Generation) to understand dependency graphs across the entire repository.

   Capability: Detects breaking changes across files (e.g., An API signature change in File A breaking a caller in File B).



4. Intelligent Refactoring & Optimization
   Concept: Goes beyond bug fixing to improve code maintainability and performance.

   Capability: Suggests algorithmic improvements (e.g., reducing complexity from $O(n^2)$ to $O(n)$.



Metric	Traditional Methods (Search-Based / Symbolic)	LLM-Based Generation (GPT-4 / CodeLlama)	Analysis
Code Coverage	High (90%+)	Medium-Low	Brute-force efficiency: Traditional tools excel at exhaustive path exploration.
Bug Detection	High (Stronger)	Low (Weaker)	LLM struggles with deep corner.
Mutation Score	Medium	High (Superior)	LLMs write stricter assertions based on logic, not just syntax.
Readability	Low (Machine Code)	High (Human-like)	Maintainability: LLM output is human-readable; traditional output is often "spaghetti code."
Realism	Low (Random Data)	High (Domain-Aware)	Domain Context: LLMs generate realistic business data (e.g., valid JSONs) vs. random noise.

Catalog

Defects4J

Defects4J is a collection of reproducible bugs and a supporting infrastructure with the goal of advancing software engineering research.

The bugs:

Each bug has the following properties:

- Issue filed in the corresponding issue tracker, and issue tracker identifier mentioned in the fixing commit message.
- Fixed in a single commit.
- Minimized: the Defects4J maintainers manually pruned out irrelevant changes in the commit (e.g., refactorings or feature additions).
- Fixed by modifying the source code (as opposed to configuration files, documentation, or test files).
- A triggering test exists that failed before the fix and passes after the fix – the test failure is not random or dependent on test execution order.

1. Comparison Experiments of Traditional Methods and LLM-based Methods:
   - Dataset: Defects4J
   - Traditional Methods: Evosuite, Randoop
   - LLM-based Methods: Qwen3, Deepseek3.2
2. Ablation Experiment:
   - Engineering ablation prompts: Zero-sample, Few-sample, Thinking chain
   - Model Selection: Qwen3, Deepseek3.2
   - Experiment Method: Direct input of specific prompts and interaction with LLMs' API.
3. Metrics:
   - Bug detection rate
   - Test generation efficiency
   - Code coverage
   - False positives
   - Test readability
   - ...