TOGBench: A Developer-Written Multi-Variant Dataset and Benchmark Suite for Test Oracle Generation

Tasfia Tasnim
tasfia.tasnim@utdallas.edu
The University of Texas at Dallas
USA

Matthew B. Dwyer
matthewbdwyer@virginia.edu
University of Virginia
USA

Soneya Binta Hossain
sbhossain@utdallas.edu
The University of Texas at Dallas
USA

Abstract

Test oracles—the checks that decide whether a program execution is correct, and therefore whether a test should pass or fail—are central to the value of unit tests. In Java test suites, such oracles are commonly encoded as *assertions* or *expected-exception* checks, yet constructing them remains a costly manual step. Automated test oracle generation (TOG) aims to reduce this effort, and recent LLM-based approaches have made rapid progress. However, evaluation has not kept pace: existing benchmarks often rely on automatically generated tests, reduce oracle generation to single-assert prediction, simplify developer-written tests, or cover only a narrow range of oracle forms. As a result, they provide limited evidence about how TOG techniques perform on the richer and more heterogeneous oracles found in real test suites.

We introduce $OE25_{dev}$, a developer-written, multi-variant dataset for TOG, and TOGBench, an end-to-end benchmark suite for evaluating generated oracles in runnable Java systems. $OE25_{dev}$ is curated from unit tests across 25 open-source Java systems spanning 56 modules and preserves six oracle categories across realistic settings, including single-oracle, multi-oracle, mixed assertion-and-exception, and developer-authored custom-oracle tests. Rather than treating oracle generation as a static text-prediction task, TOGBench reintegrates generated oracles into executable test suites and evaluates them through compilation, execution, false-positive analysis, and mutation testing. Our evaluation shows that $OE25_{dev}$ preserves substantially greater structural complexity than prior TOG benchmarks and exposes a marked performance drop for representative TOG models on developer-written tests, particularly for assertion oracles. These results position TOGBench as a more realistic foundation for measuring and advancing automated test oracle generation.

CCS Concepts

• **Software and its engineering → Software testing and debugging**; *Empirical software validation.*

Keywords

test oracle generation, software testing benchmarks, developer-written unit tests

ACM Reference Format:
Tasfia Tasnim, Matthew B. Dwyer, and Soneya Binta Hossain. 2026. TOGBench: A Developer-Written Multi-Variant Dataset and Benchmark Suite for Test Oracle Generation. In *Proceedings of the 3rd ACM International Conference on AI-Powered Software (AIware '26), July 06–07, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3805760.3814927

1 Introduction

A test oracle is the mechanism that decides whether a program's observed behavior is correct for a given test execution—that is, whether the test passes or fails. More generally, an oracle can be viewed as a possibly partial decision procedure that maps an execution's stimuli and observations to an accept/reject outcome [1]. The long-recognized *oracle problem* is that such a decision procedure is often unavailable, incomplete, or too costly to apply in practice [4, 24]. In unit testing, oracles are most often encoded as assertion checks and expected-exception specifications embedded in test code. Figure 1 shows two representative examples.

Method Under Test (MUT)

```
int divide(int a, int b) {
  if (b == 0) throw new IllegalArgumentException("divisor cannot be zero");
  return a / b;
}
```

(a) Assertion Oracle

```
@Test
void testDivide() {
  int result = divide(10, 2);
  assertEquals(5, result); // oracle
}
```

(b) Exception Oracle

```
@Test
void testDivideByZero() {
  assertThrows(
    IllegalArgumentException.class,
    () -> divide(10, 0)); // oracle
}
```

Figure 1: Example assertion and exception oracles for the `divide` method. (a) The assertion oracle in `testDivide` checks that `divide(10, 2)` returns the expected value `5` using `assertEquals`. (b) The exception oracle in `testDivideByZero` checks that calling `divide(10, 0)` raises an `IllegalArgumentException` using `assertThrows`.

Test oracle generation (TOG) has become increasingly important as learning-based tools, including large language models (LLMs), are used to assist test development. Yet generating realistic oracles remains challenging: a generated oracle must be syntactically valid in its surrounding test context, semantically aligned with the intended behavior, and strong enough to detect faults without introducing false positives [9].

Existing datasets and benchmarks have enabled rapid progress, but many rely on simplifying assumptions that limit their relevance to *in-the-wild* developer-written tests. Some are derived from automatically generated tests with repetitive patterns; others formulate TOG as predicting a single oracle type—usually either an assertion or an expected exception [5, 10], and rely on static similarity metrics rather than end-to-end executability and fault-detection strength. In practice, developer-written tests exhibit much richer oracle behavior, including the six oracle categories shown in Table 1, mixed assertion-and-exception logic, and developer-authored custom checks. Consequently, prior benchmarks under-represent the diversity, structure, and project-specific conventions of real test suites. Many are also purely static datasets, offering no support for

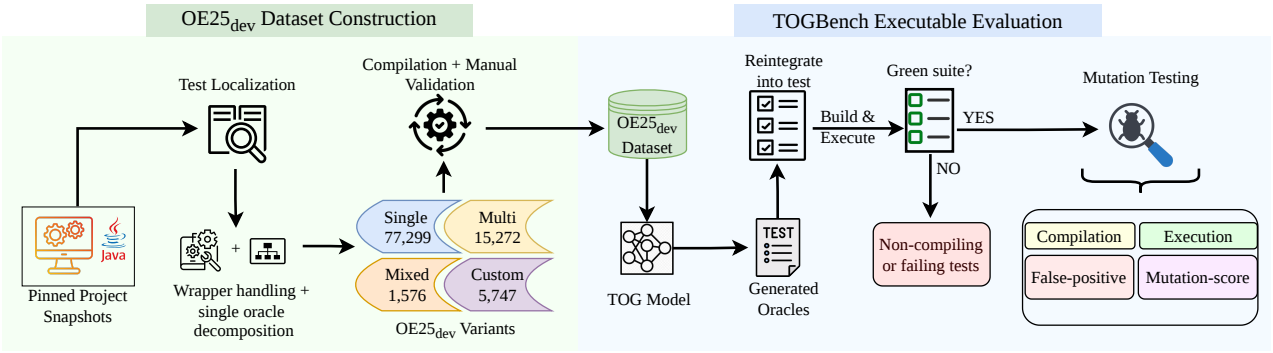

Figure 2: Overview of the end-to-end TOGBench pipeline, including OE25$_{dev}$ dataset construction and oracle evaluation.

reintegrating generated oracles into runnable test suites or evaluating them through fault-detection analysis, and several remain limited in scale [6, 11, 23].

To address these limitations, we present TOGBench, a multi-variant benchmark suite built around OE25$_{dev}$, a developer-written dataset curated from 25 open-source Java projects spanning 56 modules. A central design choice in TOGBench is to avoid reducing TOG to a single simplified prediction task. Instead, it supports four complementary configurations—OE25$_{dev}$-Single, OE25$_{dev}$-Multi, OE25$_{dev}$-Mixed, and OE25$_{dev}$-Custom—that allow researchers to study oracle generation under isolated single-oracle prediction, original tests with multiple oracles, mixed assertion-and-exception tests, and developer-defined custom assertion wrappers. Within OE25$_{dev}$-Single, OE25$_{dev}$ preserves six distinct oracle categories, as shown in Table 1, capturing a richer range of developer-written oracle behavior than benchmarks that collapse the problem into a narrow assertion-versus-exception split.

Beyond the dataset itself, TOGBench provides a reusable end-to-end toolchain for model training and benchmarking. The released instances and reference oracle blocks support supervised training and held-out inference; the tool suite then reintegrates generated oracles into executable tests, rebuilds the target project, executes the resulting suites, and evaluates outcomes through compilation, execution, false-positive analysis, and mutation testing. The current release, summarized in Table 2, contains 77,299 curated single-oracle instances, 15,272 multi-oracle tests, 1,576 mixed-oracle tests, and 5,747 custom assertion tests.

In summary, this paper makes the following contributions:

- **Four dataset variants** of developer-written tests, covering single-oracle, multi-oracle, mixed-oracle, and custom-assertion settings.
- **A six-category oracle taxonomy** that includes four oracle types absent from prior TOG datasets, better reflecting how developers write tests in practice.
- **The most structurally diverse TOG dataset to date**, combining six oracle categories across four dataset variants over 25 open-source Java systems.
- **An end-to-end evaluation harness** that reintegrates generated oracles into runnable test suites and evaluates them via compilation, execution, false-positive analysis, and mutation testing.

We study the following two research questions to evaluate our dataset and benchmark:

RQ1 (Real-world complexity). To what extent does OE25$_{dev}$ reflect the complexity of real-world developer-written tests, and how does it differ from existing oracle datasets?

RQ2 (End-to-end model performance). How do representative TOG models perform on TOGBench for assertion and exception oracle generation under executable end-to-end evaluation?

2 Background and Related Work

We review recent LLM-based test oracle generation, simplifying assumptions in existing benchmarks, and evaluation limitations that motivate TOGBench.

2.1 LLM-Based Test Oracle Generation

Recent LLM-based TOG methods can be grouped into three broad families: context-conditioned completion from a test prefix and focal-method context [5, 10, 23], tool-assisted prompting with iterative repair using static-analysis or execution feedback [3, 8, 19], and documentation-augmented inference using Javadoc and retrieval [7, 12, 14]. Across these families, generated oracles are typically limited to standard assertions and expected exceptions, and most methods reduce TOG to predicting a single masked oracle from curated context rather than reasoning over the full structure of a developer-written test.

2.2 Simplified Oracle Formulations and Developer-Written Diversity

Many prior datasets cast TOG as single oracle prediction and completion, retain only single-oracle tests, impose context-length limits, or rely on automatically generated tests with regular templates [5, 6, 10, 12, 22, 23]. In practice, developer-written oracle logic is often non-local, spanning try-catch regions, branching logic, helper methods, and framework-specific APIs; Table 1 summarizes the six categories preserved by OE25$_{dev}$. Benchmarks that reduce such tests to a single oracle therefore understate both the diversity and dependencies that realistic TOG must handle.

2.3 Evaluation Limitations and Positioning of TOGBench

Evaluation remains a central limitation in prior TOG work. Exact match and related text-similarity measures are weak proxies for oracle correctness and adequacy [18]. Stronger criteria, such as compilation success, execution behavior, false-positive rate, and bug detection strength, require reintegrating generated oracles into runnable tests and explicitly handling build and test failures [8, 10, 15]. Yet many existing datasets remain static, cover a narrow range of oracle forms, or are disconnected from the full project context needed for standardized end-to-end comparison [6, 22, 23]. To address this gap, TOGBench targets developer-written tests, preserves six oracle categories across single-, multi-, mixed-, and custom-oracle settings, and provides a reusable harness for oracle reintegration, compilation, execution-based validation, false-positive analysis, and mutation-based adequacy assessment. Together, these features provide a more realistic and reproducible basis for comparing TOG methods.

3 TOGBench and the OE25$_{dev}$ Dataset

TOGBench is an end-to-end benchmark suite for Java test oracle generation. It comprises three components: pinned snapshots of 25 open-source Java systems spanning 56 Maven modules; OE25$_{dev}$, a dataset of developer-written oracle-generation instances mined from those snapshots; and an executable harness for reintegrating generated oracles and evaluating them through compilation, execution, and mutation testing. Throughout the paper, TOGBench refers to the full suite, whereas OE25$_{dev}$ refers only to the released instance dataset.

3.1 Subject Systems and Reproducibility Requirements

We construct OE25$_{dev}$ from the 25 Java systems from [10, 11]. The systems cover core utilities, collections, configuration, bytecode tooling, networking, parsing, and web frameworks, representing diverse developer communities. Several of these systems have also been used in prior test-generation research [10, 11, 13, 17, 25].

Because TOGBench is executable, each subject snapshot must satisfy the following reproducibility requirements:

- Maven-based build system with a deterministic, reproducible build under JDK 8.
- Compatibility with PIT [2] for mutation analysis.
- Stable, non-flaky test execution under our controlled execution environment.
- A pinned commit or tag serving as the benchmark snapshot.

The full list of 25 Java systems, along with per-system statistics such as modules, SLOC, test framework, and snapshot commit, is available in our released artifact [20, 21].

3.2 Dataset Instances

An *oracle block* is a developer-written statement or structured code region that encodes the expected behavior checked by a test. Each single-oracle OE25$_{dev}$ instance masks one such block and records the context needed to reconstruct and execute the test:

$$I = \langle id,\ test\_prefix,\ mut,\ doc,\ meta \rangle.$$

Here, `id` uniquely identifies the instance; `test_prefix` is the masked test context with the target oracle replaced by `<place_holder>`; `mut` is the focal method under test; `doc` is its Javadoc when available; and `meta` stores reconstruction and build information. The removed block is retained as the reference oracle for supervised fine-tuning and executable evaluation.

3.3 Oracle Taxonomy

Developer-written tests express oracle intent through a broader set of structural patterns than the assertion-only or must-throw settings commonly used in prior work. OE25$_{dev}$ preserves six developer-written oracle categories, summarized in Table 1 and illustrated in Listing 1. Distinguishing these forms matters because each yields a different prefix structure, target oracle representation, and reintegration behavior. Collapsing them into a single exception label would obscure meaningful differences in both training and evaluation.

Table 1: Developer-written oracle categories in OE25$_{dev}$.

Oracle Type	Description
ASSERTION_ONLY	Standard assertion over output state
MUST_THROW	Exception must be thrown
MUST_NOT_THROW	No exception should be thrown
IF_THROWN_ASSERT	Exception properties are checked only if an exception is thrown
MUST_THROW_WITH_PROPERTIES	Exception must be thrown with expected properties
FAIL_ONLY	No valid execution should reach `fail()`

3.4 OE25$_{dev}$ Variants

Let t be a developer-written test method with ordered oracle blocks $O(t) = \langle o_1, \ldots, o_k \rangle$. We release four complementary variants:

- **OE25$_{dev}$-Single.** One instance per oracle block. For each target oracle o_i, we preserve the original test prefix up to o_i, replace o_i with `<place_holder>`, remove any non-target oracle blocks in the preserved prefix, and truncate the method at the oracle boundary.
- **OE25$_{dev}$-Multi.** Original tests containing at least two standard oracle blocks of the same family, i.e., multiple assertions or multiple exception checks, released at test-method granularity without single-oracle decomposition.
- **OE25$_{dev}$-Mixed.** Original tests containing at least one assertion oracle and at least one exception oracle, released at test-method granularity without single-oracle decomposition.
- **OE25$_{dev}$-Custom.** Invocations of project-defined assertion wrappers or helper checks, recorded as custom-oracle instances. When the wrapper definition is resolvable, we also inline it to expose internal standard oracles for OE25$_{dev}$-Single.

For the Multi, Mixed, and Custom variants, the `mut` field records the focal method or methods associated with the oracle blocks in oracle order.

Listing 1: Six oracle types found in developer-written tests in OE25$_{dev}$.

```
// (1) ASSERTION_ONLY: standard value/state check, no
     exception intent
assertEquals(2, x.size());

// (2) MUST_THROW: fail() enforces that an exception
     must be raised
try {
    EnumUtils.isValidEnum(null, "PURPLE");
    fail("NullPointerException");
} catch (NullPointerException e) { }

// (3) MUST_THROW_WITH_PROPERTIES: exception required +
     properties checked
try {
    Foo.parse("x");
    fail("Expected IllegalArgumentException");
} catch (IllegalArgumentException e) {
    assertTrue(e.getMessage().contains("x"));
}

// (4) MUST_NOT_THROW: catch exists only to fail if
     anything is thrown
try {
    method = MethodUtils.getAccessibleMethod(
        PublicSubBean.class, "setFoo", String.class);
} catch (final Throwable t) {
    fail("getAccessibleMethod() threw " + t);
}

// (5) IF_THROWN_ASSERT: properties checked only if
     exception occurs
try {
    Objects.requireNonNull(null);
} catch (final NullPointerException e) {
    assertEquals("The value must not be null.",
         e.getMessage());
}

// (6) FAIL_ONLY: fail() guards an unreachable branch,
     no try-catch
for (final Method method : methods) {
    if (methodName.equals("<init>")) {
    } else {
        fail("unexpected method " + method.getName());
    }
}
```

3.5 Construction Pipeline

Algorithm 1 outlines how each subject project is converted into the four OE25$_{dev}$ variants. The pipeline discovers test methods, localizes oracle blocks and custom assertion wrappers, derives variant-specific instances, and records the metadata needed to reconstruct and execute each rewritten test.

Test discovery and oracle detection. The pipeline discovers test under `src/test/java` using framework-specific rules for JUnit 3/4/5 and TestNG. It then localizes standard assertions, exception-checking idioms such as `assertThrows`, `assertDoesNotThrow`, and `try–catch+fail`, and project-defined custom assertion wrappers.

Focal-method and documentation mapping. For each single-oracle instance, the pipeline identifies the focal method under test and attaches Javadoc when available. For assertion oracles, the focal method is the nearest production call whose result or state is checked; for exception oracles, it is the production call inside the guarded `try` block or `assertThrows` body.

Algorithm 1 Pipeline for constructing OE25$_{dev}$ variants.

Require: Project P
Ensure: dataset $\mathcal{D}_{\text{Single}}, \mathcal{D}_{\text{Multi}}, \mathcal{D}_{\text{Mixed}}, \mathcal{D}_{\text{Custom}}$, metadata $\mathcal{M}_{\text{Single}}, \mathcal{M}_{\text{Multi}}, \mathcal{M}_{\text{Mixed}}, \mathcal{M}_{\text{Custom}}$

Initialize $\mathcal{D}_{\text{Single}}, \mathcal{D}_{\text{Multi}}, \mathcal{D}_{\text{Mixed}}, \mathcal{D}_{\text{Custom}} \leftarrow \emptyset$
Initialize $\mathcal{M}_{\text{Single}}, \mathcal{M}_{\text{Multi}}, \mathcal{M}_{\text{Mixed}}, \mathcal{M}_{\text{Custom}} \leftarrow \emptyset$
for all $t \in$ TestMethods **do**
  Custom $\leftarrow$ ExtractCustomBlocks(t)
  $t_{\text{inl}} \leftarrow$ InlineCustomBlocks(t, Custom)
  Assert $\leftarrow$ ExtractAssertionBlocks(t_{inl})
  Except $\leftarrow$ ExtractExceptionBlocks(t_{inl})
  Std $\leftarrow$ MergeBySourceOrder(Assert, Except)
  Meta $\leftarrow$ CollectMetadata(P, t, t_{inl}, Custom, Std, DocMap)
  for all $c \in$ Custom **do**
    Add($\mathcal{D}_{\text{Custom}}, \langle t, c \rangle$, Meta, $\mathcal{M}_{\text{Custom}}$)
  end for
  if $|\text{Assert}| \geq 1 \wedge |\text{Except}| \geq 1$ **then**
    AddUnique($\mathcal{D}_{\text{Mixed}}, t$, Meta, $\mathcal{M}_{\text{Mixed}}$)
  end if
  if $|\text{Assert}| \geq 2 \vee |\text{Except}| \geq 2$ **then**
    AddUnique($\mathcal{D}_{\text{Multi}}, t$, Meta, $\mathcal{M}_{\text{Multi}}$)
  end if
  for all $t' \in$ SingleInstances(Std) **do**
    Add($\mathcal{D}_{\text{Single}}, t'$, Meta, $\mathcal{M}_{\text{Single}}$)
  end for
end for
$(\mathcal{D}_{\text{Single}}, \mathcal{D}_{\text{Multi}}, \mathcal{D}_{\text{Mixed}}, \mathcal{D}_{\text{Custom}}) \leftarrow$ RepairAndValidate($\mathcal{D}_{\text{Single}}, \mathcal{D}_{\text{Multi}}, \mathcal{D}_{\text{Mixed}}, \mathcal{D}_{\text{Custom}}$)

Repair and validation. Oracle decomposition and rewriting can introduce minor syntactic issues, so the pipeline applies targeted repairs and rebuilds the rewritten tests in cloned test classes. Only instances that compile and execute consistently under the harness are retained; validated green suites are then eligible for mutation analysis with PIT.

3.6 Construction Challenges

Constructing OE25$_{dev}$ required a pipeline capable of detecting oracle blocks, decomposing multi-oracle tests, and rewriting them into compilable single-oracle instances without disrupting the surrounding test logic. The main challenges were:

- resolving and inlining project-specific custom assertions while preserving scope, types, and identifier bindings;
- identifying the true oracle when assertions are embedded in helper calls or intertwined with auxiliary test logic;
- isolating a single oracle from a multi-oracle test while preserving the required setup, control flow, and execution order;
- preserving annotations, `throws` clauses, inheritance relationships, nested classes, and package-private access;
- supporting JUnit 3/4/5, TestNG, and legacy test-discovery patterns within a unified pipeline, despite their different conventions for declaring and executing tests; and
- filtering flaky or environment-dependent tests, such as those involving network or GUI components, while maintaining reproducibility.

3.7 Executable Evaluation Harness

Given an instance I, a TOG method generates an oracle block $\widehat{o}$ to replace `<place_holder>`. Using the metadata associated with each OE25$_{dev}$ instance, the evaluation harness reinserts $\widehat{o}$ into the corresponding test method, rebuilds the system under test, and executes the test suite.

Some generated oracle blocks may introduce compilation errors or cause tests to fail on the original, correct implementation, indicating false-positive oracles. Such invalid outputs are discarded. The remaining oracle blocks compile successfully and preserve the original green test suite, making them suitable for the downstream bug-detection study using PIT mutation testing.

4 Dataset Characterization and Benchmark Evaluation

This section first characterizes the artifacts introduced in this work and then answers two research questions.

First, we discuss OE25$_{dev}$, the curated collection of developer-written tests and oracle instances, together with TOGBench, the executable evaluation protocol built on top of that collection. This characterization establishes the scale, oracle coverage, framework support, and evaluation capabilities of our artifacts. We then study two research questions: RQ1 examines whether OE25$_{dev}$ preserves the structural complexity of real-world developer-written tests, and RQ2 evaluates whether TOGBench enables end-to-end assessment of oracle-generation models.

4.1 Artifact Characterization

A test-oracle-generation resource has two distinct components: a data component and an evaluation component. The data component should provide enough scale and oracle diversity to represent realistic developer-written tests. The evaluation component should allow generated oracles to be reintegrated, compiled, executed, and assessed within their original project contexts. We therefore characterize OE25$_{dev}$ and TOGBench separately: OE25$_{dev}$ captures the curated test-oracle data, while TOGBench captures the executable evaluation protocol constructed from that data.

We characterize OE25$_{dev}$ using two dimensions. First, Table 2 reports dataset scale, including the number of projects, modules, original test methods, curated single-oracle instances, multi-oracle test methods, mixed-oracle test methods, and custom-assertion test methods. Second, Table 3 compares the data and evaluation properties of our artifacts with prior oracle-generation resources. The comparison separates dataset-level properties, such as oracle granularity and oracle-type coverage, from benchmark-level properties, such as framework support, executable reintegration, and mutation-based or bug-detection assessment.

Table 2 summarizes the scale of OE25$_{dev}$. Across 25 systems and 56 modules, OE25$_{dev}$ contains 100,118 original test methods and 77,299 curated single-oracle instances derived from developer-written tests. The original test suites also include 15,272 multi-oracle test methods and 1,576 mixed-oracle test methods. These counts show that developer-written tests frequently contain more than isolated single assertions. Instead, they often combine multiple checks within one test method, creating interdependent validation logic that is more structurally complex than single-oracle-only settings.

Table 2: Summary of OE25$_{dev}$.

Statistic	Value
Systems	25
Modules	56
SLOC	776,279
Original test methods	100,118
Curated single-oracle instances	77,299
ASSERTION_ONLY	73,023
MUST_THROW	2,899
MUST_NOT_THROW	694
IF_THROWN_ASSERT	257
MUST_THROW_WITH_PROPERTIES	319
FAIL_ONLY	107
Multi-oracle test methods	15,272
Mixed-oracle test methods	1,576
Custom-assertion test methods	5,747

The oracle distribution in OE25$_{dev}$ is intentionally imbalanced. ASSERTION_ONLY accounts for most curated single-oracle instances, with 73,023 examples, whereas FAIL_ONLY appears in only 107 instances. This imbalance reflects how developers write tests in practice: standard assertions are common, while fail-only or specialized exception-property checks occur in narrower situations. We preserve this natural distribution rather than artificially balancing the dataset, so that OE25$_{dev}$ represents the oracle patterns that test-oracle-generation models are likely to encounter in real systems.

Table 3 compares OE25$_{dev}$ and TOGBench with prior oracle-generation resources. At the dataset level, prior resources mainly focus on single-oracle instances and provide limited coverage of mixed oracle structures, custom assertion wrappers, and modern exception APIs. In contrast, OE25$_{dev}$ preserves both single- and multi-oracle tests, covers six oracle categories, and includes APIs such as `assertThrows` and `assertDoesNotThrow`. These properties increase the structural diversity of the oracle-generation task.

At the framework level, OE25$_{dev}$ includes tests written for JUnit 3, JUnit 4, JUnit 5, and TestNG. This broader coverage exposes models to a wider range of assertion APIs, exception-handling idioms, and project-specific testing conventions than prior resources limited to narrower framework settings.

At the benchmark-protocol level, TOGBench supports executable reintegration and mutation-based or bug-detection assessment. This distinction is important: OE25$_{dev}$ provides the oracle-generation instances, while TOGBench defines how generated oracles are evaluated. By reintegrating generated oracles into compilable project snapshots, TOGBench enables evaluation beyond textual similarity.

Although OE25 [11] also supports executable and mutation-based evaluation, its tests are generated by EvoSuite and are therefore shaped by generation templates. This leads to more uniform test structures, predictable assertion forms, and limited variation in project-specific validation logic. In contrast, OE25$_{dev}$ is derived

Table 3: Comparison of oracle-generation resources. Dataset properties describe the curated test data; benchmark properties describe framework coverage and executable-evaluation support.

Category	Property	ATLAS [23]	SF110 [6]	OE25 [11]	$OE25_{dev}$ / TOGBench [20]
Dataset	Oracle granularity	Single	Single	Single	Single + Multi
	Oracle types	Assertions	Assertions + Exceptions	Assertions + Exceptions	6 oracle categories
	Exception APIs, e.g., `assertThrows`	✗	✗	✗	✓
	Exception handling via try/fail/catch	✓	✓	✓	✓
	Mixed assertion–exception tests	✗	✗	✗	✓
	Custom assertion wrappers	✗	✗	✗	✓
	Test authorship	Developers	EvoSuite	EvoSuite	Developers
	Curation strategy	GitHub projects, < 1 k tokens, single assertion	SF110 projects from SourceForge	Java utility packages	Java utility packages
Framework	JUnit versions	4	3/4	4	3/4/5
	TestNG	✗	✗	✗	✓
	# unique assertion APIs	9	8	10	15
Protocol	Executable reintegration	✗	✗	✓	✓
	Mutation-based or bug-detection assessment	✗	✗	✓	✓

from developer-written tests across multiple frameworks and preserves custom assertion wrappers, mixed assertion–exception validation, and project-specific idioms. As a result, $OE25_{dev}$ provides a more heterogeneous data foundation, while TOGBench provides an executable protocol for evaluating generated oracles under realistic settings.

Overall, this characterization establishes that $OE25_{dev}$ is a large and structurally diverse collection of developer-written oracle instances, and that TOGBench provides the executable infrastructure needed to evaluate generated oracles behaviorally. The following research questions build on this characterization: RQ1 analyzes the structural complexity of the test methods in $OE25_{dev}$, and RQ2 evaluates oracle-generation models using the executable protocol provided by TOGBench.

4.2 RQ1: Real-World Test Complexity

RQ1 examines whether $OE25_{dev}$ preserves the structural complexity of developer-written tests and how this complexity compares with prior oracle-generation datasets. This question focuses on the data artifact: if a dataset simplifies test bodies, removes complex setup logic, or retains only isolated oracle statements, it may underestimate the reasoning required for realistic test-oracle generation. We therefore measure whether $OE25_{dev}$ contains the control flow, contextual scope, and path complexity that appear in real developer-written tests.

Setup: We compute method-level structural metrics using PMD [16], a static analysis tool that reports established Java metrics such as cyclomatic complexity, cognitive complexity, NCSS, and NPath complexity. We apply PMD to each test method and extract the resulting method-level measurements into a structured metrics table. For each metric, we report the standard deviation (σ), which captures variation across test methods, and the maximum value, which captures the most complex cases present in each dataset.

OE25 [11] serves as the primary executable baseline because its EvoSuite-generated tests are distributed as Maven projects. SF110 [6] and ATLAS [23] are not distributed as ready-to-build project snapshots with the metadata needed for project-level execution. For these datasets, we place each test method inside a minimal Java wrapper class and run PMD directly on the resulting source file. This allows us to compute the same method-level structural metrics for all datasets.

We compute the following metrics:

- *Cyclomatic Complexity (CC)* counts the number of linearly independent control-flow paths through a method.
- *Cognitive Complexity* estimates how difficult a method is to understand by penalizing nesting and non-linear control flow.
- *NCSS* counts non-commenting source statements and approximates the amount of executable test context surrounding the oracle.
- *NPath Complexity* counts the number of acyclic execution paths through a method.

Results: Table 4 reports structural metrics for SF110, ATLAS, OE25, and the three $OE25_{dev}$ variants. The prior datasets exhibit relatively low structural spread and low upper-bound complexity. SF110 has the lowest values across most metrics, with CC max = 3, cognitive complexity max = 2, and NPath max = 1. OE25 shows a similar pattern, with CC max = 2 and NPath max = 3, reflecting the regular structure of EvoSuite-generated tests. ATLAS contains developer-written tests, but its curation process favors short, single-assertion examples. As a result, ATLAS has a very low CC spread ($\sigma = 0.09$), although it includes an isolated high NPath value of 8,192.

Table 4: Structural comparison of oracle-generation datasets. Bold denotes the two largest values in each column.

	Control flow		*Cognitive load*		*Context size*		*Path complexity*	
Dataset	**CC (σ)**	**CC (max)**	**Cog. (σ)**	**Cog. (max)**	**NCSS (σ)**	**NCSS (max)**	**NPath (σ)**	**NPath (max)**
SF110 [6]	0.44	3	0.11	2	1.72	108	0.0	1
ATLAS [23]	0.09	7	0.98	13	3.87	68	25.35	8,192
OE25 [11]	0.29	2	0.14	4	7.21	123	0.30	3
OE25$_{dev}$ (Single)	0.83	18	2.53	45	8.21	**242**	9.38	960
OE25$_{dev}$ (Multiple)	**1.00**	**63**	**3.51**	**62**	**13.76**	**1,333**	$\mathbf{3.7 \times 10^{16}}$	$\mathbf{4.6 \times 10^{18}}$
OE25$_{dev}$ (Mixed)	**2.46**	**39**	**3.52**	**50**	**12.60**	146	$\mathbf{7.7 \times 10^{7}}$	$\mathbf{2.2 \times 10^{9}}$

Note: Cog. = Cognitive Complexity, σ = standard deviation, and max = maximum value.

Table 5: Per-project oracle-generation accuracy (%) under executable evaluation. Bold marks the lower value within each matched OE25–OE25$_{dev}$ pair for the same model, project, and oracle type.

Model	**Dataset**	**joda-time**		**commons-pool2**		**commons-beanutils**		**commons-collections4**		**commons-configuration2**	
		Except	**Assert**	**Except**	**Assert**	**Except**	**Assert**	**Except**	**Assert**	**Except**	**Assert**
TOGA [5]	OE25	16	17.3	**33**	12.4	**31**	22	**34**	35.2	29	25.8
	OE25$_{dev}$	**13.8**	**16**	46.1	**9.6**	37.4	**11.5**	71.1	**5.5**	**0**	**4.3**
TOGLL [10]	OE25	99	54	89	46.5	**91**	60.8	**99.2**	78.9	**96.4**	67.1
	OE25$_{dev}$	**94**	**17.8**	**83.3**	**25.6**	92.1	**21.2**	100	**25.8**	100	**10.7**
Doc2OracLL [12]	OE25	99.9	81.8	100	86.2	**99.5**	77.0	100	89.4	**90.1**	75.1
	OE25$_{dev}$	**91.4**	**49.3**	**70.8**	**25**	99.6	**15.3**	100	**33.6**	100	**31.9**

In contrast, OE25$_{dev}$ contains substantially more structurally complex tests. Even the Single variant reaches CC max = 18, cognitive complexity max = 45, and NCSS max = 242, exceeding the corresponding maxima of SF110, ATLAS, and OE25 on these metrics. This shows that even when OE25$_{dev}$ is decomposed into single-oracle instances, the surrounding developer-written test context remains more complex than the contexts represented in prior datasets.

The difference becomes more pronounced when multi-oracle and mixed-oracle tests are preserved. OE25$_{dev}$ (Multiple) reaches CC max = 63 and NCSS max = 1,333, indicating that multi-oracle developer tests often contain substantially larger setup and validation contexts. OE25$_{dev}$ (Mixed) has the largest spread in cyclomatic complexity (σ = 2.46) and cognitive complexity (σ = 3.52), suggesting that tests combining assertion and exception behavior vary more widely in structure than single-oracle tests.

Path complexity shows the largest gap. While ATLAS has an NPath maximum of 8,192, OE25$_{dev}$ (Multiple) reaches 4.6×10^{18} and OE25$_{dev}$ (Mixed) reaches 2.2×10^{9}. These values reflect the exponential growth in execution paths caused by branching, nesting, fixture setup, and multi-step validation logic in developer-written tests.

These results follow from the construction of each dataset. SF110 and OE25 rely on EvoSuite-generated tests, whose structures are shaped by generation templates and therefore tend to have limited branching and shallow nesting. ATLAS draws from developer-written tests, but its filtering process removes many long or structurally rich cases. In contrast, OE25$_{dev}$ preserves developer-written test structure, including project-specific setup code, multi-oracle validation, mixed assertion–exception logic, and custom validation idioms.

Finding (RQ1): OE25$_{dev}$ preserves substantially more structural complexity than SF110, ATLAS, and OE25. This difference is visible in control-flow complexity, cognitive complexity, executable context size, and path complexity, especially for the Multiple and Mixed variants. Prior datasets suppress this complexity either through automated test generation or through curation filters that favor short, single-oracle tests.

4.3 RQ2: End-to-End Benchmark Evaluation

RQ2 examines whether TOGBench enables end-to-end behavioral evaluation of oracle-generation models and how representative models perform on developer-written tests compared with EvoSuite-generated tests. This question focuses on the benchmark protocol. Textual similarity alone cannot determine whether a generated oracle compiles, executes correctly, or behaves as intended in its

original project context. A realistic benchmark must therefore reintegrate generated oracles into system test suites and evaluate them through compilation and execution.

Setup: We evaluate three representative oracle-generation models: TOGA [5], TOGLL [10], and Doc2OracLL [12]. We apply the executable evaluation protocol to two datasets: OE25, which contains EvoSuite-generated tests, and $OE25_{dev}$, which contains developer-written tests. Because these models support only single-oracle prediction, this experiment is restricted to assertion and exception oracle instances.

For each test instance, we replace the original oracle with the model-generated oracle and reintegrate the modified test into its original test class and project snapshot. We then compile and execute the project using Maven under the project's native build configuration. This procedure evaluates generated oracles in their executable context rather than as isolated text predictions.

We compute exception-oracle accuracy as follows:

$$\text{Exception}_{accuracy}(P) = \frac{T_{pred}}{T_e} \times 100(\%) \quad (1)$$

Here, T_e denotes the total number of exception-oracle instances in project P, and T_{pred} denotes the number of instances for which the model correctly predicts the exception oracle.

We compute assertion-oracle accuracy as follows:

$$\text{Assertion}_{accuracy}(P) = \frac{T_a - (T_{ce} + T_{fp} + T_{em})}{T_a} \times 100(\%) \quad (2)$$

Here, T_a denotes the total number of assertion-oracle instances in project P, T_{ce} denotes the number of generated tests that fail to compile, T_{fp} denotes the number of generated tests that produce false-positive behavior, and T_{em} denotes the number of empty oracle outputs. We apply this accuracy definition uniformly to all three models.

We report accuracy only for assertion and exception oracles because TOGA, TOGLL, and Doc2OracLL are designed for these oracle types. A per-category breakdown across the full six-category taxonomy of $OE25_{dev}$ would therefore not be meaningful for these tools. ATLAS [23] and SF110 [6] are excluded from this executable comparison because they are not distributed with the project metadata and build infrastructure required for full test reintegration.

Results: Table 5 reports per-project accuracy for exception and assertion oracle generation. The five projects are included because both OE25 and $OE25_{dev}$ contain enough assertion and exception oracle instances for meaningful comparison. The reported values reflect executable outcomes: generated oracles must be reintegrated, compiled, and executed successfully to count as correct.

Across all three models, assertion-oracle accuracy is consistently lower on $OE25_{dev}$ than on OE25. TOGLL achieves assertion accuracy between 46.5% and 78.9% on OE25, but only between 10.7% and 25.8% on $OE25_{dev}$. Averaged across the five projects, TOGLL's assertion accuracy drops from 61.5% on OE25 to 20.2% on $OE25_{dev}$, a decrease of 41.3 percentage points.

Doc2OracLL shows the same pattern. Its average assertion accuracy decreases from 81.9% on OE25 to 31.0% on $OE25_{dev}$, a drop of 50.9 percentage points. TOGA also performs worse on $OE25_{dev}$, with average assertion accuracy decreasing from 22.5% to 9.4%, a drop of 13.1 percentage points. Overall, assertion-oracle accuracy is lower on $OE25_{dev}$ in all 15 matched model–project comparisons.

Exception-oracle accuracy is more stable across the two datasets. TOGLL achieves average exception accuracy of 94.9% on OE25 and 93.9% on $OE25_{dev}$, a difference of only 1.0 percentage point. Doc2OracLL decreases from 97.9% on OE25 to 92.4% on $OE25_{dev}$. TOGA shows more variation across projects, but the overall contrast between OE25 and $OE25_{dev}$ is still less pronounced for exceptions than for assertions.

Across all oracle types, models perform worse on $OE25_{dev}$ in 21 of the 30 matched model–project–oracle-type comparisons. The degradation is concentrated in assertion generation, which requires models to reason over richer setup code, project-specific object states, and longer validation contexts. Exception generation appears less sensitive to this added complexity, likely because many exception oracles depend on more localized control-flow patterns.

Finding (RQ2). TOGBench enables end-to-end behavioral evaluation of oracle-generation models by reintegrating generated oracles into project test suites and assessing them through compilation and execution. Under this protocol, models that perform well on EvoSuite-generated tests degrade substantially on $OE25_{dev}$, especially for assertion oracles. This result shows that developer-written tests expose oracle-generation challenges that prior generated-test benchmarks do not fully capture.

5 Threats to Validity and Limitations

Our oracle definition is operational because oracles must be detected automatically. $OE25_{dev}$ therefore focuses on explicit Java assertion and exception-checking idioms, including JUnit/TestNG assertions, `assertThrows`, `try–catch+fail`, Java `assert`, and project-specific `assert*` helpers. This may miss implicit oracles encoded through framework behavior, side effects, or domain-specific helpers, biasing $OE25_{dev}$ toward syntactically explicit oracle statements.

The single-oracle variant is a controlled simplification, not a semantics-preserving refactoring. Disabling surrounding oracles and truncating after the target oracle can remove inter-oracle dependencies or post-oracle behavior. Custom assertion helpers may also have side effects. Thus, single-oracle instances should be interpreted as isolated checks under the original setup, not replacements for the full developer-written tests.

The construction pipeline may also introduce errors in oracle detection, classification, or rewriting. We mitigate this risk by requiring transformed tests to compile and execute and by auditing sampled transformations, but subtle semantics-changing errors may remain. Finally, TOGBench prioritizes reproducibility by including Maven projects that build reliably and excluding flaky, network-dependent, or configuration-sensitive tests. This improves repeatability but may underrepresent integration-heavy testing scenarios.

6 Conclusion

Realistic benchmarks for test-oracle generation remain limited. Existing datasets often rely on generated tests, filter out complex developer-written tests, or support only text-based evaluation. As

a result, they may underestimate the difficulty of producing oracles that compile, execute, and detect faults in real projects.

This work introduces $OE25_{dev}$, a curated collection of developer-written Java tests, and TOGBench, an executable evaluation protocol built around it. $OE25_{dev}$ covers 25 systems and 56 modules, with 77,299 runnable single-oracle instances, including 73,023 assertion-only and 4,276 exception- or fail-related instances. It also preserves richer structures: 1,576 mixed assertion–exception tests, 15,272 multi-oracle tests, and 5,747 custom-assertion instances.

TOGBench reintegrates generated oracles into project test suites and evaluates them through compilation, execution, false-positive detection, and mutation testing. Our results show that developer-written tests in $OE25_{dev}$ are structurally more complex than prior resources and that existing models degrade substantially on them, especially for assertion generation. We plan to expand $OE25_{dev}$ with more systems and extend the pipeline beyond Java, with the goal of making TOGBench a living testbed for executable and fault-revealing oracle generation.

7 Availability and Reproducibility

Data, scripts, and documentation are available in the figshare repository [20] and on GitHub [21].

References

[1] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. doi:10.1109/TSE.2014.2372785

[2] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java (demo) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 449–452. doi:10.1145/2931037.2948707

[3] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. 2024. Effective test generation using pre-trained Large Language Models and mutation testing. *Information and Software Technology* 171 (2024), 107468. doi:10.1016/j.infsof.2024.107468

[4] Martin D. Davis and Elaine J. Weyuker. 1981. Pseudo-oracles for non-testable programs. In *Proceedings of the ACM '81 Conference (ACM '81)*. Association for Computing Machinery, New York, NY, USA, 254–257. doi:10.1145/800175.809889

[5] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: A Neural Method for Test Oracle Generation. In *44th IEEE/ACM International Conference on Software Engineering (ICSE 2022)*. ACM, 2130–2141. doi:10.1145/3510003.3510141

[6] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 8:1–8:42. doi:10.1145/2685612

[7] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic Generation of Oracles for Exceptional Behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 213–224. doi:10.1145/2931037.2931061

[8] Ishrak Hayet, Adam Scott, and Marcelo d'Amorim. 2025. ChatAssert: LLM-Based Test Oracle Generation With External Tools Assistance. *IEEE Transactions on Software Engineering* 51, 1 (2025), 305–319. doi:10.1109/TSE.2024.3519159

[9] Soneya Binta Hossain. 2024. Ensuring Critical Properties of Test Oracles for Effective Bug Detection *(ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 176–180. doi:10.1145/3639478.3639791

[10] Soneya Binta Hossain and Matthew B. Dwyer. 2025. TOGLL: Correct and Strong Test Oracle Generation with LLMS. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1475–1487. doi:10.1109/ICSE55347.2025.00098

[11] Soneya Binta Hossain, Antonio Filieri, Matthew B. Dwyer, Sebastian Elbaum, and Willem Visser. 2023. Neural-Based Test Oracle Generation: A Large-Scale Evaluation and Lessons Learned. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 120–132. doi:10.1145/3611643.3616265

[12] Soneya Binta Hossain, Raygan Taylor, and Matthew Dwyer. 2025. Doc2OracLL: Investigating the Impact of Documentation on LLM-Based Test Oracle Generation. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE084 (June 2025), 22 pages. doi:10.1145/3729354

[13] 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* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. doi:10.1145/2610384.2628055

[14] 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)*. 279–289. doi:10.1109/ICST62969.2025.10988926

[15] Davide Molinelli, Luca Di Grazia, Alberto Martin-Lopez, Michael D. Ernst, and Mauro Pezzè. 2025. Do LLMs Generate Useful Test Oracles? An Empirical Study with an Unbiased Dataset. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Seoul, Korea, Republic of). IEEE Press, 278–290. doi:10.1109/ASE63991.2025.00031

[16] PMD Developers. 2024. PMD: An extensible cross-language static code analyzer. https://github.com/pmd/pmd.

[17] David Schuler and Andreas Zeller. 2011. Assessing Oracle Quality with Checked Coverage. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. 90–99. doi:10.1109/ICST.2011.32

[18] Jiho Shin, Hadi Hemmati, Moshi Wei, and Song Wang. 2024. Assessing Evaluation Metrics for Neural Test Oracle Generation. *IEEE Transactions on Software Engineering* 50, 9 (2024), 2337–2349. doi:10.1109/TSE.2024.3433463

[19] Weifeng Sun, Hongyan Li, Meng Yan, Yan Lei, and Hongyu Zhang. 2023. Revisiting and Improving Retrieval-Augmented Deep Assertion Generation. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, 1123–1135. doi:10.1109/ASE56229.2023.00090

[20] Tasfia Tasnim, Matthew B. Dwyer, and Soneya Binta Hossain. 2026. TOGBench artifact: AIware-2026. doi:10.6084/m9.figshare.31562056

[21] Tasfia Tasnim, Matthew B. Dwyer, and Soneya Binta Hossain. 2026. TOGBench artifact code. https://github.com/assert-lab/OE25-DEV. GitHub repository.

[22] Johannes Villmow, Jonas Depoix, and Adrian Ulges. 2021. ConTest: A Unit Test Completion Benchmark featuring Context. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, Royi Lachmy, Ziyu Yao, Greg Durrett, Milos Gligoric, Junyi Jessy Li, Ray Mooney, Graham Neubig, Yu Su, Huan Sun, and Reut Tsarfaty (Eds.). Association for Computational Linguistics, Online, 17–25. doi:10.18653/v1/2021.nlp4prog-1.2

[23] 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 42nd International Conference on Software Engineering (ICSE 2020)*. ACM, 507–519. doi:10.1145/3377811.3380429

[24] Elaine J. Weyuker. 1982. On Testing Non-Testable Programs. *Comput. J.* 25, 4 (11 1982), 465–470. doi:10.1093/comjnl/25.4.465

[25] Yucheng Zhang and Ali Mesbah. 2015. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 214–224. doi:10.1145/2786805.2786858