# Language-Generalised Test Oracle Generation Using Large Language Models

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)


| | |
|---|---|
| eingereicht von: | Liadan Anandarajah |
| geboren am: | 18.11.1999 |
| geboren in: | London, Großbritannien |
| Gutachter/innen: | Prof. Dr. Lars Grunske |
| | Prof. Dr. Yannic Noller |



eingereicht am Institut für Informatik der Humboldt-Universität zu Berlin am:
23.12.2025

# Abstract

The oracle problem remains a fundamental obstacle to scalable software testing: even when test inputs are available, determining outcome correctness often requires a behavioural specification that is unavailable in practice [2, 43]. Although LLM-based oracle generation has been explored, evaluation is largely confined to Java-centric benchmarks, despite evidence that LLM code generation performs most strongly for Python [4, 45]. Motivated by this discrepancy, this thesis investigates what a language-generalised LLM-driven oracle-generation pipeline can look like when instantiated for Python environments, while remaining comparable to established Java-based workflows. The proposed approach synthesises *executable* and *discriminative* test oracles from a stable prompt contract constructed from method source code, available documentation, and bug-context text. Generated oracles are rendered as Python implementations constrained by language-specific micro-skeletons, enabling consistent execution and analysis across settings. The pipeline is evaluated primarily in a Defects4J/EvoRepair configuration by executing Java methods via a JPype-based harness and assessing fault-discrimination behaviour, and is complemented by a smaller BugsInPy/Tests4Py case study in a native Python environment. The results indicate that LLM-driven oracle generation is feasible and often effective, particularly for exception- and guard-driven defect signatures. However, robustness is limited by cross-language runtime binding and by semantic under-specification for return-value and stateful predicates, motivating selective post-processing and targeted human intervention for the remaining hard cases.

# Contents

# 1 Introduction

The oracle problem in software testing represents one of the most persistent and complex challenges in the domain of software quality assurance. It refers to the difficulty, ambiguity, and often infeasibility of determining whether the output of a program under test is correct [2,17,49]. As originally articulated by Weyuker (1982), a program may be deemed non-testable if a test oracle does not exist, or if the cost of verifying the correctness of the outputs is prohibitively high [43]. The severity of the oracle problem has only increased in contemporary software systems, particularly in fields with complex algorithms, scientific computing, and machine learning, where outputs can be stochastic, approximate, or domain-specific [35].

The significance of the oracle problem lies in its impact on test automation. While the generation of test inputs has seen considerable advances and has been largely automated, validating the correctness of outputs remains a critical bottleneck [2]. Traditional testing methods presuppose the existence of a perfect oracle; an external mechanism capable of providing an authoritative judgment on the correctness of outputs [43]. However, in many real-world scenarios, such perfect oracles are unavailable or require extensive human expertise and intervention [21].

Various automated strategies have been proposed to alleviate this limitation. Metamorphic testing validates outputs based on known input-output relationships rather than explicit expectations [11]. Pseudo-oracles compare outputs from independent implementations, while specification-based testing relies on formal models to describe expected behaviours [31,38]. Regression testing leverages previous versions of a system as baselines for comparison [48]. Despite these advances, none of these techniques offer a universal solution, particularly when confronted with the complexities of modern, cross-language software ecosystems [2,27].

Prior empirical studies on large language model (LLM) based oracle generation remain relatively limited and commonly rely on long-public benchmarks, which may be present in model training corpora and thus threaten validity through data leakage. Recent work explicitly notes that "the few studies" on LLM oracle generation often use modest-sized public benchmarks and argues this can inflate performance estimates, motivating broader and more leakage-aware evaluation settings [22].

A further challenge is that code-LLM capabilities are not uniform across programming languages. Many code-generation evaluations have historically centred Python, which is heavily represented in training data and benchmark practice. MultiPL-E, for example, highlights that contemporary models are typically evaluated primarily on Python and that performance correlates with language popularity and related factors [4]. In addition, multilingual code benchmarks report substantial drops when models move beyond Python, indicating that cross-language generalisation is not guaranteed (e.g., models trained on Python achieving limited pass@1 performance in other languages) [4,45]. Accordingly, the contribution of this work is not only to evaluate whether LLMs can produce usable oracles, but also to assess whether the underlying oracle-generation approach transfers across languages under comparable experimental conditions. This bridges a gap between predominantly Java-centric oracle studies and the practical need for language-generalised

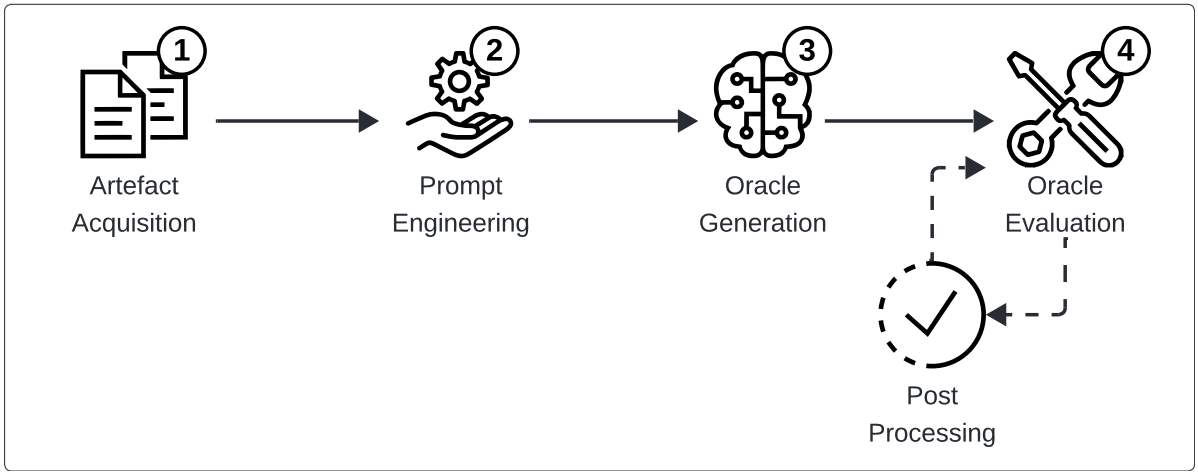


Figure 1: Simplified overview of the thesis workflow.

testing pipelines [28].

Building on this rationale, this thesis proposes an LLM-driven workflow for test-oracle generation that is language-generalised at the pipeline level. While prior research has predominantly focused on monolingual infrastructures, the proposed approach is explicitly designed to be language-generalised. Figure 1 summarises the core idea: ① Subject-specific artefacts that characterise the method under test (MuT) and its intended behaviour are collected (e.g., code and accompanying natural-language context). ② These artefacts are organised into a structured prompt representation that specifies what information is provided to the LLM and in what form. ③ These prompts are provided to an LLM to synthesise oracle implementation in a prescribed interface. ④ The generated oracle is applied within an evaluation harness to produce a standardised verdict for observed executions, using a three-valued outcome: `PASSING`, `FAILING`, or `UNDEFINED`. Where necessary, lightweight post-processing is applied to ensure the generated oracle is executable within the harness (e.g., resolving minor formatting or integration mismatches).

The primary evaluation setting uses Java methods with bug reports sourced from Defects4J [16] and test artefacts mined from EvoRepair [31]. Oracles are synthesised as Python modules that can be executed within a unified harness and their effectiveness is assessed by whether they distinguish buggy from expected behaviour under benchmark-derived executions. This setting grounds the main empirical claims of the thesis in an execution-based evaluation with method-level subjects.

A smaller complementary case study then applies the same oracle-generation workflow to a second language - namely Python subjects from BugsInPy [44], contextualised via Tests4Py [36]. Here, method-level code, docstrings/documentation, and bug reports are used to generate oracles, which are assessed primarily through structured qualitative analysis (e.g., alignment with bug-report symptoms and available benchmark oracles) rather than large-scale execution-based measurement.

## 1.1 Research Questions

The overarching goal of this thesis is to investigate the feasibility and practical effectiveness of LLM-driven test oracle generation in a cross-language setting.

The specific objectives of the research are:

- To develop a reusable workflow for generating test oracles with LLMs across languages and projects.
- To assess how well the generated oracles support reliable bug-focused evaluation in benchmark settings.
- To identify prompting and interface constraints that improve oracle usability and reduce weak outputs.

From these objectives, the research addresses the following research questions:

**RQ1:** To what extent can GPT-4o-Mini generate correct and strong test oracles from method-level code and natural-language documentation?

**RQ2:** How effective are the generated oracles at distinguishing buggy from fixed behaviour in an execution-based benchmark evaluation?

**RQ3:** Which oracle-generation behaviours and failure modes are consistent across Python and Java, and which appear language-dependent?

RQ2 is answered primarily through execution-based experiments on Java methods from Defects4J using benchmark-derived test suites and ground-truth oracle references where available. The Python component (BugsInPy complementary case study) is analysed qualitatively rather than through execution-based benchmarking. The answers to these questions provide empirical and practical insights into the capabilities and limitations of LLM-driven oracle generation, informing future work on scalable oracle construction and cross-language testing workflows.

Thus, the main contributions of this thesis are as follows:

- **Language-generalised oracle-generation pipeline.** A reusable workflow for synthesising oracles from a stable prompt contract, combining method-level artefacts (MuT code, documentation, bug report) with a language-specific oracle micro-skeleton and a uniform three-valued oracle interface.
- **Two instantiations under a unified oracle interface.** An implementation of the workflow primarily in a Java evaluation setting (Defects4J/EvoRepair) accompanied by a Python case study (BugsInPy/Tests4Py), where only the language-dependent invocation and runtime interfacing components differ.
- **Empirical and qualitative oracle assessment.** Execution-based evaluation of generated oracles in the primary Java setting (buggy vs. expected discrimination), complemented by structured qualitative analysis in the Python case study, including cross-run variance analysis to characterise non-determinism and common failure modes.

## 1.2 Structure

The remainder of this thesis is structured as follows. Chapter 2 reviews relevant literature and positions this work within existing research. It first introduces the oracle problem and established oracle techniques, then discusses recent developments in the use of LLMs for software engineering and code generation. Finally, it surveys LLM-based approaches to test oracle generation, namely the foundational methods and documentation-driven techniques. Chapter 3 provides the technical and experimental foundations required for the subsequent methodology and evaluation. It introduces key terminology and oracle quality concepts used throughout the thesis and describes the benchmarks and artefacts employed (Defects4J and EvoRepair as the primary evaluation context, with BugsInPy and Tests4Py as a complementary case study). Chapter 4 presents the proposed framework and methodology, including the overall workflow, data preparation, prompt design and filtering criteria. Chapter 5 introduces the Defects4J/EvoRepair and BugsInPy/Tests4Py instantiations of the pipeline. Additionally discussed are the mining and translation of benchmark-derived tests, and the procedure used to assess oracle correctness and fault-detection strength. Chapter 6 reports the experimental results, examining the qualitative findings of all generated oracles from the BugsInPy and Defects4J samples. Subsequently, it focuses on the Defects4J-based empirical results data. Chapter 7 discusses these findings, including implications, limitations, and threats to validity. Finally, Chapter 8 concludes the thesis by summarising the main contributions and outlining directions for future research. Additional materials and supplementary information are provided in the Appendix.

# 2 Related Work

Research on test oracles spans decades, motivated by the *oracle problem*: determining whether a program's observed behaviour is correct can be expensive, ambiguous, or infeasible [2]. This chapter first reviews approaches to alleviating the oracle problem that are not large language model (LLM) approaches. The subsequent section discusses how LLMs have been used more broadly in testing and program analysis, before focusing on LLM-based test oracle generation.

## 2.1 Non-LLM Approaches to the Oracle Problem

The oracle problem predates learning-based techniques and reflects the practical difficulty of deciding expected behaviour. Weyuker characterises *non-testable* programs as those for which an oracle does not exist or correctness checking requires extraordinary effort, arguing that routine oracle availability is often an unrealistic assumption [43].

Barr et al. survey mitigation strategies and classify a wide range of manual, specification-based, and automated oracle techniques [2]. Their synthesis emphasises that oracles are frequently partial or approximate, and that testing practice often combines multiple weak signals rather than relying on a definitive correctness criterion.

Pseudo-oracles (or dual coding solutions) compare outputs against an independently implemented version of the same functionality [43]. However, the approach adds substantial development and maintenance cost and is therefore limited in general applicability [2].

Another direction avoids concrete expected outputs by checking behavioural relations. Metamorphic testing defines metamorphic relations that constrain how outputs should change under systematic input transformations, enabling failure detection when exact outputs are unknown [5]. Follow-up work extends this by using such relations to generate additional test cases [6], but suitable relations still require domain knowledge and manual effort.

Automated test generation can also reduce oracle dependence by focusing on structural objectives (e.g. coverage) and using coarse-grained oracles such as crashes or exceptions [27]. In repair-oriented settings, evolutionary testing can act as a behavioural oracle by using the existing test suite to distinguish plausible patches that preserve regression behaviour [31].

Overall, these approaches reduce oracle cost via approximation, comparison, or behavioural consistency, but typically depend on additional specification effort or existing artefacts. This context motivates LLM-based oracle generation as another mechanism for approximating intended behaviour from available natural-language and code artefacts.

## 2.2 LLMs in Testing and Program Analysis

LLMs have rapidly moved from general-purpose code assistants to components of end-to-end testing and analysis workflows. Recent surveys and roadmaps show that LLMs are now applied across the unit-testing lifecycle, including test generation, test augmentation, bug reproduction, program repair support, and test maintenance. Many approaches rely

on hybrid pipelines that combine LLM generation with execution, ranking, and repair heuristics [1, 7, 14]. Across these settings, a recurring theme is that LLMs behave as stochastic generators: practical usefulness depends less on a single prompt and more on engineering constraints (context selection, compilation checks, execution feedback, filtering, and repair loops) that increase the yield of executable and semantically meaningful artefacts [1, 7].

A core application is unit test generation. Empirical evidence suggests that, when provided with function implementations and usage cues, LLMs can generate tests that achieve substantial coverage, but that reliability is sensitive to prompt context and the surrounding harness (imports, fixtures, environment assumptions) [34]. In parallel, pre-LLM neural approaches such as AthenaTest demonstrate that learning-based test generation can produce readable, developer-like tests when trained on large corpora of focal methods and associated tests [37]. More recent work explores hybridisation with classical test generation: CodaMOSA, for example, queries an LLM to propose candidate tests when search-based generation stalls, using those candidates to escape coverage plateaus and redirect exploration [20]. Beyond example-based unit tests, LLMs have also been investigated for producing property-based tests from natural-language specifications, highlighting both the promise of documentation-derived properties and the need for rigorous soundness evaluation [40].

LLMs have also been used as semantic translators in workflows that start from natural-language artefacts. LIBRO targets bug reproduction by prompting an LLM with a bug report and then ranking and post-processing generated tests to identify failure-reproducing candidates. This demonstrates that LLMs can operationalise natural-language problem descriptions, but also that success depends heavily on downstream validation steps [18]. Complementary to generation from scratch, test completion models such as TECO study how learning test-specific semantics can support completing partially written tests, reinforcing the importance of modelling the typical test structure (arrange–act–assert) and, crucially, the oracle/assertion step [23].

This broader context matters directly for LLM-based oracle generation because many of the same failure modes reappear: (i) compilation and execution fragility, (ii) sensitivity to context selection, and (iii) the tendency to produce oracles that mirror observed behaviour rather than capturing the intended specification. Recent work explicitly studies this "actual vs. expected behaviour" issue for LLM-generated test oracles, showing that LLM-based test generation can still drift towards asserting current behaviour, motivating stronger grounding signals and validation designs when synthesising oracles [19].

## 2.3 LLM-based Test Oracle Generation

Existing work on LLM-based oracle generation is typically introduced through foundational approaches that demonstrate feasibility and define baselines. On this foundation, subsequent research has increasingly focused on documentation-driven techniques that leverage natural-language artefacts as the principal signal for oracle synthesis.

### 2.3.1 Foundational approaches

One of the first substantial contributions to LLM-driven test oracle generation was proposed by Hossain et al. (2025) with the introduction of TOGLL, a framework that utilises fine-tuned LLMs to generate test oracles for Java projects [12]. TOGLL separates the oracle generation process from test prefix creation and systematically examines the impact of different levels of input context, including test prefixes, documentation, and method signatures. Their experimental study, conducted across 110 Java projects, demonstrated that TOGLL produces 3.8 times as many correct assertion oracles and 4.9 times as many exception oracles than the prior state-of-the-art model TOGA [10]. Moreover, TOGLL detected ten times more unique bugs than TOGA and outperformed EvoSuite in real-world defect detection. The study also highlighted the importance of providing rich contextual information to the LLMs, as seen in the superior performance of the most comprehensive prompt variant. However, the approach continues to face challenges related to ambiguous documentation and false positives.

In parallel, Kang et al. (2023) introduced LIBRO, a complementary framework that focuses on LLM-based test case generation from natural language bug reports rather than direct oracle generation [18]. LIBRO employs prompt engineering combined with ranking and selection mechanisms to filter and prioritise valid test cases. Evaluated against the Defects4J benchmark, LIBRO successfully generated failure-reproducing test cases for 33% of studied bugs, achieving an accuracy of 71.4% in distinguishing valid from invalid test cases. While LIBRO does not directly target oracle generation, its contributions to structured prompt design and validation strategies are relevant to the broader field of LLM-driven software testing.

### 2.3.2 Documentation-based approaches

A deeper research direction focuses on leveraging documentation, particularly Javadocs, to guide test oracle generation. Hossain et al. (2024) investigated how various components of Javadoc documentation, such as `@return`, `@param`, and `@throws` tags, affect LLM-generated oracle quality [13]. Their findings indicate that Javadoc comments alone can produce test oracles with accuracy nearly equivalent to those generated using full method implementations. This challenges the prevailing assumption that source code is essential for generating high-quality oracles. Incorporating Javadocs led to a 10-20% improvement in oracle accuracy, with the description and `@return` elements proving to be the most influential.

Jiang et al. (2024) further advanced documentation-based oracle generation by focusing on validating API contract compliance in client code using LLMs and JDK Javadocs [15]. Their approach involves structured prompting strategies, including few-shot learning and chain-of-thought reasoning to generate oracles that validate expected outputs and exception conditions for standard Java library APIs. The evaluation demonstrated that 97% of generated oracles compiled successfully and 96% accurately captured the intended behaviours. However, the applicability of this method is limited to standard Java APIs and does not generalise to arbitrary project-specific codebases.

# 3 Background

This chapter defines the terminology and experimental context used throughout the thesis. This serves to establish a precise vocabulary, introduce the benchmarks and artefacts used in the empirical study, and outline the cross-language execution setting and large language model (LLM) specific considerations that are required to interpret the system architecture and results.

## 3.1 Terminology

To avoid ambiguity, this thesis uses the following definitions.

**Test oracle.** A *test oracle* is any mechanism that determines whether an observed program outcome should be considered correct or incorrect for a given test case [2]. In this thesis, an oracle is an executable predicate implemented as Python code that consumes the observed execution outcome (return value) of a singular method under test (MuT) and returns a verdict.

**Correct vs. strong.** An oracle is *correct* if its verdicts align with the intended program behaviour, i.e., it does not systematically accept faulty behaviour or reject correct behaviour [2]. An oracle is *strong* if it is discriminative, i.e., it detects meaningful behavioural differences and is not vacuous (e.g., checking only that execution terminates) [12]. In this thesis, oracle strength is operationalised primarily via the oracle's ability to distinguish buggy from expected program behaviour in an execution-based benchmark setting (Chapter 6).

**Passing and failing.** A test execution is *passing* if the oracle returns a verdict indicating that the observed outcome conforms to the oracle's expectation; it is *failing* if the oracle returns a verdict indicating a violation. Where necessary, this thesis also uses an *undefined* category for executions that cannot be meaningfully judged (e.g., harness error, environment failure, or missing context), to avoid conflating infrastructure failures with semantic failures.

**Bug report.** A *bug report* is a natural-language description of an observed defect, typically including symptoms, expected behaviour, and (sometimes) reproduction information. Bug reports provide an intent signal that is often absent from code alone and are therefore treated as a first-class input to prompting in this work, analogous to their use in LLM-based bug reproduction [18].

**Documentation.** *Documentation* refers to developer-provided natural-language artefacts describing behaviour, inputs, outputs, and exceptional conditions. For Python, this includes docstrings and adjacent documentation; for Java, this includes Javadocs (e.g., `@param`, `@return`, `@throws`) [13, 15]. Documentation is treated as a behavioural specification proxy, particularly relevant for oracle synthesis.

| Resource | Language | Primary unit | Artefacts used |
|---|---|---|---|
| Defects4J [16] | Java | Bug + project version | Buggy/fixed program versions; metadata for reproducibility |
| EvoRepair [31] | Java | Bug + repair context | Benchmark-derived tests and curated oracle references consumed by this thesis |
| BugsInPy [44] | Python | Bug + project version | Buggy/fixed versions for qualitative oracle assessment |
| Tests4Py [36] | Python | System-level bug | Benchmark tests/oracles to contextualise BugsInPy cases |

Table 1: Benchmarks and artefacts used in this thesis.

## 3.2 Benchmarks and Artefacts

This thesis uses established benchmarks to ground oracle evaluation in real defects and reproducible program versions. Broadly, the evaluation relies on: (i) Java defects and associated artefacts for execution-based assessment, and (ii) a smaller Python sample for complementary qualitative analysis. Table 1 provides an overview of the benchmarks used in this thesis, particularly the artefacts they provide for experimentation.

### 3.2.1 Defects4J

Defects4J is a curated dataset of 854 real-world Java bugs designed to enable controlled testing studies [16]. Each Defects4J entry provides reproducible program versions, including a buggy version and a corresponding fixed version, along with metadata to checkout and build the subject consistently. Each bug is accompanied by a developer-written test suite that includes at least one triggering test that fails on the faulty version and passes on the fixed version, enabling regression-style evaluation. Defects4J also provides a supporting infrastructure that standardises common experimental tasks (e.g., checking out versions and executing tests) across subjects. In this thesis, Defects4J provides the primary execution-based evaluation context: LLM-generated oracles are assessed by running them against the buggy checkout with failing values that either invoke the bug or passing values that don't. This enables the analysis of whether the oracle verdict aligns with the expected behavioural difference.

### 3.2.2 EvoRepair

EvoRepair proposes an evolutionary testing approach in the context of program repair and provides artefacts that enable systematic evaluation across bugs [31]. Concretely, EvoRepair is implemented as an extension of EvoSuite and formulates a co-evolutionary workflow in which candidate patches and tests are generated in tandem. To mitigate the missing-specification issue for generated tests, EvoRepair constructs 39 additional oracle assertions by instrumenting buggy Defects4J programs using lightweight behavioural

knowledge extracted from bug reports. This thesis consumes EvoRepair resources primarily as (i) a source of benchmark-derived tests relevant to specific methods/bugs and (ii) a source of manually curated oracle references. These artefacts are mined and adapted to the execution setting of this thesis (including translation into a Python-accessible form for JPype execution), providing a grounded basis to evaluate whether an LLM-generated oracle can distinguish buggy behaviour in practice.

### 3.2.3 BugsInPy

BugsInPy is a database of real bugs in Python programs intended to support controlled debugging and testing studies [44]. Similar to Defects4J, it provides reproducible buggy and fixed program versions. At the infrastructure level, BugsInPy follows a Defects4J-style architecture consisting of a bug database of 493 bugs, a database abstraction layer for checkout/build, and a test execution framework for running experiments consistently across subjects. Beyond basic test execution, the framework also supports common testing/debugging activities such as selecting test subsets, generating inputs via fuzzing, and computing mutation and coverage metrics. In this thesis, BugsInPy is used as a small complementary case study to observe oracle-generation behaviour in Python, where LLM code-generation performance is commonly benchmarked and model competence is often comparatively strong (as discussed in Chapter 1).

### 3.2.4 Tests4Py

Tests4Py is a benchmark for the system testing of Python projects comprised of 73 bugs [36]. It is derived from BugsInPy and augments each bug with an explicit oracle, a system interface, and the capability to incorporate both system and unit test generators [36]. To foster diversity in evaluation, Tests4Py provides curated test sets containing both passing and failing tests, and each bug is accompanied by at least one failing unit test [36]. In this thesis, Tests4Py is used to contextualise the Python-side analysis and to relate generated oracles to benchmark outcomes.

## 3.3 Large Language Models and Prompting

LLMs are neural sequence models that generate text by predicting the next token conditioned on preceding context. Modern LLMs are typically based on transformer architecture which replaces recurrence with attention mechanisms and is designed to be highly parallelisable [39]. A practical consequence of this formulation is that LLM behaviour can be steered at inference time through the provided prompt, including instructions, constraints, and in-context examples.

Prompting is often described in terms of *zero-shot*, *one-shot*, and *few-shot* settings, depending on whether the prompt contains no examples, a single example, or multiple examples of the desired input–output behaviour [3]. Beyond providing examples, prompt engineering commonly involves structuring the task (e.g., specifying output formats or intermediate steps) and iterating on prompt context to improve correctness and

consistency; for reasoning-heavy tasks, chain-of-thought prompting has been shown to improve performance by supplying exemplars that explicitly encode intermediate reasoning steps [42].

In addition to prompt content, decoding parameters influence the variability of generated outputs. In particular, `temperature` controls sampling randomness: lower values produce more focused and deterministic outputs, while higher values increase diversity; OpenAI's API documentation further recommends tuning either `temperature` or `top_p` (nucleus sampling), but not both simultaneously [24].

# 4 System Architecture

This chapter describes the architecture of the proposed oracle-generation workflow. The presentation proceeds in two stages. The language-generalised pipeline is introduced, making explicit which parts are reusable across subjects and which require language-specific adaptation. This pipeline is instantiated for the Java-based Defects4J/EvoRepair evaluation and the Python-based BugsInPy/Tests4Py case study in Chapter 5.

## 4.1 Design Goals and Architectural Principles

The system is designed around three goals:

**G1: Dataset- and language-generalised pipeline.** At the architectural level, oracle generation is defined as a sequence of modules that can operate on any subject as long as the required inputs can be provided in a predefined schema. Concretely, a new subject can be integrated once the *prompt contract* can be populated with method-level artefacts (e.g., method under test (MuT) source code, documentation, bug report).

**G2: Stable prompt contract, modular extractors.** Only the *prompt contract* is treated as a stable intermediate artefact: a structured record that defines which information must be supplied to the large language model (LLM) and in what format. All other components (extractors, harness adapters, test translators) are modular and can be swapped per dataset/language.

**G3: Traceability and reproducibility.** All LLM calls are traceable via stored prompt instances and generated oracle files. To account for non-determinism, oracle generation is repeated across multiple runs under identical configurations, enabling analysis of variance across generations (Section 4.2.3).

## 4.2 Pipeline Overview

Figure 2 summarises the workflow. Conceptually, the pipeline consists of three stages:

1. ① **Artefact acquisition.** Subject-specific extractors collect the MuT source code, relevant documentation, and a bug report (or equivalent natural-language description).

2. ② **Prompt engineering.** Extracted artefacts are mapped into a *prompt contract* instance and rendered into a user prompt using a fixed template, together with a language-specific oracle micro-skeleton.

3. ③ **Oracle generation.** The LLM is invoked to produce an executable oracle implementation, conditioned using one-shot learning with an exemplar user prompt/oracle pair.

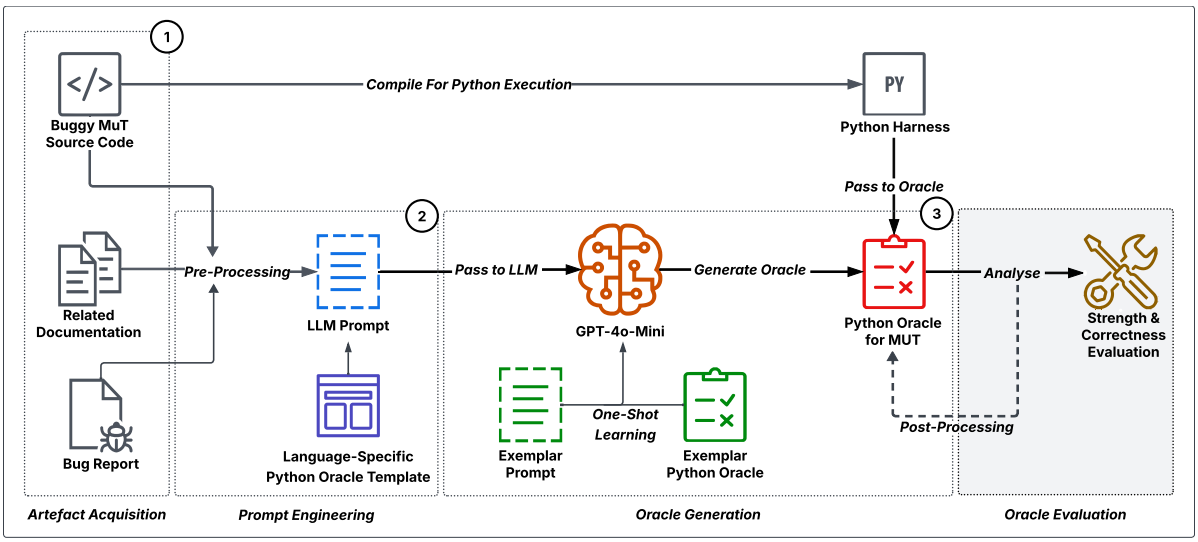


Figure 2: End-to-end oracle-generation and evaluation pipeline.

Whilst not a part of the system architecture's core pipeline, the thesis workflow heavily relies on evaluation varying methods to determine the answers to the research questions detailed in Chapter 1.1. The exact methodology used for these evaluations are dependent on the available artefacts relative to the test subject, and thus will be discussed with the experimentation settings of this in the following chapter.

The remainder of this section details each stage as a reusable module.

### 4.2.1 Artefact acquisition

**Artefacts for the prompt contract.** The prompt contract is a structured schema that defines the information required to generate an oracle for a MuT. It acts as the interface between extraction and generation: once populated with the acquired artefacts, downstream steps do not depend on the originating dataset.

A typical prompt contract instance contains:

- **Identity metadata:** subject identifier (e.g., project name, bug ID), language, MuT name, enclosing class/module, and file path.
- **Oracle scaffold:** language-specific micro-skeleton that defines the oracle interface and how the MuT is invoked.
- **Behavioural context:** bug report (natural language), and method-level documentation (plus optionally class-level context and helper-method documentation).
- **Program context:** buggy MuT source code.

Table 2 provides a concrete overview of the fields of the prompt contract.

Table 2: Prompt contract fields used to populate the user prompt template.

| Field | Description |
|---|---|
| language | Used to select the appropriate micro-skeleton |
| project_name | Project/package name to which the buggy MuT belongs |
| mut_name | Method name |
| container_name | Enclosing class/module name |
| mut_docs | Available MuT documentation |
| container_docs | Optional, available enclosing class/module documentation |
| helper_docs | Optional, available method documentation of any helpers used in the MuT |
| bug_report | Natural-language description of the defect |
| buggy_mut_code | Buggy method body |

### 4.2.2 Prompt engineering

**System prompt.** Oracle generation uses a universal system prompt (Appendix 29) that specifies high-priority behavioural constraints for the model and fixes the expected output contract [25]. Concretely, the prompt (i) assigns the model a domain role ("software testing engineer") to steer domain framing and reduce stylistic drift, (ii) defines the oracle interface (single MuT invocation and a three-valued verdict), and (iii) constrains interpretation of evidence by requiring a general, runtime-computable bug-signature predicate rather than hard-coded examples. Role assignment is widely used to steer response style and task framing in prompt-engineering practice and surveys [33]. Accordingly, in this thesis the role instruction is used to stabilise format and enforce the oracle contract.

**User prompt template.** The user prompt is produced from a template populated by the prompt contract instance. The template is intentionally stable across subjects; variability is introduced only through contract values and the language-specific micro-skeleton. To reduce ambiguity between *instructions* and *data*, the template uses explicit section headers (e.g., `[DOCUMENTATION]`, `[BUG REPORT]`, `[METHOD UNDER TEST]`) and encloses code artefacts in fenced code blocks (triple backticks) annotated with the programming language (e.g., ```` ```java ````, ```` ```python ````). Fenced blocks are a standard Markdown mechanism for unambiguously delimiting code content and optionally attaching an "info string" (commonly used for language identification), improving readability and reducing the risk that code tokens are misinterpreted as natural-language instructions [8].

Finally, the template follows an instruction "sandwich" pattern: it begins with a concrete task directive and ends with a final directive restating hard constraints (e.g., no hard-coding, compute predicates at runtime). This mirrors established prompt-injection defences in which untrusted content is placed between two trusted instruction layers, helping preserve the intended task framing even when intermediate sections contain adversarial or noisy text [32].

**Oracle micro-skeleton.** The oracle micro-skeleton is a minimal code scaffold that standardises: (i) how the MuT is invoked, (ii) which execution outcome is observed, and (iii) how verdicts are returned. Oracles expose a single entry point (e.g., `oracle(self / object instance calling the MuT, args)`), consume the captured execution outcome, and return one of: `PASSING`, `FAILING`, or `UNDEFINED`. This design is inspired by the verdict-oriented structure used in system-testing benchmarks, specifically Tests4Py-style oracles. The invocation pattern aligns with method-level checking common in repair-oriented oracle definitions, specifically EvoRepair-style oracles. Language-specific invocation details are confined to the micro-skeleton and the harness adapter. The purpose of this layer is to (i) invoke the MuT in its native runtime, (ii) capture observable outcomes (return values and/or exceptions), and (iii) normalise these outcomes into a uniform structure to be examined through the oracle's predicates. Crucially, this isolates language dependencies from prompt assembly and oracle generation: once a suitable micro-skeleton exists for a language/runtime, the remainder of the pipeline remains unchanged. The generic oracle micro-skeleton used as the common basis for language-specific instantiations is provided in Appendix 30).

In practice, multiple implementation strategies are possible depending on the target ecosystem. For Python MuTs, the harness can import and invoke the target function directly within the same process. For Java, one approach is embedding a JVM and calling methods through a Python-accessible boundary such as `JPype`[1], enabling direct access to Java objects and exceptions. For .NET/C#, analogous interop can be achieved via `pythonnet`[2], allowing Python to load CLR assemblies and invoke managed methods. For R, `rpy2`[3] enables embedding an R interpreter and calling R functions with automatic data conversion. For JavaScript, lightweight embeddable runtimes such as `QuickJS`[4] can be invoked from Python via bindings to execute scripts and collect results. Alternative methods involve executing an isolated interpreter via subprocess invocation for the specified language. While these mechanisms differ in runtime embedding and marshalling semantics, they serve the same architectural role: exposing language-specific execution through a stable, language-generalised oracle interface.

### 4.2.3 Oracle generation

**One-shot conditioning with an exemplar.** Oracle generation uses a one-shot setup: a fully populated exemplar prompt is provided as the first user message, followed by the exemplar oracle as an assistant message. Finally, the target subject prompt is supplied as the second user message, as demonstrated in Listing 1. This conditions the LLM on (i) the expected prompt structure and (ii) the oracle style consistent with the micro-skeleton.

[1] `https://github.com/jpype-project/jpype.git`
[2] `https://github.com/pythonnet/pythonnet.git`
[3] `https://github.com/rpy2/rpy2.git`
[4] `https://github.com/PetterS/quickjs.git`

```
def prompt_llm(api_key: str,
               model: str,
               system_prompt: str,
               user_prompt: str,
               exemplar_code: str,
               exemplar_user_prompt: str) -> str:

    client = OpenAI(api_key=api_key)

    messages = []

    messages.append({"role": "system", "content": system_prompt})
    messages.append({"role": "user", "content": exemplar_user_prompt})
    messages.append({"role": "assistant", "content":
        f"```python\n{exemplar_code}\n```"})
    messages.append({"role": "user", "content": user_prompt})

    completion = client.chat.completions.create(model=model,
        messages=messages, temperature=0)
    return completion.choices[0].message.content
```

Listing 1: One-shot prompting sequence: system message, exemplar user prompt, exemplar oracle (assistant), and target user prompt.

**Batch generation and storage.** For each subject, the oracle generation script renders the user prompt from the prompt contract and calls the model. The oracle code is extracted from the response and written to a dedicated Python file named after the subject identifier. To study non-determinism, the same generation procedure is repeated multiple times (five runs in this thesis), producing a set of oracle variants per subject.

**Post-processing and execution readiness.** The LLM output is treated as executable oracle code that must be aligned with (i) an often underspecified natural-language problem description (bug report and documentation) and (ii) the concrete execution environment of the subject under test. Accordingly, post-processing is not limited to formatting or extraction of the generated function body; it also includes resolving ambiguities and mismatches that arise when translating high-level behavioural descriptions into an operational predicate. Empirical studies on LLM-based unit test generation report that generated artefacts frequently require repair or curation to compile, execute, and meaningfully reflect intended behaviour in a project context [26, 41, 46].
The next section explains how these stages are instantiated and evaluated in the two experimental settings. These instantiations and all related code and setup can be found in the public GitHub repository: `LINK.TO.REPO`

# 5 Experimental Setup

This chapter describes how the architecture introduced in Chapter 4 is instantiated and evaluated in two settings: (i) Defects4J with EvoRepair-derived artefacts for execution-based assessment on Java subjects, and (ii) BugsInPy with Tests4Py resources as a complementary Python case study. The chapter first summarises the experimental configuration shared across both settings, before detailing the dataset-specific instantiations. All related code, generated oracle and results can be found in the project repository[5].

## 5.1 Shared Configuration and Evaluation

Both experimental settings use the same generation procedure and similar evaluation concepts to ensure comparability of outcomes across subjects and across repeated generations.

**Model and decoding configuration.** All oracle generations are produced using `GPT-4o-mini` with `temperature=0`. A deterministic decoding configuration is used to reduce variance attributable to sampling and to focus cross-generation variance on model and context effects rather than random exploration.

**Prompting setup.** Oracle synthesis follows the one-shot conditioning strategy introduced in Section 4.2.3. Concretely, each target prompt is preceded by a single exemplar prompt–oracle pair to condition the model on the expected prompt structure and the required oracle micro-skeleton interface. The exemplar itself is first constructed without examples (i.e., a zero-shot reading), but it is provided explicitly to stabilise formatting and reduce contract violations.

**Generations and controlled conditions.** For each subject, oracle generation is repeated for five generations under identical experimental conditions (same model, same temperature, same prompt template, same exemplar, and the same extracted subject artefacts). This enables analysis of cross-generation stability and variance while keeping the surrounding pipeline constant.

**Evaluation concept.** Independent of dataset, oracle evaluation combines qualitative alignment checks with execution-based evidence where feasible. The evaluation focuses on:

- **Bug-report alignment (qualitative).** Cross-checking whether the oracle predicates operationalise the defect symptoms and expected behaviour described in the bug report.
- **Oracle reference comparison (qualitative).** Comparing the generated oracle logic against a reference oracle when available (e.g., benchmark-provided or curated oracles).

[5] `https://github.com/liadanxa/LanguageGeneralisedTestOracleGenerationUsingLargeLanguageModels.git`

| Project–Bug | MuT | Class |
|---|---|---|
| Chart-1 | getLegendItems | AbstractCategoryItemRenderer |
| Lang-35a | add(two args) | ArrayUtils |
| Lang-35b | add(three args) | ArrayUtils |
| Lang-50a | getDateInstance | FastDateFormat |
| Lang-50b | getDateTimeInstance | FastDateFormat |
| Lang-61a | deleteAll | StrBuilder |
| Lang-61b | replaceAll | StrBuilder |
| Math-8 | sample | DiscreteDistribution |
| Time-4 | with | Partial |

Table 3: Sample of Defects4J/EvoRepair subjects (project/bug ID, method under test, and enclosing class).

- **Execution-based discrimination (empirical).** Executing the oracle via the harness on inputs that invoke the bug and on inputs that do not, and recording whether the oracle distinguishes behaviours as expected.

The following two sections describe how this shared setup is instantiated for the Defects4J/EvoRepair and BugsInPy/Tests4Py settings respectively. The dataset-specific subsections retain the same evaluation lens but differ in the available artefacts, harness integration, and the degree to which reference oracles are available.

## 5.2 Defects4J with EvoRepair

This section instantiates the pipeline for the primary evaluation setting: Java methods under test (MuTs) from Defects4J, with tests and reference oracles mined from EvoRepair resources.

**Subjects and identifier normalisation.** The EvoRepair subject set comprises 39 bugs from four projects (Chart, Lang, Math, and Time). Three Lang bugs require two distinct MuT-level reference oracles (Lang-35, Lang-50, and Lang-61). These are represented as separate subjects using a suffix convention (e.g., `{BugID}_a` and `{BugID}_b`). Overall, this yields 42 MuT-level subjects in total, compiled into a JSON index[6]. Per project, this corresponds to 3 subjects for Chart, 18 for Lang, 18 for Math, and 3 for Time. For each subject, the prompt contract records the project, bug identifier, MuT name, and enclosing class (Table 3). The subject `time-4` is used as the one-shot exemplar, leaving the remaining 41 subjects for oracle generation.

[6] `https://github.com/liadanxa/LanguageGeneralisedTestOracleGenerationUsingLargeLanguageModels/tree/master/Defects4J_experiment/artefacts`

**Artefact acquisition.** Dataset-specific scripts populate the prompt contract using:

- **Buggy MuT source code[7]:** extracted from the Defects4J buggy checkout.
- **Documentation[8]:** Javadocs for (i) the MuT, (ii) the enclosing class, and (iii) helper methods referenced by the MuT where available.
- **Bug reports[9]:** mined from Defects4J artefacts and exported into plain text for prompt inclusion.

Once the contract is populated, prompt rendering follows the shared template described in Section 5.1.

**Harness and micro-skeleton.** The Java micro-skeleton[10] as part of the user template uses JPype-based invocation and normalises Java exceptions into a uniform structure. This constrains each oracle to (i) invoke the MuT exactly once and (ii) return a three-valued verdict consistent with the shared oracle interface.

**Reference oracles and tests.** Qualitative checks use the EvoRepair reference oracles and the mined bug report text to assess whether each generated oracle operationalises an appropriate bug-signature predicate. Empirical evaluation uses tests mined from EvoRepair's experimental results[11].

**EvoRepair test procurement.** Execution-based evaluation in the Defects4J setting requires concrete MuT invocations that (i) trigger the bug and (ii) do not trigger the bug. These invocations construct two benchmark-derived test pools per subject: a *failing* pool (bug-triggering) and a *passing* pool (non-bug-triggering). The procurement process is designed to yield MuT-scoped, executable harness inputs while keeping evaluation conditions comparable across subjects.

**Mining passing and failing MuT invocations.** Test fragments are extracted from EvoRepair-generated test suites and partitioned using EvoRepair's instrumentation. Concretely, MuT call sites annotated with `[Defects4J_BugReport_Violation]` are treated as failing invocations, while MuT call sites without such markers are treated as passing invocations. To ensure subject correctness at the MuT level, the mining step retains only fragments that (i) invoke the expected MuT by name and (ii) match the MuT signature derived from the prompt-contract artefacts (argument count). Listing 2 illustrates the core logic of the mining procedure. The script traverses EvoRepair iteration

[7] https://github.com/liadanxa/LanguageGeneralisedTestOracleGenerationUsingLargeLanguageModels/tree/master/Defects4J_experiment/artefacts/buggy_methods

[8] https://github.com/liadanxa/LanguageGeneralisedTestOracleGenerationUsingLargeLanguageModels/tree/master/Defects4J_experiment/artefacts/javadocs

[9] https://github.com/liadanxa/LanguageGeneralisedTestOracleGenerationUsingLargeLanguageModels/tree/master/Defects4J_experiment/artefacts/bug_reports

[10] https://github.com/liadanxa/LanguageGeneralisedTestOracleGenerationUsingLargeLanguageModels/tree/master/Defects4J_experiment/prompts

[11] https://figshare.com/s/d800c8b6498d207f2cdd?file=42905476

directories, locates generated EvoSuite test classes for the target class under test, extracts individual test methods, and retains only those methods that contain a valid MuT invocation with an associated bug-report-violation marker. For comparability and to bound runtime, at most 100 passing and 100 failing invocations are retained per subject.

```
def has_defects4j_violation_marker(lines, call_line_index):
    k = 1
    non_empty_seen = 0
    while k <= 6 and non_empty_seen < 2:
        idx = call_line_index - k
        if idx < 0:
            break
        txt = lines[idx].strip()
        if txt == "":
            k += 1
            continue
        non_empty_seen += 1
        if "[Defects4J_BugReport_Violation]" in txt:
            return True
        k += 1
    return False

def count_call_args(call_line):
    """
    Count arguments of the MuT call to ensure
    signature compatibility.
    """
    m = re.search(r'\((.*)\)', call_line)
    if not m:
        return None
    inside = m.group(1).strip()
    if inside == "":
        return 0
    return len([p for p in inside.split(",") if p.strip()])
```

Listing 2: Excerpt of the EvoRepair test mining logic used to identify MuT-level bug-triggering tests via violation markers and signature checks.

The output of this mining step consists of raw test fragments grouped by subject and labelled as passing or failing, as shown in Appendix 8.3. These initial counts represent *pre-refinement* results: they may still contain duplicate tests, accidentally extracted methods that do not meaningfully exercise the MuT, or fragments that cannot be translated into an executable Python–JPype form. Accordingly, the mined totals are treated as an upper bound on usable test cases and are subsequently refined during translation and execution preparation, as described next.

**Java-to-JPype test translation.** Mined Java fragments are translated into Python scripts that invoke the same MuT through the JPype boundary. The translation preserves concrete input values and the MuT invocation context, while adapting Java-specific constructs into JPype-compatible equivalents. Each translated fragment is trimmed to retain a single, well-scoped MuT invocation that is compatible with the oracle micro-skeleton interface.

The translation process is implemented by a dedicated script[12] and proceeds in a fixed sequence of rewrite steps. These include (i) resolving explicit and default Java imports into fully qualified class references, (ii) rewriting EvoSuite mock classes to their corresponding JDK types, (iii) converting Java literals, array initialisations, object constructions, and casts into their JPype equivalents, and (iv) normalising static factory calls (e.g., `.from(...)` to `.from_(...)`). Where necessary, primitive and reference casts are made explicit using `JInt`, `JDouble`, or `JObject` to ensure correct JVM dispatch at runtime.

Finally, the translated test bodies are trimmed in a MuT-aware manner to retain only the execution fragment relevant to the oracle evaluation. For failing tests, this corresponds to the MuT invocation following the nearest `[Defects4J_BugReport_Violation]` marker; for passing tests, the fragment is truncated after the final MuT call. This ensures that each translated test script executes a single, well-scoped MuT invocation compatible with the oracle interface. Listings 3 and 4 illustrate this process by showing a representative Java test fragment and its translated JPype-based Python equivalent.

```
1 public void test25667() throws Throwable {
2     try {
3       // Undeclared exception:
            org.evosuite.runtime.mock.java.lang.MockRuntimeException
4       // [Defects4J_BugReport_Violation]
5       NumberUtils.createNumber("---0x(?: s|[s&&g^ ]])s*");
6     } catch(Throwable t) {
7       verifyOracleException(t);
8     }
9 }
```

Listing 3: Excerpt of a mined EvoRepair Java test fragment invoking the MuT for Lang-7.

[12] https://github.com/liadanxa/LanguageGeneralisedTestOracleGenerationUsingLargeLanguageModels/tree/master/Defects4J_experiment/test_procurement/translation

| Project–Bug | MuT | Module |
|---|---|---|
| TheFuck-1 | get_new_command | pip_unknown_command |
| Cookiecutter-3 | read_user_choice | prompt |
| PySnooper-2 | __init__ | Tracer |
| PySnooper-3 | get_write_function | snoop |
| Youtube-dl-3 | unescapeHTML | utils |

Table 4: BugsInPy/Tests4Py case-study subjects.

```
# Test 25667
# --- translated method: test25667 ---
# Undeclared exception:
    org.evosuite.runtime.mock.java.lang.MockRuntimeException
# [Defects4J_BugReport_Violation]
result = oracle_createNumber(NumberUtils, "---0x(?: s|[s&&g^ ]])s*")
__record_result__('test25667', 'FAILING', result)
```

Listing 4: JPype-translated Python test fragment executing the same MuT call.

**Post-processing and usability filtering.** After translation, an automated refinement step normalises generated scripts to reduce execution noise (e.g., import handling and JVM initialisation patterns). A final usability filtering step removes duplicate fragments that do not execute the MuT under the harness constraints and other non-executable cases. The resulting post-processed passing and failing pools constitute the final usable test material for empirical oracle evaluation; their subject-level sizes are reported in Chapter 6.

## 5.3 BugsInPy with Tests4Py

This section instantiates the pipeline for Python and serves as a small-scale complementary case study. It demonstrates transfer of the shared setup to a second language once the prompt contract can be populated with comparable MuT-level artefacts.

**Subjects and selection constraints.** Subjects are selected to preserve a MuT-level unit of analysis comparable to the Defects4J setting. Selection prioritises: (i) availability of a natural-language bug description (many BugsInPy bugs are commit-defined without a narrative), (ii) a single dominant MuT to avoid multi-location fixes that blur oracle scope, and (iii) sufficient documentation (docstrings or project documentation), or the ability to curate a minimal behavioural description when absent. A candidate pool of five subjects is curated (Table 4). The subject `thefuck-1` is used as the one-shot exemplar, leaving the remaining subjects for oracle generation.

**Artefact acquisition.** Artefact acquisition is less standardised than in Defects4J and therefore involves additional curation:

- **Buggy MuT source:** extracted from the BugsInPy buggy checkout.
- **Bug report text:** collected manually when not directly available as a structured artefact (e.g., issue text or repository discussion).
- **Documentation:** sourced from docstrings and project documentation where available; if too sparse, a minimal behavioural description is curated from surrounding context, including the GitHub repository corresponding to the project, to populate the contract.

Downstream prompt rendering and generation then follow the shared configuration in Section 5.1.

**Harness and micro-skeleton.** The Python micro-skeleton[13] as part of the user prompt template uses direct imports and native function/method invocation within the same runtime. Exceptions are handled as native Python exceptions rather than bridged JVM types; exception checks in the oracle use Python type names (or fully qualified names where appropriate). Aside from invocation and exception representation, the oracle interface and verdict semantics remain unchanged.

**Reference oracles and evaluation scope.** The Python setting is evaluated primarily qualitatively due to its smaller scale and differing availability of executable benchmark artefacts. Generated oracles are analysed against: (i) the curated bug descriptions, (ii) the corresponding Tests4Py reference oracle where applicable, and (iii) variance across the five generations. This positions BugsInPy/Tests4Py as a complementary lens on oracle-generation behaviour in Python, while execution-based discrimination claims remain grounded in the Defects4J/EvoRepair evaluation.

[13] `https://github.com/liadanxa/LanguageGeneralisedTestOracleGenerationUsingLargeLanguageModels/tree/master/BugsInPy_experiment/prompts`

# 6 Results

This chapter reports the outputs produced by the proposed pipeline and the observations obtained when inspecting and executing the generated test oracles. In accordance with the evaluation segment of the pipeline workflow, a focus is on the alignment with available reference artefacts (e.g. bug reports, EvoRepair/Tests4Py reference oracle equivalents, as well as observed cross-generation variance). Additionally, the empirical results for the executability and observed verdict behaviour of the oracles generated for the Defects4J setting on extracted EvoRepair predefined tests will be presented. Explanations and implications of these observed behaviours are deferred to Chapter 7. All generated oracles can be found in the project repository[14].

## 6.1 Defects4J Generated Oracles

This section reports results for the Defects4J/EvoRepair evaluation setting. As described in Chapter 4, the pipeline produces executable Python-based oracle implementations (via one-shot prompting) and, where benchmark-derived tests are available, supports execution-based measurement of oracle behaviour on passing and failing inputs. Importantly, not all Defects4J targets admit the same level of empirical evaluation because the availability of EvoRepair-derived passing/failing tests varies across subjects.

Across the four Defects4J projects considered (CHART, LANG, MATH, TIME), the evaluation set comprises a total of **42** method under test (MuT) targets. One target, TIME-4, is reserved as the *one-shot exemplar* and is therefore not part of the repeated oracle-generation set. For every remaining MuT target, the pipeline generates **five** oracle variants under identical configuration to account for large language model (LLM) non-determinism. This yields **41** targets with oracle generations, corresponding to **205** generated oracle artefacts in total.

EvoRepair-derived passing/failing tests are only mineable and usable for a subset of targets (as reflected by the preprocessing outcomes). Consequently, execution-based results are only available for that subset, while the remainder of this section reports qualitative properties that can be assessed for *all* generated oracles regardless of test availability. Concretely, coverage is as follows:

- **Total MuT targets:** 42
- **Targets with oracle generations (5 variants each):** 41
- **Targets with usable passing/failing tests for empirical evaluation:** 17

[14] `https://github.com/liadanxa/LanguageGeneralisedTestOracleGenerationUsingLargeLanguageModels.git`

**Reference oracle categories observed in EvoRepair.** Inspection of the EvoRepair "gold standard" oracle reveals a lack of uniformity in structure and intent. Instead, they can be grouped into four recurring categories based on the observable *predicate type* used to detect buggy behaviour. These categories emerge consistently across projects and directly inform the kinds of bug-signature predicates that an LLM-generated oracle must approximate. The four categories are: *exception-based*, *state-based*, *return-value-based*, and *guarded* oracles.

**Exception-based oracles.** Exception-based oracles define buggy behaviour entirely in terms of an exception being thrown under specific conditions. In this category, the oracle does not inspect return values or object state; instead, the presence of a particular exception constitutes the failure signal. The TIME-4 bug is representative of this category. As shown in Listing 5, the EvoRepair oracle wraps the original method execution and raises a runtime exception when the invalid partial state described in the bug report is encountered. The corresponding oracle predicate is therefore purely exception-driven.

```
1 try {
2     new Partial(result.getFieldTypes(), result.getValues());
3 } catch (IllegalArgumentException e1) {
4     throw new RuntimeException("[Defects4J_BugReport_Violation]");
5 }
```

Listing 5: Exception-based EvoRepair oracle for TIME-4.

**State-based oracles.** State-based oracles detect buggy behaviour by inspecting object state or invariants after execution. These oracles assert that certain object invariants hold (or fail to hold). CHART-12 is representative of this category: the oracle checks whether the dataset still maintains a required listener relationship. As shown in Listing 6, a bug-report violation is raised when the dataset does *not* register `this` as a listener, expressing the bug signature as a post-state invariant rather than a return-value or exception condition.

```
1 if (!((org.jfree.data.general.AbstractDataset) dataset).hasListener(this))
      {
2     throw new RuntimeException("[Defects4J_BugReport_Violation]");
3 }
```

Listing 6: State-based EvoRepair oracle for CHART-12.

**Return-value-based oracles.** Return-value-based oracles classify buggy behaviour by asserting that the value returned by the method violates a semantic constraint derived from the bug report. Unlike exception- or state-based oracles, this category operates entirely on the observable return value without relying on thrown errors or object mutation. LANG-41 exemplifies this pattern. As shown in Listing 7, the oracle checks whether the returned string ends with a trailing semicolon. The presence of this suffix constitutes the bug signature identified in the report. When the condition is satisfied, the oracle signals a bug-report violation. The failure predicate is thus expressed directly as a property of the returned value, making the oracle independent of execution context or intermediate state.

```
1 if (result.endsWith(";")) {
2     throw new RuntimeException("[Defects4J_BugReport_Violation]");
3 }
```

Listing 7: Return-value-based EvoRepair oracle for LANG-41.

**Guarded oracles.** Guarded oracles restrict failure detection to executions that satisfy a bug-relevant precondition, thereby avoiding false positives outside the scope of the defect. An initial guard identifies the portion of the input space in which the bug may occur; the failure predicate is evaluated only within this region. LANG-16 illustrates this pattern. As shown in Listing 8, the oracle is activated only when the input string is non-null and uses a hexadecimal prefix (`0X` or `-0X`). Within this guarded context, the oracle attempts to decode the value using `Integer.decode`. A decoding failure corresponds to expected behaviour, whereas a successful decode indicates a violation of the intended specification and triggers a bug-report violation. This structure ensures that the oracle applies its failure logic exclusively to executions that are semantically relevant to the reported bug.

```
1  } catch (NumberFormatException e) {
2      if (str != null && (str.startsWith("0X") || str.startsWith("-0X"))) {
3          try {
4              Integer.decode(str);
5          } catch (NumberFormatException e_decode) {
6              throw e; // expected failure for invalid hex
7          }
8          throw new RuntimeException("[Defects4J_BugReport_Violation]");
9      }
10 }
```

Listing 8: Guarded EvoRepair oracle for LANG-16.

In total, 19 subjects are governed by *exception-based* oracles, while a further 10 subjects use *return-value-based* predicates. *State-based* oracles account for 5 subjects and finally, 8 subjects are characterised by *guarded* oracles. Table 5 provides an overview of the total number of EvoRepair reference oracles for each category. These categories provide a compact characterisation of the oracle styles present in the Defects4J/EvoRepair

| Oracle category | Count |
|---|---|
| Exception oracle | 19 |
| Return-value predicate oracle | 10 |
| Stateful predicate oracle | 5 |
| Guarded signature oracle | 8 |
| **Total** | **42** |

Table 5: Distribution of EvoRepair reference oracle categories in the evaluated Defects4J target set.

references. In the remainder of this chapter, they are used as an analytical lens to describe the qualitative alignment of LLM-generated oracles with their corresponding reference oracles, as well as to observe patterns in the types of oracle predicates produced by the proposed pipeline across subjects and generations.

### 6.1.1 Exemplar Oracle Creation

Due to all subsequent Defects4J oracle generations using one-shot conditioning, the exemplar prompt/oracle pair constitutes a first-class result artefact: its structure and oracle style directly shape the format and failure predicates of downstream generations. In this study, TIME-4 (`Partial.with`) was selected as the exemplar because its MuT is (i) well-scoped at method level, (ii) accompanied by substantial method, class-level as well as helper-method Javadoc, and (iii) associated with a comparatively explicit defect narrative describing an invalid state that should be rejected (duplicate or incompatible field types when merging partials leading to a `IllegalArgumentException`). Together, these properties make the subject representative of the intended oracle interface while keeping the expected bug signature sufficiently concrete to operationalise as a predicate.

The exemplar oracle was produced in a *zero-shot* setting using the universal system prompt (Appendix 29) and a fully populated user prompt instance (curated from the prompt contract). The resulting oracle[15] followed the Java-compatible micro-skeleton: it invoked the MuT exactly once via the runtime bridge, normalised exceptions into a stable identifier, and returned a three-valued verdict (`PASSING`, `FAILING`, `UNDEFINED`). Beyond structural compliance, the prompt constraints were aimed at preventing "example copying"; accordingly, the exemplar oracle was inspected to ensure its failure logic was expressed as a runtime-computable condition (i.e., a bug-signature predicate over observed outcomes) rather than hard-coding concrete values quoted in the bug report.

The exemplar was then qualitatively validated against the EvoRepair oracle reference for TIME-4. This comparison served two purposes. First, it checked that the exemplar oracle targets the same failure phenomenon as the reference (i.e., it treats the bug as the emergence of an invalid partial state rather than a generic input-validation failure).

[15] `https://github.com/liadanxa/LanguageGeneralisedTestOracleGenerationUsingLargeLanguageModels/tree/master/Defects4J_experiment/prompts`

Second, it ensured that the exemplar remained compatible with the downstream evaluation harness assumptions (single-call pattern, exception normalisation, and a stable verdict contract). No post-processing was required for the TIME-4 exemplar oracle; it executed within the harness out-of-the-box and was therefore used unchanged as the assistant-side conditioning oracle in all one-shot generations. Listing 9 highlights the failing predicate for Time-4 in the generated python oracle, equivalent to EvoRepair's counterpart (see Listing 5)

```
FAIL_EXCEPTIONS = {
    "java.lang.IllegalArgumentException": "thrown when fieldType is null
        or invalid",
}
...
try:
    ret = CALL_MUT(receiver_or_class, fieldType, value)

except jpype.JException as e:
    et = e.getClass().getName() # JPype: Java exception class
    msg = str(e)

    if et in FAIL_EXCEPTIONS:
        return "FAILING", f"{et}: {FAIL_EXCEPTIONS[et]}", {"exception":
            et, "args": (fieldType, value)}
    return "UNDEFINED", f"{et}: {msg}", {"exception": et, "args":
        (fieldType, value)}
```

Listing 9: Generated oracle failing predicate for TIME-4.

### 6.1.2 Bug Report Alignment

This section reports the qualitative alignment between LLM-generated oracles and the natural-language bug reports associated with the Defects4J subjects. This analysis observes whether the oracle logic reflects the failure conditions described in the bug report. Alignment was assessed by manually comparing each generated oracle against its corresponding bug report and classifying the result into one of three categories: *fully aligned*, *partially aligned*, or *not aligned*.

**Fully aligned.** An oracle is classified as *fully aligned* if it operationalises the core bug signature described in the bug report without omitting essential constraints. In these cases, the failure predicate expressed by the oracle directly mirrors the documented defect, either through an equivalent exception condition, a value-based predicate, or a state-based invariant.

LANG-39 provides a representative example of full alignment, as shown in Listing 10. Across all generated variants, the oracle predicates correctly target the string-replacement behaviour described in the bug report, detecting erroneous substitutions under the same conditions articulated in the natural-language description[16]:

*The following Test Case for replaceEach fails with a null pointer exception.*

The generated oracles do not miss relevant failure cases, indicating a close correspondence between the inferred bug signature and the documented behaviour.

```
FAIL_EXCEPTIONS = {
    "java.lang.NullPointerException": "null input passed to replaceEach
        method",
}
...
try:
    ret = CALL_MUT(receiver_or_class, text, searchList, replacementList,
        repeat, timeToLive)
except jpype.JException as e:
    et = e.getClass().getName() # JPype: Java exception class
    msg = str(e)
    ...
    if et in FAIL_EXCEPTIONS:
        return "FAILING", f"{et}: {FAIL_EXCEPTIONS[et]}", {"exception":
            et, "args": (text, searchList, replacementList)}
    return "UNDEFINED", f"{et}: {msg}", {"exception": et, "args": (text,
        searchList, replacementList)}
```

Listing 10: First generated oracle failing predicate for LANG-39.

**Partially aligned.** An oracle is considered *partially aligned* when it captures some aspect of the bug report but deviates in scope, precision, or completeness. Typical cases include oracles that: (i) detect the correct exception or condition but fail to enforce all reported guards, (ii) introduce overly broad predicates that may flag non-buggy behaviour, or (iii) rely on secondary symptoms rather than the primary bug signature described in the report.

CHART-12 exemplifies this category. The 2nd generated oracle for this subject correctly associates the defect with an invalid dataset state following method execution, consistent with the bug report's emphasis on listener registration. However, several variants reduce the failure predicate to a coarse state check that does not fully encode the listener-related invariant described in the report[17]:

> *When dataset is passed into constructor for MultiplePiePlot, the dataset is not wired to a listener, as it would be if setDataset is called.*

[16] https://issues.apache.org/jira/browse/LANG-552
[17] https://sourceforge.net/p/jfreechart/patches/213/

As a result, the oracle remains bug-relevant but does not precisely mirror the documented defect condition, as shown in Listing 11.

```
1 if ret is not None and len(ret.getItems()) == 0:
2     return "FAILING", "Returned legend items collection is empty
          unexpectedly.", {"ret": str(ret), "args": args}
```

Listing 11: 2nd generated oracle failing predicate for CHART-12.

**Not aligned.** Oracles are classified as *not aligned* when their failure predicates do not correspond to the bug report in a meaningful way. This includes cases where the oracle: (i) misinterprets the reported defect mechanism, or (ii) encodes assumptions not supported by the documentation or report.

MATH-53 illustrates this outcome. All generated oracles for this subject introduce failure predicates that are not grounded in the numerical or probabilistic conditions discussed in the bug report[18]:

> *Subtract includes an isNaN test and returns Complex.NaN if either complex argument isNaN; but add omits this test.*

Instead, the oracle logic focuses on generic execution outcomes that neither reflect the reported failure mode nor approximate its intended bug signature, as demonstrated in Listing 12. Consequently, these oracles do not meaningfully align with the natural-language description of the defect.

```
1 if (args[0].getReal() != args[0].getReal() or args[0].getImaginary() !=
      args[0].getImaginary() or
2     ret.getReal() != ret.getReal() or ret.getImaginary() !=
          ret.getImaginary()):
3     return "FAILING", "NaN value detected in either input or result",
          {"ret": str(ret), "args": args}
```

Listing 12: 5th generated oracle failing predicate for MATH-53.

Across all Defects4J oracle generations, qualitative inspection shows that the majority of generated oracles exhibit substantial alignment with the corresponding bug reports. As summarised in Table 6, 130 oracle instances fully capture the bug-signature described in the report, either by reproducing the relevant exception condition or by encoding an equivalent semantic predicate. A smaller but non-negligible portion of 47 oracles exhibit partial alignment, where some aspects of the reported defect are reflected but important constraints or conditions are missing. Finally, a limited number of 28 oracle instances show no meaningful alignment with the bug report, typically focusing on generic failure conditions or unrelated behaviours.

[18] https://issues.apache.org/jira/browse/MATH-618

| Bug report alignment | Count |
|---|---|
| Fully aligned | 130 |
| Partially aligned | 47 |
| Not aligned | 28 |
| **Total** | **205** |

Table 6: Distribution of oracle generation bug report alignment in the evaluated Defects4J target set.

| Defects4J subject | 1st gen. | 2nd gen. | 3rd gen. | 4th gen. | 5th gen. |
|---|---|---|---|---|---|
| Chart-12 | Partially | Partially | Fully | Partially | Not at all |
| Lang-22 | Not at all | Not at all | Partially | Partially | Not at all |
| Math-20 | Partially | Partially | Fully | Partially | Partially |
| Math-60 | Partially | Partially | Not at all | Partially | Not at all |

Table 7: Distribution of oracle bug report alignment per generation in the evaluated Defects4J target set where variance was observed.

While the majority of Defects4J subjects exhibited stable qualitative alignment across all five oracle generations, a small subset showed observable variance in how closely the generated oracles aligned with the corresponding bug reports. Specifically, variance across generations was observed for Chart-12, Lang-22, Math-20, and Math-60. In these cases, different oracle generations (gen.) for the same subject were classified into different alignment categories (fully aligned, partially aligned, or not aligned), despite being generated under identical prompting and configuration (see Table 7).

**Chart-12 bug report alignment variance.** Across the five oracle variants for Chart-12, the most frequently observed outcome is *partial* alignment (gen. 1, 2, and 4). In these generations, the oracle consistently anchors its bug signature on a `NullPointerException` and adds a secondary check that treats a `None` dataset argument as failing. This captures a generic robustness concern (null-handling) but does not operationalise the defect described in the bug report, which concerns an *incorrect dataset–listener relationship* (i.e., missing listener wiring for a *non-null* dataset). Hence, the oracle logic overlaps with plausible failure modes yet remains mis-scoped with respect to the reported symptom, leading to a *partially aligned* classification.

Deviations occur in two directions. Gen. 3 is the only variant that becomes *fully* aligned: instead of treating `NullPointerException` as the primary signature, it attempts to express the reported invariant by introducing an explicit listener-related predicate (e.g., querying listener state via a `getListeners`-like mechanism). This directly targets the bug report's condition; whether the dataset is correctly registered with the expected listener, and therefore matches the intended failure signature more closely than null-centric checks.

Conversely, gen. 5 drops further to *not at all* aligned. This oracle contains no meaningful

non-exception predicate and effectively reduces bug detection to "`NullPointerException` implies FAILING". Since the CHART-12 defect is typically observable without raising an exception (i.e., it manifests as an incorrect post-state/invariant violation rather than an immediate crash), this variant neither checks the dataset–listener condition nor distinguishes the bug-relevant state from benign executions, which motivates the *not at all* alignment classification.

**Lang-22 bug report alignment variance.** Across the five generations, the dominant outcome for LANG-22 is *not aligned* (gen. 1, 2, and 5). In these variants, the oracle does not operationalise the bug report's return-value specification; namely that `greatestCommonDivisor(Integer.MIN_VALUE, 2^k)` should yield the expected positive power-of-two result (e.g., `2` for `k=1`) rather than an incorrect value. Instead, the oracle shifts to a different failure theory: it treats `ArithmeticException` as the primary bug signature and, on the non-exception path, flags only a coarse condition such as `ret == 0`. This reasoning does not match the report narrative (which is about a specific incorrect *return value* region, not an exception-driven failure mode), and the `ret == 0` check is too weak and semantically disconnected from the described defect, leading to a *not at all* alignment classification. The remaining two generations (gen. 3 and 4) deviate from this dominant pattern by attempting to encode the return-value constraint described in the bug report, which improves alignment to *partially aligned*. Concretely, these variants introduce a predicate that resembles the reported scenario (involving `Integer.MIN_VALUE` and a power-of-two argument) and check whether the result violates the expected value constraint. However, the improvement remains partial because the predicate is implemented in an *example-bound* way: it relies on the concrete values shown in the bug report rather than deriving a general rule (e.g., recognising the $2^k$ structure and computing the expected result from inputs at runtime). As a result, gen. 3 and 4 capture the right *type* of oracle (return-value predicate) but with limited generality, whereas gen. 1, 2 and 5 remain misaligned because they default to an exception-centric signature and an unrelated return-value check.

**Math-20 bug report alignment variance.** For MATH-20, the dominant outcome across generations is *partial* bug-report alignment (gen. 1, 2, 4, and 5). In these variants, the oracle operationalises one concrete aspect of the report by treating a zero `checkFeasibleCount` (encoded as `args[6] == 0`) as a `FAILING` condition. This is judged *partially* aligned because it captures a bug-relevant *precondition* discussed in the report (a root-cause scenario), but it does not encode the report's *observable failure effect*: that the offspring/result violates the declared bounds. As a consequence, these oracles can also over-approximate the defect and yield false positives (they fail even when no out-of-bounds outcome occurs), and they omit the second case described in the report where out-of-bounds behaviour can occur despite `checkFeasibleCount` being non-zero.

Deviation occurs in gen. 3, which is classified as *not at all* aligned. Unlike the other generations, this oracle does not implement any concrete predicate for either (i) the `checkFeasibleCount == 0` condition or (ii) the actual out-of-bounds outcome. Instead,

the only bounds-related logic remains as commented or hypothetical checks, and the implementation therefore defaults to `PASSING` on the non-exception path. Due to the report's behaviour not being operationalised into an executable bug-signature predicate, gen. 3 is treated as not aligned with the bug report.

**Math-60 bug report alignment variance.** MATH-60 most frequently exhibits *partial* alignment (gen. 1, 4 and 5). These variants map convergence-related exceptions (e.g., `ConvergenceException` and broader `MathException`) to a failing verdict, which matches the bug report's primary symptom that extreme-value inputs should not trigger convergence failures. However, the non-exception predicates are limited to generic sanity checks (e.g., CDF within $[0, 1]$, not NaN, not $\pm\infty$) and do not encode the report's tail-behaviour requirement that extreme inputs (including infinities) should yield values close to 0 or 1. The resulting oracle therefore captures part of the bug signature (exception avoidance) but not the full behavioural intent for extreme-value returns.

The strongest deviation is observed in gen. 2 and 3, both assessed as *fully* aligned. These variants (i) treat convergence-related exceptions as failing outcomes and (ii) add an explicit tail-behaviour predicate: for inputs far below/above the mean (e.g., mean $\pm 20\cdot$ sd), the oracle expects the return value to be approximately 0 or 1, respectively. This directly reflects the bug report's emphasis on handling extreme values without failure while producing sensible tail probabilities.

Overall, this analysis shows that while LLM-generated oracles often recover bug-report –consistent failure predicates, both cross-generation variance and partial misalignment remain observable, motivating the complementary use of reference-oracle comparison and execution-based evaluation in the following sections.

### 6.1.3 EvoRepair Reference Oracle Equivalence

This section reports the qualitative comparison between each generated oracle and the corresponding EvoRepair oracle reference. In contrast to bug-report alignment (Section 6.1.2), this analysis focuses on whether the *operational bug-signature predicate* implemented by the generated oracle matches the predicate encoded by EvoRepair's instrumentation (i.e., the condition under which EvoRepair raises a `[Defects4J_BugReport_Violation]`). The comparison is therefore concerned with behavioural equivalence at the level of the failing predicate, rather than surface similarity in wording or structure. This equivalence is similarly measured by a three-level rubric: *fully equivalent*, *partially equivalent*, or *not equivalent*.

**Fully equivalent.** A generated oracle is classified as *fully equivalent* if it reproduces the EvoRepair failing predicate in a way that is behaviourally consistent across the relevant input region. Concretely, this requires that the oracle triggers `FAILING` on the same executions that would be marked as bug-report violations by EvoRepair (same exception signature, return-value predicate, or state predicate).

An example of this is TIME-14; the generated oracle encodes the same defect signature as the EvoRepair reference, as shown in Listing 13. It targets the same failure mode and applies the failure predicate in the same bug-relevant execution context. In qualitative inspection, the oracle's decision logic therefore mirrors the reference oracle at the level of what constitutes a violation, rather than only capturing loosely related symptoms, as demonstrated in Listing 14.

```
1 catch (org.joda.time.IllegalFieldValueException e) { //
      defects4j.instrumentation
2     if (this.getMonthOfYear() == 2 // defects4j.instrumentation
3             && this.getDayOfMonth() == 29
4             && e.getMessage().equals("Value 29 for dayOfMonth must be in
                  the range [1,28]")) { // defects4j.instrumentation
5         throw new RuntimeException("[Defects4J_BugReport_Violation]"); //
              defects4j.instrumentation
6 }
```

Listing 13: EvoRepair failing predicate for TIME-14.

```
1 except jpype.JException as e:
2     et = e.getClass().getName()
3     msg = str(e)
4
5     if et == "org.joda.time.IllegalFieldValueException" and args[0] > 0:
6         # Check if the MonthDay is set to February 29th
7         if receiver_or_class.getMonthOfYear() == 2 and
              receiver_or_class.getDayOfMonth() == 29:
8             return "FAILING", f"{et}: {FAIL_EXCEPTIONS[et]}", {"exception":
                  et, "args": args}
```

Listing 14: 4th gen. failing predicate for TIME-14.

**Partially equivalent.** A generated oracle is classified as *partially equivalent* if it captures a substantial portion of the EvoRepair failing predicate but deviates in at least one important dimension. Typical partial-equivalence patterns include: (i) targeting the correct failure symptom but failing to operationalise the precise reference condition (under-approximation), or (ii) encoding the correct signature only for a subset of the relevant input region (e.g., hard-coded exemplar values rather than a general predicate).

MATH-103 illustrates this case. Both the EvoRepair oracle and the generated oracle are exception-driven and signal failure when numerical convergence breaks down. However, the EvoRepair oracle explicitly targets `org.apache.commons.math.ConvergenceException` (see Listing 15), whereas the generated oracle maps the broader `org.apache.commons.math.MathException` to a failing verdict (see Listing 16. While this still captures convergence-related failures, it weakens equivalence by potentially classifying additional, non-bug-related numerical exceptions as failures.

```
} catch (org.apache.commons.math.ConvergenceException e) {
    throw new RuntimeException("[Defects4J_BugReport_Violation]");
}
```

Listing 15: EvoRepair failing predicate for MATH-103.

```
FAIL_EXCEPTIONS = {
    "org.apache.commons.math.MathException": "thrown when the algorithm
        fails to converge",
}
...
except jpype.JException as e:
    et = e.getClass().getName() # JPype: Java exception class
    msg = str(e)

    if et in FAIL_EXCEPTIONS:
        return "FAILING", f"{et}: {FAIL_EXCEPTIONS[et]}", {"exception":
            et, "args": args}
    return "UNDEFINED", f"{et}: {msg}", {"exception": et, "args": args}
```

Listing 16: $4^{\text{th}}$ gen. failing predicate for MATH-103.

**Not equivalent.** A generated oracle is classified as *not equivalent* if its failing predicate does not correspond to the EvoRepair reference predicate in any meaningful way. This includes cases where the oracle (i) replaces the reference predicate with an unrelated crash signature, or (ii) defaults to vacuous behaviour (e.g., always `PASSING` on the non-exception path) in a setting where the reference oracle is value- or state-driven.

For LANG-41, the generated oracle is not equivalent to the EvoRepair reference because the core predicate is structurally inverted. The EvoRepair oracle signals a failure when the returned string ends with a semicolon (see Listing 17), whereas the generated oracle instead expects a semicolon-suffixed type name and fails when this expectation is not met, as demonstrated in Listing 18. As a result, the oracle enforces a different semantic condition than the bug report specifies.

```
if (result.endsWith(";")) {
    throw new RuntimeException("[Defects4J_BugReport_Violation]");
}
```

Listing 17: EvoRepair failing predicate for LANG-41.

| Reference oracle equivalence | Count |
|---|---|
| Fully equivalent | 116 |
| Partially equivalent | 50 |
| Not equivalent | 39 |
| **Total** | **205** |

Table 8: Distribution of oracle generation reference equivalence in the evaluated Defects4J target set.

```
    if isinstance(args[0], list): # Assuming arrays are passed as lists in
        Python
        expected_class_name = f"{args[0][0].__class__.__name__};" #
            Expecting a semicolon for array types
        if ret != expected_class_name:
            return "FAILING", f"Expected '{expected_class_name}' but got
                '{ret}'", {"ret": str(ret), "args": args}
```

Listing 18: 2nd gen. failing predicate for Lang-41.

Across all Defects4J oracle generations, qualitative comparison against EvoRepair reference oracles shows that a majority of generated oracles exhibit substantial semantic equivalence. As summarised in Table 8, 116 oracle instances are fully equivalent to their corresponding EvoRepair oracles, capturing the same bug-signature predicate through equivalent exception conditions or value-based checks. A further 50 instances exhibit partial equivalence, where the generated oracle reflects the general failure mode but diverges in scope, specificity, or triggering conditions (e.g., by catching a broader or alternative exception). Finally, 39 oracle instances show no meaningful equivalence to the reference oracle, typically encoding an unrelated or inverted predicate that does not correspond to the benchmark-defined failure condition.

While the majority of Defects4J subjects exhibited stable levels of equivalence with the corresponding EvoRepair reference oracles across all five generations, a subset of subjects showed observable variance in equivalence classification across generations. Specifically, variation was observed for five subjects: Chart-12, Lang-22, Lang-46, Lang-55, and Math-20. For these subjects, different oracle generations for the same MuT were classified into different equivalence categories (fully equivalent, partially equivalent, or not equivalent), despite being generated under identical prompting conditions and model configuration.

Table 9 summarises the per-generation equivalence classifications for the affected subjects. The observed variance reflects differences in how consistently the generated oracles captured the precise failure predicates encoded in the EvoRepair reference oracles. These findings highlight the sensitivity of oracle generation to stochastic variation even under fixed prompts, and motivate closer inspection of cross-generation stability in oracle semantics.

| Defects4J subject | 1st gen. | 2nd gen. | 3rd gen. | 4th gen. | 5th gen. |
|---|---|---|---|---|---|
| Chart-12 | Partially | Partially | Fully | Partially | Not at all |
| Lang-22 | Not at all | Not at all | Partially | Partially | Not at all |
| Lang-46 | Partially | Not at all | Not at all | Not at all | Partially |
| Lang-55 | Partially | Partially | Partially | Not equivalent | Partially |
| Math-20 | Partially | Partially | Not at all | Partially | Partially |

Table 9: Distribution of oracle equivalence with EvoRepair reference oracle predicates across generations for Defects4J subjects where variance was observed.

**Chart-12 reference oracle equivalence variance.** For CHART-12, the most frequently observed equivalence category across generations is *partial equivalence*. In gen. 1, 2, and 4, the generated oracles primarily operationalise the bug as an exception-driven failure, treating a `NullPointerException` as the bug signature and additionally failing when the dataset argument is `null`. While these predicates capture a failure mode that can occur in practice, they do not reflect the core defect described in the EvoRepair reference oracle, which concerns missing listener registration for a non-null dataset. As a result, these generations partially align with the intended oracle semantics but mischaracterise the underlying failure condition.

A deviation from this dominant pattern occurs in gen. 3 which achieves *full equivalence*. This oracle is the only variant that encodes a state-based predicate consistent with the reference oracle by explicitly checking whether the returned dataset lacks registered listeners (e.g., via a listener-related accessor). This predicate directly captures the invariant violated by the bug and mirrors the EvoRepair oracle's intent.

The remaining deviation is observed in gen. 5, which is classified as *not equivalent*. In this case, the oracle reduces the bug signature entirely to the presence of a `NullPointerException` without any additional state-based checks. Since the reported defect manifests without requiring an exception and is observable through incorrect object state, this oracle fails to encode any meaningful aspect of the reference oracle predicate.

**Lang-22 reference oracle equivalence variance.** Across the five generations, the most frequent outcome for LANG-22 is *not equivalent* (gen. 1, 2, and 5) because these oracles do not encode the EvoRepair oracle's core return-value predicate for the bug-specific input region, i.e., the special-case constraint that for ($u =$ `Integer.MIN_VALUE`, $v = 2$) (and the swapped order) the method must return 2. Instead, these generations frame the bug primarily via an `ArithmeticException` signature and/or weak generic predicates such as `ret == 0` or `ret <= 0`, which do not match the structure or intent of the reference oracle and would misclassify fixed executions. This pattern deviates in gen. 3 and gen. 4, which are *partially equivalent* because they recover parts of the reference structure but fail to reproduce it fully. Gen. 3 identifies the correct triggering input case (including the swapped order), yet signals failure unconditionally for that case and thus omits the decisive return-value check (it would still "fail" even when the method

returns the correct value). Gen. 4 comes closer structurally by combining an input guard with a return-value constraint but encodes an incorrect expected value (failing on `ret != -1073741824` rather than checking whether the result equals 2). Overall, the variance reflects differences in whether the model operationalises the bug as a *specific guarded return-value requirement* (partial equivalence) or collapses it into a broader exception-/sanity-style signature (no equivalence).

**Lang-46 reference oracle equivalence variance.** For LANG-46, variance in reference oracle equivalence is primarily driven by how precisely the generated oracles encode the string-transformation constraint specified in the EvoRepair reference. Across the five generations, *non-equivalence* is the most frequently observed outcome (gen. 2, 3 and 4). Gen. 2 fails whenever a slash appears in the returned value, which incorrectly classifies correct behaviour (leaving `/` unchanged) as buggy. Gen. 3 and 4 invert the intended predicate entirely by requiring that slashes be escaped in the output (i.e., enforcing `ret == input.replace('/', '\/')`). This is the opposite of the EvoRepair specification which treats newly escaped slashes as the violation. As a result, these generations systematically fail to capture the intended bug-signature, illustrating how small deviations in return-value expectations can lead to complete divergence from the reference oracle. The remaining generations deviate toward *partial equivalence*. In these cases, the oracles correctly identify the core symptom of the bug, namely, that a forward slash (`/`) in the input should not be newly escaped to `\/` in the output and therefore flag executions where this transformation occurs as failing. However, these predicates remain coarser than the reference oracle, as they do not implement the reference oracle's additional guard that exempts inputs where the slash is already escaped (i.e., preceded by a backslash).

**Lang-55 reference oracle equivalence variance.** For LANG-55, *partial equivalence* is observed most frequently (gen. 1–3 and gen. 5). Across these generations, the oracles adopt the same high-level *stateful* structure as the EvoRepair reference by comparing a pre- and post-execution time value, but they operationalise a coarser invariant than the reference oracle: rather than checking that `stopTime` remains unchanged specifically when the stopwatch is in the `SUSPENDED` state, they typically enforce a generic uniformity rule (e.g., "time must not increase after `stop`") and omit the required state guard. In addition, several generations treat `IllegalStateException` as bug-indicative, which does not match the reference predicate. The main deviation occurs in gen. 4, which is classified as *not equivalent* because it inverts the intended direction of the check and flags decreasing time (`final_time < initial_time`) as failing; this neither captures the reference oracle's guarded invariant nor detects the same bug-report-violation condition.

**Math-20 reference oracle equivalence variance.** Across the five generations, *partial equivalence* is observed most frequently (gen. 1, 2, 4, and 5), because these oracles operationalise a bug-relevant *precondition* from the report; specifically, they flag executions where `checkFeasableCount == 0` (mapped to `args[6] == 0`) as `FAILING`. While

this captures a plausible root-cause signal, it does not reproduce the EvoRepair oracle's post-condition, which checks the *returned point* against the upper bound (i.e., a concrete return-value bounds predicate). Deviation occurs in gen. 3, which is classified as *not equivalent* because it does not implement either the gating condition or any executable bounds predicate corresponding to the reference oracle's violation check. As a result, no concrete predicate aligns with the reference oracle failing condition on the no-exception path.
Overall, the equivalence results indicate that LLM-generated oracles frequently recover the benchmark-defined bug-signature predicate, but remaining partial and non-equivalent cases, together with observed cross-generation variance, motivate validating oracle behaviour empirically on the EvoRepair-derived passing/failing pools in the following section.

### 6.1.4 Oracle Executability

**Usable passing and failing test pools.** For the Defects4J/EvoRepair setting, execution-based discrimination is evaluated on two post-processed pools per subject: passing invocations (non-bug-triggering) and failing invocations (bug-triggering, as labelled by EvoRepair's violation markers). The pool sizes reported in Table 10 reflect the *final usable* test material after Java-to-JPype translation, automated refinement, and removal of duplicates and non-executable fragments. Only subjects with non-zero post-processed passing and failing pools contribute to the execution-based measurements reported in the remainder of the Defects4J results; subjects without usable pools remain part of the qualitative results (bug-report alignment and reference-oracle comparison) but are excluded from empirical discrimination.

The following section thus reports *discrimination performance strictly on the available EvoRepair-derived tests*. As a prerequisite for interpreting these empirical outcomes, the generated oracles are first assessed for *executability* within the JPype-based harness. Executability is operationalised using the documented post-processing notes: an oracle instance is considered executable if it can be run end-to-end on the translated test scripts without the oracle itself crashing, with edits recorded where required to restore executability or remove harness-incompatible conditions. In total, this analysis covers all oracle instances for the 17 subjects for which EvoRepair passing/failing tests are available (i.e., $17 \times 5 = 85$ oracle instances).

**Executability distribution and post-processing severity.** Table 11 summarises the distribution of post-processing severity across the empirically evaluated oracle set. A total of 28/85 oracles (32.9%) execute out-of-the-box without edits. Another 17/85 (20.0%) become executable after *minor* edits (e.g., null-safety guards or small invocation corrections). The majority of the remaining instances require *moderate* edits (35/85, 41.2%), typically due to JPype-access incompatibilities (e.g., private-field access treated as Python attributes) or overly strict predicates that systematically crash or mishandle valid inputs under the harness. Finally, 5/85 oracles (5.9%) require *substantial* edits, reflecting cases where the generated oracle contains placeholder logic that must be replaced with a concrete, executable specification before any empirical execution is possible.

| Subject | Passing total | Failing total | Combined total |
|---|---|---|---|
| Chart-1 | 25 | 88 | 113 |
| Chart-5 | 99 | 97 | 196 |
| Lang-16 | 95 | 100 | 195 |
| Lang-39 | 91 | 90 | 181 |
| Lang-45 | 49 | 82 | 131 |
| Lang-46 | 47 | 90 | 137 |
| Lang-50a | 94 | 95 | 189 |
| Lang-51 | 96 | 4 | 100 |
| Lang-55 | 100 | 96 | 196 |
| Lang-61a | 86 | 12 | 98 |
| Lang-7 | 96 | 95 | 191 |
| Math-22 | 93 | 92 | 185 |
| Math-49 | 98 | 61 | 159 |
| Math-56 | 25 | 94 | 119 |
| Math-73 | 7 | 18 | 25 |
| Math-95 | 75 | 82 | 157 |
| Math-98 | 97 | 81 | 178 |
| ALL | 1273 | 1277 | 2550 |

Table 10: Final post-processed (executable/usable) passing and failing totals per subject after translation, refinement, and usability filtering.

Concretely, *all five* substantial edits arise from the five generations of MATH-56. In each variant, the generated oracle introduces a helper routine (`compute_expected_counts`) but leaves the core mapping logic as an explicit placeholder (either as a "fill in expected output" comment or a simplified sketch) rather than implementing the multidimensional index decomposition required by the MuT's specification, as illustrated in Listing 19. To make the oracle executable and semantically meaningful, the placeholder is replaced by an operational implementation derived from the library's intended mixed-radix semantics (row-major decomposition over the per-dimension sizes), as demonstrated in Listing 20. This intervention is necessary because, without a correct computation of the expected counts, the oracle cannot discriminate between correct and incorrect behaviour on the no-exception path. Empirically, it either degenerates into vacuous checks or misclassifies executions due to the missing predicate. In addition, several MATH-56 variants also require a small interoperability fix at the comparison boundary (converting the JPype `int[]` return value into a representation comparable to the Python-computed expected counts), ensuring that the repaired specification can actually be evaluated within the harness. For transparency and reproducibility, every manual modification applied during post-processing is explicitly preceded by a commented notifier within the oracle code, documenting the rationale for the change and clearly distinguishing original generated logic from subsequent corrective edits.

| **Executability status** | **Count** | **Share** |
|---|---|---|
| Executable out-of-the-box (no edits) | 28 | 32.9% |
| Executable after minor edits | 17 | 20.0% |
| Executable after moderate edits | 35 | 41.2% |
| Executable after substantial edits | 5 | 5.9% |
| **Total** | **85** | **100.0%** |

Table 11: Executability distribution for Defects4J oracle instances evaluated empirically (17 subjects × 5 generations).

```
def compute_expected_counts(receiver_or_class, index):
    # Logic to compute expected counts based on the index and the
        dimensions of the counter
    # This is a placeholder; actual implementation will depend on the
        specifics of the MultidimensionalCounter
    total_size = receiver_or_class.getSize()
    dimension = receiver_or_class.getDimension() # Assuming there's a
        method to get the dimension
    counts = [0] * dimension

    # Compute expected counts based on the index
    # This logic should reflect the expected behavior of the getCounts
        method
    # For example, if the counter is 2D with sizes 2 and 4, the expected
        counts for index 3 would be [0, 3]
    # Implement the logic here based on the actual behavior of the
        MultidimensionalCounter

    return counts # Return the computed expected counts
```

Listing 19: 4th gen. placeholder function for MATH-56.

| Edit type | Count |
|---|---|
| Type conversion fix | 16 |
| Overly strict predicate removal | 15 |
| Null-safety fix | 12 |
| API mismatch (JPype) | 9 |
| Invocation fix | 6 |
| Namespace / class resolution fix | 5 |
| Placeholder logic replacement | 5 |
| Helper / access workaround | 5 |
| Exception mapping fix | 5 |
| Predicate correction | 2 |

Table 12: Frequency of post-processing edit types across empirically evaluated Defects4J oracle instances.

```
def compute_expected_counts(receiver_or_class, index):
    total_size = receiver_or_class.getSize()
    if index < 0 or index >= total_size:
        return None
    sizes = list(receiver_or_class.getSizes())
    n = len(sizes)
    counts = [0] * n
    idx = int(index)
    for dim in range(n - 1, -1, -1):
        size_d = int(sizes[dim])
        counts[dim] = idx % size_d
        idx //= size_d
    return counts
```

Listing 20: Manually curated replacement function for the MATH-56 placeholder.

**Edit-type taxonomy.** Beyond severity, the post-processing notes support a coarse taxonomy of *edit types* that recur across subjects and generations. Table 12 reports the observed frequencies (counting an oracle instance once per edit type required). The most common interventions concern type interoperability and API access across the JPype boundary (e.g., explicit conversions, resolving missing attributes corresponding to private Java fields, and adjusting invocation patterns). A second cluster reflects specification quality issues in the generated predicates (overly strict conditions, incorrect exception mappings, or predicate inversions) which must be corrected to avoid systematic misclassification or runtime failures.

**Metrics and aggregation conventions.** Empirical oracle performance is evaluated using standard confusion-matrix–derived metrics computed over post-processed EvoRepair test pools. For methodological clarity, each generated oracle was evaluated against two disjoint test sets mined from EvoRepair's experimental artefacts: *passing tests*, consisting of inputs that do not trigger the reported defect, and *failing tests*, consisting of inputs that explicitly exercise the bug-report violation. This separation enables a direct assessment of each oracle's discrimination capability between correct and buggy program behaviour under identical experimental conditions. For each subject, Table 10 defines the final pool sizes and their class balance after translation, refinement, and noise removal. Let $n_{\text{pass}}$ denote the number of usable passing tests and $n_{\text{fail}}$ the number of usable failing tests for a given subject, such that $n_{\text{pass}} + n_{\text{fail}}$ equals the combined total reported in Table 10.

Under the harness' per-set evaluation convention, confusion counts are interpreted as follows. On the *passing* pool, True Positives (TP) correspond to passing tests correctly classified as `PASSING`, while False Negatives (FN) correspond to passing tests that are instead labelled `FAILING` or `UNDEFINED`. On the *failing* pool, True Negatives (TN) correspond to failing tests correctly classified as `FAILING`, while False Positives (FP) correspond to failing tests that are instead labelled `PASSING` or `UNDEFINED`. Consequently, the pool sizes are given by $n_{\text{pass}} = \text{TP} + \text{FN}$ and $n_{\text{fail}} = \text{TN} + \text{FP}$. Reporting $n_{\text{pass}}$ and $n_{\text{fail}}$ alongside performance metrics guards against misinterpretation when subjects are strongly imbalanced. In the present corpus, notable imbalances include Lang-51 (96:4), Lang-61a (86:12), and Math-73 (7:18), whereas several subjects remain close to balanced (e.g., Chart-5 99:97, Math-22 93:92).

For each oracle instance, we report passing-set accuracy ($\text{TN}/(\text{TN} + \text{FP})$), failing-set accuracy ($\text{TP}/(\text{TP} + \text{FN})$), overall accuracy, precision, recall, and F1-score computed over the combined passing and failing pools.

**Macro and micro averaging.** Performance aggregates are reported using both macro- and micro-averaging schemes. *Macro averages* are computed as unweighted means over subjects, treating each subject equally regardless of test-pool size. As such, macro averages characterise *typical subject-level behaviour* but may overweight subjects with very small passing or failing pools.

*Micro averages* are computed by first pooling confusion-matrix counts within a grouping (e.g., across all subjects, or within a project family) by summing $\sum \text{TP}$, $\sum \text{TN}$, $\sum \text{FP}$, and $\sum \text{FN}$, and then deriving the corresponding performance metrics from these totals (e.g., $\text{Acc.}_{\text{micro}} = (\sum \text{TP} + \sum \text{TN})/(\sum \text{TP} + \sum \text{TN} + \sum \text{FP} + \sum \text{FN})$). This aggregation weights subjects proportionally to the volume of available test data and therefore reflects *aggregate behaviour* under the EvoRepair-derived test distribution.

For completeness, dispersion estimates (standard deviations) are also reported for both macro and micro aggregates, computed over the underlying per-generation measurements within each grouping. Throughout the remainder of this section, macro averages are interpreted as describing typical oracle behaviour across subjects, whereas micro averages are used to contextualise overall performance under test-volume weighting.

**Overall discrimination performance (macro and micro).** Across the full Defects4J empirical corpus, the evaluation comprises 6081 true positives, 284 false negatives, 5704 true negatives, and 681 false positives across the combined passing and failing pools.

The *macro-averaged* results (mean over subjects) characterise typical subject-level behaviour by assigning equal weight to each subject, irrespective of the size of its test pool. Under this aggregation, the mean accuracy is 0.9335 with a standard deviation of 0.1053, while precision, recall, and F1 score average 0.9447, 0.9606, and 0.9430 respectively. The corresponding standard deviations indicate moderate dispersion across subjects, reflecting diverseness in oracle behaviour for different bugs and projects.

The *micro-averaged* results, by contrast, are computed from pooled confusion-matrix counts across all subject–generation instances, thereby weighting performance by the available EvoRepair-derived test volume. Under this aggregation, the overall accuracy is 0.9243, with precision, recall, and F1 score of 0.8993, 0.9554, and 0.9265 respectively. Dispersion estimates for the micro-aggregated metrics are computed over the underlying per-generation measurements and reflect the stability of performance across the full set of evaluated oracle executions. Together, macro and micro aggregates provide complementary views: the former summarises typical subject behaviour, while the latter reflects aggregate discrimination performance over the full empirical workload (see Table 20 in Appendix 8.4.

**Project-level discrimination performance.** When performance is aggregated at the project level using macro averages (i.e., means computed over subjects, treating each subject equally regardless of test volume), the overall accuracy across all empirically evaluated subjects is 0.9335 with a standard deviation of 0.1053. Precision and recall average 0.9447 and 0.9606, respectively, yielding a macro-averaged F1 score of 0.9430.

Disaggregating by Defects4J project family, CHART exhibits the strongest macro-level performance, with a mean accuracy of 0.9888 ($\sigma = 0.0112$), perfect precision, and high recall (0.9778), resulting in a macro F1 of 0.9882. The low dispersion indicates that performance is consistently close to saturation across the two evaluated CHART subjects. The MATH project family follows, with a mean accuracy of 0.9347 ($\sigma = 0.1012$), precision and recall averaging 0.9402 and 0.9588, respectively, and a macro F1 of 0.9400. Finally, LANG exhibits a lower mean accuracy of 0.9204 ($\sigma = 0.1151$), with comparatively higher dispersion across subjects, reflecting greater diverseness in discrimination behaviour within this project family. The corresponding macro precision (0.9354), recall (0.9580), and F1 (0.9348) nonetheless remain high in absolute terms. Table 21 in Appendix 8.5 reports the full set of macro-aggregated project metrics.

To complement the macro aggregates, micro-averaged project metrics are reported by pooling confusion counts across all evaluated subject–generation instances within each Defects4J project family and then computing the summary measures from these totals. Under this test-volume-weighted aggregation, CHART attains an accuracy of 0.9858 ($\sigma = 0.0224$) with precision 1.0000 ($\sigma = 0$), recall 0.9645 ($\sigma = 0.0444$), and F1 0.9819 ($\sigma = 0.0235$). For LANG, the micro-averaged accuracy is 0.9128 ($\sigma = 0.1507$), with precision 0.8850 ($\sigma = 0.1616$), recall 0.9610 ($\sigma = 0.0683$), and F1 0.9214 ($\sigma = 0.1087$).

For MATH, the corresponding micro averages are accuracy 0.9210 ($\sigma = 0.1012$), precision 0.8986 ($\sigma = 0.1338$), recall 0.9418 ($\sigma = 0.0922$), and F1 0.9197 ($\sigma = 0.0877$). The pooled micro aggregate across all projects is reported as accuracy 0.9243 ($\sigma = 0.1271$), precision 0.8993 ($\sigma = 0.1434$), recall 0.9554 ($\sigma = 0.0758$), and F1 0.9265 ($\sigma = 0.0965$), and is summarised alongside the per-project values in Table 22 (Appendix 8.5).

**Subject-level discrimination performance.** The subject-level macro aggregates (mean and standard deviation across the five generations) indicate that several subjects produced near-identical oracles across all generations (e.g., CHART-1, MATH-49, MATH-56, MATH-73, MATH-98), while others exhibit measurable variance across generations that is reflected primarily in passing-set false positives, failing-set false negatives, or both. Notable cases include: (i) LANG-50A, where 1st gen. achieves very high performance, but generations 2–5 collapse to markedly lower accuracy due to a sharp increase in false positives (passing tests misclassified as `FAILING`); (ii) LANG-46, where one generation shows a pronounced passing-set degradation (many false positives) while the failing-set behaviour remains strong (high recall), yielding a low overall accuracy for that generation but not for the remaining ones; (iii) MATH-95, where performance is dominated by false positives across generations (substantial fraction of passing tests flagged as `FAILING`), producing a comparatively low accuracy (0.7325) despite perfect recall on failing tests. Table 23 Appendix 8.6 reports the mean and standard deviation across generations for each subject.

When aggregating discrimination performance per subject using micro-averaging (i.e., pooling confusion-matrix counts across generations and weighting by test volume), the same subjects again exhibit near-ceiling behaviour, including CHART-1, MATH-49, MATH-56, MATH-73, and MATH-98, all of which achieve perfect micro-averaged accuracy, precision, recall, and F1. Subjects that displayed variance under macro aggregation remain identifiable under micro averaging, but with their performance now weighted by the size of the available test pools. In particular, LANG-50A retains a comparatively low micro-averaged accuracy (0.6085) driven by a high false-positive rate on passing tests despite perfect recall, while LANG-46 exhibits reduced micro-averaged precision (0.7591) alongside high recall, reflecting the influence of a single generation with substantial passing-set misclassification. Similarly, MATH-95 continues to show a lower micro-averaged accuracy (0.7325) due to systematic false positives, even though failing-set recall remains perfect. Table 24 Appendix 8.6 reports the full set of micro-averaged subject-level metrics together with their dispersion estimates.

**Per-generation discrimination performance.** To expose how performance changes across the five generations within each subject, Appendix 8.7 lists the confusion-matrix counts and derived metrics for every subject–generation pair (17 subjects $\times$ 5 generations).

Several concrete cross-generation deviations are visible in the per-generation table:

- **Stable "perfect" subjects.** For multiple subjects, all five generations achieve perfect discrimination on the available test pools (TP+TN equals the total number of tests, FP=FN=0, accuracy=1), including Chart-1, Math-49, Math-56, Math-73, and Math-98. In these cases, both passing-set accuracy and failing-set accuracy remain at 1.0 across generations, suggesting that the generated predicates (after post-processing) consistently capture a decisive behavioural signature that cleanly separates bug-triggering from non-triggering inputs under the EvoRepair-derived pools. Practically, these subjects form a useful "sanity check" set: if an implementation change were to disrupt these results, it would likely indicate issues in the harness, the JPype boundary, or the mined test pool rather than subtle oracle ambiguity.

- **FN-heavy behaviour on failing tests.** A recurring error mode is the presence of false negatives (FN) on the failing set, where the oracle fails to flag bug-triggering executions and instead returns `PASSING` (or, more generally, does not return `FAILING`). This pattern corresponds to an overly permissive oracle, a missing trigger predicate, or an incomplete encoding of the defect condition (e.g., checking a necessary-but-not-sufficient symptom, omitting a critical guard, or using an imprecise proxy that does not activate for all failing inputs). Quantitatively, this appears as reduced recall (TP/(TP+FN)) while the passing-set behaviour can remain stable (TN high, FP low). The most visible instances are the recall collapses described below (e.g., Chart-5, Lang-16, `Lang-39`) where the principal degradation is an increase in FN rather than a rise in FP.

- **Recall drops on failing tests with preserved passing behaviour.**
  - Chart-5 shows a generation-specific recall reduction in gen. 3 and gen. 5: TP decreases from 99 to 88 while FN increases from 0 to 11 (Acc.=0.943878, Rec.=0.888889), with TN and FP unchanged (TN=97, FP=0). This indicates missed failing cases rather than increased false alarms on passing tests. The oracle remains conservative on non-bug inputs but becomes insufficiently sensitive to the bug-triggering pattern for a subset of failing tests.
  - Lang-16 exhibits a larger recall degradation in gen. 3– 5: TP drops from 93 to 72 and FN increases from 2 to 23 (Acc.=0.882051, Rec.=0.757895), while TN remains 100 and FP remains 0. Again, the performance loss is concentrated in failing-set discrimination, consistent with a weakened or overly narrow failure predicate that fails to activate for many bug-triggering inputs.
  - Lang-39 shows a milder recall reduction in gen. 3 (TP: 86→80; FN: 5→11) but maintains strong passing-set behaviour (e.g., gen. 2: TN=90, FP=0). This suggests a partial loss of coverage over the failing region rather than a general increase in spurious failures.

- **FP-heavy behaviour on passing tests.** Lang-46 gen. 3 is a pronounced outlier: although failing-set accuracy remains high (TP=46, FN=1; Rec.=0.978723), passing-set discrimination collapses (TN=17, FP=73), yielding Acc.=0.459854 and Prec.=0.386555. All other generations for Lang-46 revert to near-perfect performance (TN=90, FP=0, FN=1; Acc.=0.992701). This sharp, isolated shift is most consistent with a generation-specific predicate polarity or gating error that causes the oracle to over-trigger on non-bug inputs (e.g., checking for a benign property as if it were bug-indicative, or inverting a return-value expectation).

- **Systematic false positives in a block of generations.** Lang-50a shows sharp variance across generations, driven almost entirely by false positives on the passing set. Gen. 1 performs strongly (TN=93, FP=2; Acc.=0.989418), whereas gen. 2–gen. 5 each exhibit TN=3 and FP=92 (Acc.=0.513228). Notably, TP=94 and FN=0 throughout, so failing-set accuracy remains 1.0 while passing-set accuracy collapses. This is indicative of a systematically over-aggressive failure condition in those generations: the oracle appears to implement a predicate that is satisfied by many non-bug inputs, effectively conflating normal behaviour with the defect signature.

- **Improvement to perfect discrimination in a later generation.** Lang-45 improves from partial failing-set discrimination (gen. 1–gen. 4: TP=42, FN=7; Rec.=0.857143) to perfect discrimination in gen. 5 (TP=49, FN=0; Acc.=1), while maintaining TN=82 and FP=0 across all generations. This pattern suggests that gen. 5 captures an additional missing constraint (or corrects a prior under-approximation) that closes the gap on the failing set without sacrificing specificity on the passing set.

- **Minor intra-subject variance from a small number of FP/FN.** Lang-55 remains strong overall but varies slightly: gen. 2 and 3 are perfect (Acc.=1), while gen. 1 has FN=3 (Rec.=0.97) and gen. 4 introduces FP=4 (Prec.=0.961538). This indicates small, generation-dependent shifts in how strictly the oracle gates the failing condition, where modest changes in predicate structure can trade sensitivity (recall) against specificity (1−FP rate) even when overall accuracy remains high.

Overall, the per-generation table shows that when performance drops occur, they tend to be *directional*: either (i) reduced failing-set accuracy via additional FNs, or (ii) reduced passing-set accuracy via additional FPs.

## 6.2 BugsInPy Generated Oracles

This section reports qualitative results for the BugsInPy/Tests4Py evaluation setting. As described in Chapter 4, the proposed pipeline generates executable Python-based oracle implementations using one-shot prompting. Consequently, this section focuses primarily on qualitative analysis of oracle structure, bug-report alignment, and reference oracle equivalence. The BugsInPy case study comprises **five** subjects in total. One subject,

| Subject | Method doc | Class/module doc | Helper docs |
|---|---|---|---|
| THEFUCK-1 | synthesised | docstrings | synthesised |
| COOKIECUTTER-3 | docstrings | docstrings | docstrings |
| PYSNOOPER-2 | synthesised | docstrings | synthesised |
| PYSNOOPER-3 | synthesised | docstrings | – (no helpers used) |
| YOUTUBE-DL-3 | synthesised | synthesised | synthesised |

Table 13: Documentation sources for the BugsInPy case-study subjects.

THEFUCK-1, is used as a *one-shot exemplar* and is therefore generated only once. The remaining four subjects (COOKIECUTTER-3, PYSNOOPER-2, PYSNOOPER-3, and YOUTUBE-DL-3) each admit **five** independently generated oracle variants to account for LLM non-determinism.

**Case-study subject selection and artefact availability.** The BugsInPy case study subjects were selected primarily on the basis of artefact availability rather than on any assumption of representativeness. In particular, BugsInPy fixed commits are not consistently accompanied by natural-language rationale (e.g., issue threads, commit messages, or release notes) that can be used as a traceable bug-report proxy. To preserve comparability with the Defects4J analysis, which relies heavily on benchmark-provided bug reports and reference oracles, subjects were therefore restricted to those for which (i) the buggy MuT could be recovered, and (ii) a usable bug report (or equivalent textual defect description) was available.

**Documentation gaps and manual synthesis.** Where benchmark artefacts did not include sufficient method-level or helper-level documentation to support oracle interpretation, contextual information was manually synthesised from the upstream GitHub repositories. This synthesis was limited to descriptive context (e.g., module intent, interface contracts, and helper-function behaviour) and was used to fill gaps in method documentation, module/class documentation, and helper documentation where relevant. Table 13 summarises where documentation was taken directly from the benchmark artefacts versus synthesised from repository context.

**Reference oracle styles in Tests4Py.** Inspection of the Tests4Py reference oracles reveals that the sample set exhibit a range of oracle styles that partially overlap with those observed in EvoRepair for Defects4J. In particular, two oracle categories recur directly: *exception-based* oracles and *return-value-based* predicate oracles. In addition, a third category emerges in Tests4Py that is not present in EvoRepair: *differential* oracles. Compared to the EvoRepair reference oracle, Tests4Py oracles are generally more diverse in structure and often embed richer control flow, test harness interaction, or comparison logic. As a result, categorisation is necessarily coarser, and fine-grained alignment between generated and reference oracles is discussed at a higher level of abstraction.

**Exception-based oracles.** Exception-based oracles define buggy behaviour in terms of an execution raising an error or violating an expected exception-related condition. This style mirrors the exception-based category observed in Defects4J. The subjects PySnooper-2 and PySnooper-3 fall into this category. In both cases, the Tests4Py reference oracle classifies executions as failing when a specific exception is raised (or, conversely, when an expected exception-related behaviour is absent). The bug signature is therefore expressed entirely through exception handling, without inspection of return values or post-state (see Listing 21.

```
# Oracle API call for PySnooper-2
PySnooperAPI(
            b"TypeError: __init__() got an unexpected keyword argument
                'custom_repr'"
        )
# Oracle API call for PySnooper-3
PySnooperAPI(b"NameError: name 'output_path' is not defined")
```

Listing 21: Tests4Py reference oracle API calls for PySnooper-2 and PySnooper-3.

**Return-value predicate oracles.** Return-value predicate oracles classify buggy behaviour by inspecting properties of the value produced by the MuT. This category is directly analogous to return-value-based EvoRepair oracles. The subject cookiecutter-3 exemplifies this pattern: the Tests4Py oracle evaluates the returned output against a semantic constraint derived from the bug report (in this case, a regular-expression-based condition). When the returned value violates this constraint, the oracle reports a failure (see Listing 22).

```
self.choice_pattern = re.compile(r"Choose from \d+(, \d)+ \(\d+(, \d)+\)")
...
if self.choice_pattern.search(output):
            return TestResult.FAILING
        return TestResult.PASSING
```

Listing 22: Tests4Py reference oracle failing predicate for cookiecutter-3.

**Differential oracles.** A third category, specific to the BugsInPy/Tests4Py setting, consists of *differential* oracles. Rather than expressing a bug signature directly in terms of exceptions, return values, or state predicates, these oracles compare the behaviour of two program variants, typically a buggy and a fixed implementation, and classify an execution as failing when the observed outputs differ. This style is observed for thefuck-1 (see Listing 23) and youtube-dl-3 (see Listing 24).

| Subject | Reference oracle category |
|---|---|
| THEFUCK-1 | Differential oracle |
| COOKIECUTTER-2 | Return-value-based oracle |
| PYSNOOPER-2 | Exception-based oracle |
| PYSNOOPER-3 | Exception-based oracle |
| YOUTUBE-DL-3 | Differential oracle |

Table 14: Observed reference oracle categories in the BugsInPy/Tests4Py case study.

```
expected = str(process.args[2])
...
result = str(process.stdout.decode("utf8"))
...
if result == expected:
    return TestResult.PASSING, ""
else:
    return TestResult.FAILING, f"Expected {expected}, but was {result}"
```

Listing 23: Tests4Py reference oracle differential predicate for THEFUCK-1.

```
result_buggy = unescapeHTML(query)
result_fixed = unescapeHTML_fixed(query)

if result_buggy != result_fixed:
    raise AssertionError("Input does not match the expected outcome!")
```

Listing 24: Tests4Py reference oracle differential predicate for YOUTUBE-DL-3.

Table 14 summarises the observed oracle categories for the BugsInPy subjects. These categories provide a lightweight but informative characterisation of the oracle styles present in the Tests4Py reference oracle. In the remainder of this section, they are used to structure the qualitative comparison between generated oracles, bug reports, and reference implementations, with particular attention to the challenges posed by differential oracle semantics and limited empirical test availability.

### 6.2.1 Exemplar Oracle Creation

Due to all subsequent BugsInPy oracle generations relying on one-shot conditioning, the initial exemplar prompt/oracle pair serves as a central reference artefact: its structure and failure logic may influence how later oracles express predicates, apply guards, and normalise outcomes. The subject THEFUCK-1 was chosen as the exemplar because its MuT is narrowly scoped, and its defect description admits a clear operationalisation as a *differential* specification (i.e., buggy and fixed executions are expected to disagree on a well-defined subset of inputs).
Together, these properties make THEFUCK-1 suitable for conditioning while remaining

representative of the command-oriented input/output style characteristic of the BugsInPy setting.

The exemplar oracle was generated in a *zero-shot* configuration using the universal system prompt (Appendix 29) and a fully instantiated user prompt derived from the prompt contract. The resulting oracle[19] was manually reviewed to confirm adherence to the Python oracle micro-skeleton: it invokes the MuT exactly once via the harness interface, maps observed outcomes to a stable three-valued verdict (`PASSING`, `FAILING`, `UNDEFINED`), and expresses failure conditions through runtime-evaluable predicates rather than hard-coded test cases or constants. Listing 25 illustrates the exemplar's dominant failure signal, expressed as a guarded exception signature (`IndexError`) that is only interpreted as bug-indicative under a pip-style error-message context.

In contrast to the Defects4J exemplar (TIME-4), a direct, line-by-line qualitative equivalence check between the generated exemplar oracle and the Tests4Py reference oracle for `thefuck-1` is not used as a comparison baseline because it implements a *differential* oracle, detecting failures via divergence between buggy and fixed executions. This is structurally and semantically incompatible with the single-execution, predicate-based oracle forms produced by the proposed pipeline. Instead, the exemplar oracle was first assessed qualitatively against the original bug report to confirm that its failure predicate targets the reported crash scenario; namely, the `IndexError` arising when malformed `pip` commands trigger an empty Regex match during command correction. Further behavioural validation was performed by executing the zero-shot exemplar oracle against the subject's Tests4Py *diversity tests*. Under this evaluation, the exemplar oracle achieved perfect agreement with the diversity-test ground-truth labels (100% accuracy over the PASSING/FAILING cases), providing empirical support that its failure predicate is consistent with the intended behavioural partition induced by the benchmark's test design. These results[20] highlight the corresponding predicate region used for classification. On this basis, the THEFUCK-1 zero-shot oracle was retained unchanged and used as the assistant-side conditioning exemplar for all subsequent one-shot generations in the BugsInPy case study.

[19] `https://github.com/liadanxa/LanguageGeneralisedTestOracleGenerationUsingLargeLanguageModels/tree/master/BugsInPy_experiment/prompts`

[20] `https://github.com/liadanxa/LanguageGeneralisedTestOracleGenerationUsingLargeLanguageModels/tree/master/BugsInPy_experiment/results`

```
FAIL_EXCEPTIONS = {
    FAIL_EXCEPTIONS = {
    IndexError: "Regex did not match pip output; empty match list indexed
        at [0] while extracting tokens",
}
...
try:
    ret = CALL_MUT(*args, **kwargs)

except Exception as e:
    etype = type(e)
    ename = etype.__name__
    msg = str(e)

    if etype in FAIL_EXCEPTIONS:
    ...
    return ( "FAILING",
           f"{ename}: {FAIL_EXCEPTIONS[etype]}",
           {"exception": repr(e), "script": pre_script,
           "output": pre_output, "args": args, "kwargs": kwargs,},
           )
```

Listing 25: Generated oracle failing predicate for THEFUCK-1.

### 6.2.2 Bug Report Alignment

This subsection summarises, for each BugsInPy subject, the extent to which the generated oracle predicates correspond to the failure phenomenon described in the associated bug report. As in the Defects4J setting, alignment is categorised as *fully aligned* (the generated oracle encodes the same failure condition as described in the bug report), *partially aligned* (the oracle captures a related but incomplete or over-approximate manifestation of the defect), or *not aligned* (the oracle targets behaviour irrelevant to the reported bug); these categories are reused here to ensure conceptual consistency across benchmarks. Cross-generation variance is reported descriptively using the alignment outcomes in Table 15.

**Cookiecutter-3.** The COOKIECUTTER-3 defect concerns an interaction failure in the prompt rendering/selection workflow (i.e., the buggy behaviour manifests *before* a stable user selection is returned). The generated oracles, however, predominantly encode a return-value predicate that checks whether the returned value is a member of the available options, thereby operationalising correctness at the level of the final selection rather than at the level of prompt construction or display. As a result, four of five generations are not aligned with the bug report, while gen. 4 exhibits partial alignment by attempting to reason about redundant prompt choices but (according to inspection) incorrectly treats the return value as containing prompt text and does not observe the relevant Click prompt output channel (see Listing 26).

This pattern indicates marked cross-generation variance in *framing* (prompt-output versus return-value reasoning), but comparatively stable non-alignment in the dominant generations.

```
if len(set(options)) != len(options):
    return (
        "FAILING",
        "Options provided are redundant",
        {"options": options},
    )
```

Listing 26: Partially aligned bug report failing predicate in 4th generated oracle for COOKIECUTTER-3.

**PySnooper-2 and Pysnooper-3.** The two PySnooper defects are naturally grouped because both bug reports describe a concrete runtime exception and the corresponding Tests4Py oracles are exception-driven. Across all available generations, the produced predicates consistently target the reported exception signature (`TypeError` for PYSNOOPER-2 and `NameError` for PYSNOOPER-3), yielding full alignment under the criterion used in this chapter (i.e., the same failure mode is operationalised as the oracle's `FAILING` condition). In contrast to COOKIECUTTER-3, cross-generation variance is minimal here: the failure predicate remains stable, and differences across generations (where present) are largely confined to superficial structural choices (e.g., message strings and normalisation details) rather than to the bug-signature condition itself.

**youtube-dl-3.** For YOUTUBE-DL-3, substantial cross-generation variation is observed in alignment with the bug report, which describes incorrect HTML entity unescaping for malformed entity patterns. Gen. 1, 2, and 5 are categorised as *not aligned*: these variants attempt to derive an expected output using helper transformation logic (e.g., `_htmlentity_transform`) under the implicit assumption that entities are well formed, and therefore fail to model malformed or truncated entity sequences that constitute the bug trigger (see Listing 27). By contrast, gen. 3 and 4 exhibit *partial alignment*. These oracles introduce broader mismatch-style predicates, typically gating on the presence of `&` in the input and flagging discrepancies in unescaping behaviour, which can incidentally detect malformed-entity cases (see Listing 28). However, these predicates remain underspecified with respect to the precise malformed-entity grammar described in the bug report, and thus do not encode the defect condition explicitly. Overall, the results indicate pronounced generation-level variance, with predicates ranging from overly constrained (assuming valid entity syntax) to overly general (detecting unescaping mismatches broadly), and no generation fully capturing the bug-report-defined failure condition.

```
if isinstance(pre_input, str):
    expected_output = re.sub(r'&([^;]+;)', lambda m:
        _htmlentity_transform(m.group(1)), pre_input)
    if ret != expected_output:
        return (
            "FAILING",
            ...
        )
```

Listing 27: Unaligned bug report failing predicate in 1st generated oracle for YOUTUBE-DL-3.

```
if isinstance(pre_input, str):
    # Example check for common HTML entities
    expected_output = pre_input.replace("&lt;", "<").replace("&gt;",
        ">").replace("&amp;", "&")
    if ret != expected_output:
        return (
            "FAILING",
            ...
        )
```

Listing 28: Partially aligned bug report failing predicate in 3rd generated oracle for YOUTUBE-DL-3.

**Cross-generation variance.** Taken together, these cases suggest that cross-generation variability is most visible when the defect narrative depends on an observation channel that is not trivially reducible to a single exception signature (e.g., prompt rendering in COOKIECUTTER-3 or malformed-entity semantics in YOUTUBE-DL-3). By contrast, for exception-described defects (PYSNOOPER-2/3), the generated predicates remain stable across generations and exhibit consistent bug-report alignment. Table 15 provides an overview of this.

### 6.2.3 Reference Oracle Equivalence

Following the Defects4J/EvoRepair setting, generated oracle to reference oracle comparison is reported using the same three-level equivalence rubric introduced earlier in this chapter. In brief, a generated oracle is treated as *fully equivalent* when its failing predicate matches the Tests4Py reference predicate in logic and observable trigger (up to superficial rephrasing); *partially equivalent* when it captures a strict subset, superset, or a proxy of the reference behaviour (thereby agreeing on some but not all relevant executions); and *not equivalent* when it targets a different failure phenomenon or omits the defining predicate of the reference oracle entirely. As in the Defects4J analysis, equivalence is assessed at the level of the predicate logic rather than surface-level implementation details (e.g., variable names, auxiliary formatting, or logging).

For YOUTUBE-DL-3, a direct equivalence assessment against the Tests4Py oracle is not performed. The Tests4Py reference is, as mentioned, *differential*, whereas the proposed pipeline produces *single-version* predicates over observed outcomes; the resulting oracle structures are therefore not comparable under the present rubric.

**Cookiecutter-3.** Across all five generations, the generated oracles are assessed as *not equivalent* to the Tests4Py reference oracle. In each case, the failing predicate inspects (or constrains) the MuT return value rather than checking the prompt-rendering string pattern that constitutes the Tests4Py bug signature (i.e., the presence of an incorrect token/encoding in the rendered prompt text). Consequently, the defining reference oracle predicate is not implemented, and no generation introduces an explicit search for the erroneous prompt string in the observable I/O.

Concretely, gen. 1–5 each apply an "option-membership" style check (or an analogous return-value constraint), which does not coincide with the reference oracle predicate that operates over prompt text (see Listing 26).

**PySnooper-2 and PySnooper-3.** All five generations of both the PYSNOOPER-2 and PYSNOOPER-3 oracles are assessed as *fully equivalent* to the Tests4Py reference oracles. Both reference oracles are exception-driven, and each generated oracle encodes the same bug signature by treating the relevant `TypeError` and `NameError` as the decisive failure signal under the harness' execution interface for PYSNOOPER-2 and PYSNOOPER-3, respectively. No cross-generation drift is observed in the core failing predicate; variations (where present) are confined to message text or ancillary bookkeeping that does not change the predicate's trigger condition.

The Tests4Py equivalence assessment indicates a clear split between (i) exception-signature subjects (PYSNOOPER-2, PYSNOOPER-3), for which the generated predicates consistently match the reference oracle across generations, and (ii) the prompt-rendering subject (COOKIECUTTER-3), for which the generated predicates target a different observable return value rather than prompt string and therefore do not reproduce the reference oracle's logic (see Table 15). The remaining BugsInPy subject (YOUTUBE-DL-3) is excluded from this equivalence analysis due to the differential nature of its Tests4Py oracle.

| Subject | Gen. | Bug report alignment | Reference oracle equivalence |
|---|---|---|---|
| cookiecutter-3 | 1 | Not at all | Not at all |
| cookiecutter-3 | 2 | Not at all | Not at all |
| cookiecutter-3 | 3 | Partially | Not at all |
| cookiecutter-3 | 4 | Not at all | Not at all |
| cookiecutter-3 | 5 | Not at all | Not at all |
| PySnooper-2 | 1 | Fully | Fully |
| PySnooper-2 | 2 | Fully | Fully |
| PySnooper-2 | 3 | Fully | Fully |
| PySnooper-2 | 4 | Fully | Fully |
| PySnooper-2 | 5 | Fully | Fully |
| PySnooper-3 | 1 | Fully | Fully |
| PySnooper-3 | 2 | Fully | Fully |
| PySnooper-3 | 3 | Fully | Fully |
| PySnooper-3 | 4 | Fully | Fully |
| youtube-dl-3 | 5 | Fully | Fully |
| youtube-dl-3 | 1 | Not at all | - |
| youtube-dl-3 | 2 | Not at all | - |
| youtube-dl-3 | 3 | Partially | - |
| youtube-dl-3 | 4 | Partially | - |
| youtube-dl-3 | 5 | Not at all | - |

Table 15: Tests4Py bug report alignment and reference oracle equivalence of generated oracles across BugsInPy subjects and generations.

Overall, the reported results characterise the artefacts produced by the pipeline in two complementary regimes: (i) a large Defects4J setting where both qualitative alignment (to bug reports and EvoRepair predicates) and execution-based discrimination metrics can be measured on mined passing/failing pools, and (ii) a smaller BugsInPy case study where qualitative traceability to bug reports and Tests4Py references is emphasised. The next chapter builds on these observations to evaluate the pipeline's effectiveness and limitations, interpreting them through the lens of the research questions introduced in Chapter 1, and assessing what the observed patterns imply for the feasibility, robustness, and limitations of LLM-based test oracle generation.

# 7 Evaluation

This chapter interprets the empirical and qualitative results presented in Chapter 6 and evaluates what they imply for the feasibility, effectiveness, and robustness of large language model (LLM) driven test oracle generation. Where Chapter 6 focused on *what was observed*, the purpose of this chapter is to assess *what these observations mean*: how well the generated oracles satisfy the stated research objectives, how convincingly they answer the research questions, and how robust the evidence is under variation in subjects, generations, and evaluation artefacts.

Concretely, this chapter evaluates oracle quality along three complementary dimensions that mirror the research questions introduced in Section 1.1:

- **Correctness and strength** of generated oracles relative to intended behaviour (**RQ1**),
- **Empirical fault-discrimination effectiveness** in an execution-based benchmark setting (**RQ2**), and
- **Cross-language behavioural consistency and divergence** between Java and Python oracle generation (**RQ3**).

Rather than treating these dimensions independently, the evaluation synthesises qualitative conformance analyses (bug-report alignment and reference oracle equivalence) with execution-based evidence, and relates both to observed generation-to-generation variance.

**Evaluation settings and scope.** As established in Chapters 5 and 6, the evaluation is conducted in two empirically distinct settings, which serve complementary roles in answering the research questions.

**Oracle quality dimensions.** To ensure consistent terminology across the thesis, oracle quality is assessed using the following dimensions throughout this chapter:

**Usability / executability:** Whether the oracle can be executed within the harness with reasonable post-processing effort (e.g., handling formatting, and converting types).

**Correctness:** Whether the oracle's verdict semantics correspond to intended behaviour: `FAILING` should align with the bug signature (i.e., executions that exhibit the defect), and `PASSING` should align with non-bug behaviour under the same specification assumptions.

**Strength:** How reliably the oracle separates bug-triggering from non-bug-triggering executions in an execution-based evaluation. Strong oracles should minimise both misses on failing tests and false alarms on passing tests.

**Robustness / generality:** Whether the oracle avoids overfitting to literal values in the bug report or documentation and behaves sensibly under input variation. In particular, robust oracles compute expectations from inputs and program context rather than relying on hard-coded constants, narrow special cases, or single exemplar traces.

These dimensions jointly operationalise the research questions: correctness and robustness primarily inform **RQ1**, discrimination informs **RQ2**, and the distribution of behaviours and failure modes across languages informs **RQ3**.

**Verdict interpretation and scoring rules.** Generated oracles return one of three verdicts, `PASSING`, `FAILING`, or `UNDEFINED`. For execution-based evaluation, "success" is defined relative to the expected test category:

- **Failing tests (bug-triggering executions):** `FAILING` is desirable; `PASSING` constitutes a miss; `UNDEFINED` is treated as inconclusive.
- **Passing tests (non-bug executions):** `PASSING` is desirable; `FAILING` constitutes a false alarm; `UNDEFINED` is treated as inconclusive.

**Variance and reproducibility lens.** To assess reproducibility and robustness beyond single-sample observations, each subject is evaluated across five independently generated oracle variants. Variance is interpreted as an empirical signal of both model irregularity and the stability of the inferred behavioural "signature" from the same input artefacts (method code, documentation, and bug context).

Stability is assessed using two complementary measures:

- **Agreement across variants (per subject and per test):** The extent to which the five oracle variants produce the same verdict on the same execution trace. High agreement indicates stable decision logic and robust output interpretation; low agreement suggests sensitivity to prompt ambiguity, parsing brittleness, or divergent inferred specifications.
- **Sensitivity to post-processing:** The extent to which outcomes change under small, reasonable post-processing modifications (e.g., numeric parsing, tolerant comparisons, or type normalising). This captures whether observed variance is driven by semantic disagreement or by brittle interpretation of harness outputs.

These variance analyses provide a reproducibility lens that complements raw accuracy: an oracle that is occasionally correct but unstable across variants may be less practically useful than a slightly weaker oracle with consistent, interpretable behaviour.

**Defects4J / EvoRepair.** The Defects4J/EvoRepair setting provides the main empirical basis for this thesis. Here, oracle quality is assessed using a combination of: (i) qualitative comparison against natural-language bug reports, (ii) qualitative equivalence analysis against EvoRepair reference oracle predicates, and (iii) execution-based discrimination on benchmark-derived passing and failing tests mined from EvoRepair's experimental artefacts. This setting enables direct measurement of oracle strength in terms of their ability to distinguish buggy from expected behaviour under controlled, reproducible conditions, and therefore forms the primary evidence base for addressing **RQ2**.

**BugsInPy / Tests4Py.** The BugsInPy/Tests4Py setting serves as a smaller-scale, complementary case study that evaluates the transferability of the proposed oracle-generation pipeline to a second programming language. Due to limited availability of uniform execution-based test artefacts and the presence of structurally different oracle styles (e.g., differential oracles), evaluation in this setting is primarily qualitative. Generated oracles are analysed with respect to bug-report alignment, reference oracle correspondence where comparable, and cross-generation variance. This setting is used primarily to inform **RQ1** and **RQ3**, rather than to provide large-scale quantitative discrimination claims.

**From results to research questions.** The remainder of this chapter is structured to reflect this dual evaluation strategy. Section 7.1 evaluates the Defects4J-generated oracles by interpreting the qualitative and empirical results through the lens of oracle correctness, strength, and robustness. Section 7.2 discusses the BugsInPy case study, focusing on cross-language oracle-generation behaviour and failure modes. Finally, Section 7.3 assesses threats to internal, external, and construct validity, situating the findings within the limitations of the chosen benchmarks, artefacts, and evaluation methodology.

## 7.1 Defects4J Generated Oracles

**Bug report alignment as an indicator of semantic correctness.** Bug report alignment provides a significant direct qualitative signal for assessing the *semantic correctness* dimension of oracle quality defined at the beginning of this chapter. An oracle that is fully aligned operationalises the same bug-signature predicate described in the natural-language report; partial alignment indicates under- or over-approximation of that predicate; and non-alignment reflects a failure to capture the reported defect mechanism altogether. As such, alignment outcomes speak directly to **RQ1**, which asks to what extent GPT-4o-Mini can generate *correct and strong* test oracles from method-level code and documentation.

Across the full Defects4J corpus (205 generated oracle instances), the dominant outcome is *full alignment* (130 instances), followed by *partial alignment* (47 instances), with a smaller tail of *not aligned* cases (28 instances). However, this aggregate picture masks substantial differences across reference oracle categories, which are informative for understanding where semantic correctness is reliably achieved and where systematic difficulties arise (see Table 16).

| Oracle type | Fully | Partially | Not at all | Total |
|---|---|---|---|---|
| Exception oracle | 85 | 5 | 0 | 90 |
| Guarded signature oracle | 32 | 8 | 0 | 40 |
| Return-value predicate oracle | 12 | 16 | 22 | 50 |
| Stateful predicate oracle | 1 | 18 | 6 | 25 |
| **Total** | **130** | **47** | **28** | **205** |

Table 16: Distribution of oracle generation bug report alignment by type category in the evaluated Defects4J target set.

**Exception-based oracles: consistently high semantic correctness.** Exception-based reference oracles exhibit the strongest bug-report alignment by a wide margin. Of the 90 generated oracle instances corresponding to exception-driven EvoRepair references, 85 are fully aligned and none are classified as not aligned. This concentration of full alignment indicates that GPT-4o-Mini is highly effective at inferring exception-centred bug signatures from method-level code and documentation. From an oracle quality perspective, these oracles tend to achieve high *semantic correctness* with relatively low ambiguity: the bug-report predicate is typically reducible to the presence (or absence) of a specific exception under identifiable conditions.

This outcome can be attributed to two reinforcing factors. First, exception signatures provide a discrete and observable failure channel that is explicitly represented in both code and bug reports, reducing interpretive uncertainty. Second, exception-based predicates align well with the execution-based oracle micro-skeleton used in the pipeline, which normalises exceptions as first-class oracle signals. The resulting behaviour suggests that, for exception-driven defects, GPT-4o-Mini reliably produces correct bug-signature predicates rather than superficial proxies, strongly supporting a positive answer to **RQ1** for this oracle category.

**Guarded signature oracles: high alignment with moderate predicate weakening.** Guarded reference oracles also demonstrate strong alignment outcomes, with 32 of 40 instances fully aligned and no instances falling into the not-aligned category. Compared to exception-based oracles, however, guarded predicates exhibit a higher proportion of partial alignment (8 instances), reflecting a recurring weakening of guard conditions rather than a complete misinterpretation of the defect.

Qualitatively, generated oracles in this category usually identify the correct failure mechanism (e.g., a decoding error, invalid format, or illegal state), but may under-specify the precise input preconditions under which the bug is intended to manifest. This pattern suggests that GPT-4o-Mini often captures the *core failure effect* but struggles to fully reconstruct compound predicates that require tight coupling between input guards and failure conditions. Nonetheless, because the failure logic remains bug-relevant and rarely degenerates into unrelated checks, guarded oracles still score highly on semantic correctness and provide further evidence that GPT-4o-Mini can infer structured bug signatures when documentation and code make the guard explicit.

**Return-value predicate oracles: mixed alignment and reduced discrimination strength.** Return-value-based reference oracles present a markedly different alignment profile. Of the 50 generated instances in this category, only 12 are fully aligned, while 16 are partially aligned and 22 are not aligned at all. This makes return-value predicates the largest contributor to non-aligned outcomes in the Defects4J corpus.

The underlying difficulty lies in the nature of return-value specifications: unlike exception signatures, return-value predicates often encode *semantic expectations* rather than discrete failure events. Bug reports in this category frequently describe incorrect numerical ranges, malformed strings, or subtle output constraints that require the oracle to infer a generalised postcondition rather than recognise a single observable symptom. Generated oracles often respond by introducing either overly weak checks (e.g., sanity conditions such as non-nullness or non-zero results) or overly specific, example-bound predicates that hard-code values mentioned in the bug report. Both behaviours undermine *semantic correctness* and *strength*, leading to partial or non-alignment classifications.

From the perspective of **RQ1**, these findings indicate that GPT-4o-Mini's ability to generate correct oracles from documentation is substantially less reliable when the bug signature must be expressed as a general return-value constraint rather than an exception or guarded failure. While some fully aligned instances demonstrate that correct inference is possible, the high rate of partial and non-alignment suggests a systematic challenge in this oracle category.

**State-based oracles: weakest alignment due to implicit invariants.** State-based reference oracles exhibit the weakest bug-report alignment overall. Out of 25 generated instances, only a single oracle is fully aligned, while 18 are partially aligned and 6 are not aligned. This distribution reflects the difficulty of reconstructing implicit object invariants from method-level documentation and local code context alone.

In many state-based defects, the bug report describes a violation of a relational or structural invariant (e.g., listener registration, internal consistency between fields) that is not directly observable through return values or exceptions. Generated oracles often approximate these defects via indirect proxies, such as null checks or generic post-state sanity conditions, which overlap with plausible failure modes but do not encode the intended invariant precisely. As a result, these oracles typically under-approximate or mischaracterise the bug signature, leading to partial alignment at best.

This outcome highlights an important boundary for **RQ1**: while GPT-4o-Mini can sometimes infer stateful predicates, semantic correctness degrades substantially when the defect requires reasoning about object relationships or invariants that are only implicitly documented. In such cases, the generated oracles lack the necessary contextual grounding to recover the full bug-report predicate.

**Overall assessment and implications for RQ1.** Taken together, the bug-report alignment results indicate a clear, category-dependent answer to **RQ1**. GPT-4o-Mini can generate *correct* test oracles from method-level code and natural-language documentation *to a substantial extent*, but this capability is unevenly distributed across oracle types.

| Oracle type | Fully | Partially | Not at all | Total |
|---|---|---|---|---|
| Exception oracle | 80 | 10 | 0 | 90 |
| Guarded signature oracle | 25 | 15 | 0 | 40 |
| Return-value predicate oracle | 10 | 13 | 27 | 50 |
| Stateful predicate oracle | 1 | 12 | 12 | 25 |
| **Total** | **116** | **50** | **39** | **205** |

Table 17: Distribution of oracle generation EvoRepair Reference oracle equivalence by type category in the evaluated Defects4J target set.

For exception-based and guarded defects, the majority of generated oracles are fully aligned, demonstrating strong semantic correctness and providing reliable bug-signature predicates. For return-value and state-based defects, alignment degrades sharply, with a large fraction of oracles either weakening or failing to capture the reported behaviour.

The overall consensus across all generated artefacts is therefore not that oracle generation is uniformly correct, but that it is *predictably effective* for certain classes of bug signatures and significantly less reliable for others.

**Reference oracle equivalence as an indicator of semantic and behavioural correctness.** While bug-report alignment assesses whether a generated oracle reflects the defect narrative, *reference oracle equivalence* evaluates a stricter criterion of oracle quality: whether the generated predicate matches the *behavioural failure condition* encoded by EvoRepair's reference oracle. Under the terminology defined at the beginning of this chapter, reference oracle equivalence is therefore a direct indicator of *semantic correctness* and, critically, of *discrimination strength*, since equivalence implies agreement on which executions should be classified as bug-triggering versus non-bug-triggering. As such, this analysis informs both **RQ1** (can correct oracles be generated?) and **RQ2** (how effective are they at separating buggy from fixed behaviour?).

Across all 205 generated oracle instances, 116 are fully equivalent to their EvoRepair counterparts, 50 are partially equivalent, and 39 are not equivalent. As with bug-report alignment, these aggregate numbers conceal strong category-dependent patterns that illuminate where GPT-4o-Mini succeeds in reconstructing executable bug-signature predicates and where it systematically diverges (see Table 17).

**Exception-based oracles: near-complete behavioural reconstruction.** Exception-based reference oracles again exhibit the strongest results. Of the 90 generated instances in this category, 80 are fully equivalent and none are classified as not equivalent. Compared to bug-report alignment, the shift from 85 fully aligned to 80 fully equivalent instances reflects a small but meaningful distinction: while most generated oracles correctly identify the relevant exception as bug-indicative, a subset weakens the predicate by catching a broader exception type or by omitting a critical contextual condition present in the EvoRepair reference.

Nevertheless, the absence of non-equivalent cases indicates that GPT-4o-Mini almost never substitutes an unrelated failure theory when dealing with exception-driven defects. From an oracle quality perspective, these results demonstrate strong *semantic correctness* and high *discrimination power*: the generated predicates agree with the reference oracle on the vast majority of executions. This consistency provides robust evidence in support of both **RQ1** and **RQ2** for exception-based bugs.

**Guarded signature oracles: correct intent with weakened precision.** Guarded reference oracles show a similar but slightly diminished pattern. Of the 40 instances, 25 are fully equivalent and 15 are partially equivalent, with no non-equivalent cases. The absence of complete divergence suggests that the model consistently recognises the *intended failure phenomenon*, but frequently relaxes or incompletely reconstructs the guard conditions that delimit the bug-relevant input space.

In practice, partial equivalence in this category often manifests as an oracle that triggers failure under the correct general conditions but either over-approximates the failure region (leading to false positives) or under-approximates it (missing some bug-triggering executions). While such oracles retain reasonable semantic correctness, their *strength* as discriminators is reduced compared to the EvoRepair reference. This pattern helps explain why some guarded subjects later exhibit generation-dependent false positives or false negatives in the execution-based evaluation, foreshadowing the results discussed under **RQ2**.

**Return-value predicate oracles: systematic divergence from reference semantics.** Return-value-based reference oracles present the weakest equivalence profile after state-based oracles. Of the 50 generated instances, only 10 are fully equivalent, while 13 are partially equivalent and a majority (27) are not equivalent. This indicates that, even when the bug-report narrative is partially captured, the generated oracle frequently fails to reproduce the precise executable predicate enforced by EvoRepair.

The underlying cause is that EvoRepair return-value predicates often encode *exact semantic constraints* like numeric bounds, or string-format invariants that must hold across a broad input region. Generated oracles tend either to (i) replace these constraints with coarse proxies (e.g., non-nullness, non-zero values), or (ii) hard-code exemplar values mentioned in the bug report, resulting in predicates that agree with the reference only on a narrow slice of executions. Both behaviours lead to non-equivalence by either over- or under-approximating the true failure condition.

From the perspective of **RQ1**, this reveals a limitation in GPT-4o-Mini's ability to infer *generalised postconditions* from documentation alone. From the perspective of **RQ2**, it explains why return-value-based subjects are more likely to exhibit false positives or false negatives in execution-based evaluation: the generated predicates do not reliably partition the input space in the same way as the reference oracle.

**State-based oracles: minimal equivalence due to implicit invariants.** State-based reference oracles exhibit the lowest equivalence overall. Of 25 instances, only a single oracle is fully equivalent, while 12 are partially equivalent and 12 are not equivalent. This sharp drop relative to bug-report alignment confirms that even when a generated oracle captures the *spirit* of a stateful defect, it rarely reconstructs the exact invariant enforced by the EvoRepair reference.

State-based EvoRepair oracles frequently rely on internal object relationships or invariants that are not explicitly surfaced in method-level documentation or return values. Generated oracles therefore tend to substitute indirect checks, such as nullability, size conditions, or generic state sanity checks, that do not coincide with the reference predicate. These substitutions may yield partial semantic overlap but almost never full behavioural equivalence.

This category thus marks a clear boundary for **RQ1**: GPT-4o-Mini struggles to infer precise state invariants from local context alone. It also highlights a key threat to discrimination effectiveness under **RQ2**, as such weakened predicates can both miss bug-triggering executions and spuriously flag correct behaviour.

**Qualitative synthesis of bug-report alignment and reference oracle equivalence.** Taken together, the bug-report alignment and EvoRepair reference oracle equivalence findings provide a coherent qualitative answer to **RQ1** and a clear lead-in to **RQ2**. In terms of *correctness (semantic conformance)*, GPT-4o-Mini most reliably produces bug-signature predicates that are both report-consistent and behaviourally equivalent to the benchmark references when the underlying oracle is *exception-based* or *guarded*: these categories offer an explicit, runtime-observable failure channel and comparatively unambiguous predicate structure, supporting robust reconstruction of the intended defect mechanism. By contrast, *return-value* and especially *stateful* predicates exhibit markedly weaker qualitative performance: generated oracles frequently under- or over-approximate the intended condition, substitute coarse proxies, or fail to recover implicit invariants, thereby reducing both *semantic correctness* and expected *strength*. Overall, the qualitative results indicate that GPT-4o-Mini can generate correct and meaningful oracles for a substantial subset of subjects, but that this capability is strongly conditioned on bug-signature observability and predicate complexity; moreover, the close correspondence between high equivalence rates and categories with clearer predicates suggests that reference oracle equivalence is likely to be a key explanatory factor for subsequent execution-based discrimination outcomes in **RQ2**.

**RQ2: overall effectiveness at discrimination.** Across the full Defects4J corpus evaluated on the available EvoRepair-derived passing/failing tests, the generated (and post-processed) oracles exhibit strong discrimination capability in aggregate, with performance characterised by high sensitivity to bug-triggering executions but a small, non-negligible tendency to over-trigger on non-bug inputs. Using the pooled confusion totals (True Positives (TP) = 6081, False Negatives (FN) = 284, True Negatives (TN) = 5704, False Positives (FP) = 681), the *failing-detection rate* (i.e., `FAILING` on failing tests) is $5704/(5704+681) = 0.8933$, while the complementary *miss rate* on failing

tests is 0.1067. On the passing pool, the *false-alarm rate* (i.e., `FAILING` or `UNDEFINED` on passing tests) is $284/(6081 + 284) = 0.0446$, corresponding to a passing-set correctness rate of 0.9554; overall accuracy on the combined pools is 0.9243. At the project level, these aggregate trends persist but with systematic shifts: CHART is near-ceiling (micro accuracy 0.9858, precision 1.0000), indicating a very low false-alarm tendency; MATH remains strong overall (micro accuracy 0.9210) but its aggregate precision is pulled down by subject-specific FP (most prominently MATH-95, which consistently flags many passing tests as `FAILING`); and LANG exhibits the greatest heterogeneity (micro accuracy 0.9128 with lower precision), driven by generation-dependent predicate polarity/gating failures that inflate false alarms in isolated cases (e.g., LANG-46 gen. 3) or across a block of generations (e.g., LANG-50A gen. 2–5). These slices jointly answer **RQ2**: under execution-based evaluation, GPT-4o-Mini–generated oracles can reliably separate buggy and fixed behaviour for many subjects, but their practical effectiveness is constrained by occasional over-aggressive failure predicates (false alarms) and, less frequently, under-approximation of the failure condition (missed failing tests), with the latter appearing as recall drops that preserve passing-set specificity (e.g., CHART-5, LANG-16).

**Oracle usability and engineering effort.** The practical utility of the proposed pipeline also depends on whether generated oracles are *usable artefacts*: i.e., whether they can be executed in the JPype-based harness with minimal engineering overhead, and whether any required post-processing primarily concerns superficial integration issues or deeper *semantic clarification* of the intended failure predicate. Accordingly, an assessment is conducted of (i) how often the oracle instances are executable without edits versus with small edits, (ii) the dominant *nature* of post-processing effort, and (iii) the implications for scalability and the necessity of a human-in-the-loop stage.

**Generated oracles executability distribution.** Across the 85 empirically evaluated oracle instances (17 subjects $\times$ 5 generations), 28 (32.9%) execute out-of-the-box without edits, and a further 17 (20.0%) become executable after *minor* edits. Thus, just over half of the evaluated oracles (45/85, 52.9%) are usable with no more than small post-processing. The remaining instances require *moderate* edits (35, 41.2%), while only 5 (5.9%) require *substantial* edits. Importantly, the substantial-edit cases are entirely concentrated in **Math-56**, where each generation contained explicit placeholder logic that prevented meaningful execution until replaced by a concrete computation of the expected result (as described in Section 6.1.4). This distribution indicates that, for most subjects, the pipeline yields executable oracles with manageable integration effort, but that a small number of targets can trigger qualitatively different engineering demands when the semantics of the method under test (MuT) require non-trivial derived computations not made explicit in the prompt context.

**Characterising post-processing effort.** To directly connect usability outcomes to the empirical results for **RQ2**, the edit types in Table 12 can be consolidated into three higher-level effort classes that explain *why* an oracle instance either (i) fails to execute end-to-end in the JPype-based harness, or (ii) executes but cannot be trusted to yield stable discrimination behaviour without targeted refinement. Importantly, each class is grounded in the observed edit-type taxonomy rather than being an independent re-labelling.

1. **Interface repairs (scaffolding compliance).** This class captures edits that reconcile the generated code with the pipeline's required oracle scaffold and Python execution environment, without materially changing the intended failure predicate. Concretely, it encompasses *namespace / class resolution fixes* (e.g., missing or incorrect imports and symbol resolution; 5 occurrences), *invocation fixes* (e.g., aligning call signatures with the harness contract; 6 occurrences), and those *null-safety fixes* that merely prevent avoidable crashes on valid inputs (12 occurrences). As such, interface repairs primarily alleviate incidental integration friction and are best interpreted as engineering overhead in the service of executability, rather than as evidence of semantic mismatch with the target defect.

2. **Runtime binding repairs (JPype interoperability).** A second, and empirically dominant, cluster concerns cross-language interoperability at the Python-Java boundary. This class encompasses *type conversion fixes* (16 occurrences), *API mismatch (JPype)* edits (9 occurrences), and *helper / access workarounds* (5 occurrences) required to resolve access patterns and overload / representation discrepancies introduced by the runtime binding. These interventions are necessary because the same oracle logic that appears coherent at the level of natural-language specification can still fail operationally when confronted with JPype's concrete typing, array representations, and member exposure rules. In terms of **RQ2**, runtime binding repairs are therefore preconditions for obtaining interpretable discrimination measurements, rather than changes that (intentionally) strengthen or weaken the oracle's semantic predicate.

3. **Semantic clarification (specification repair under underspecification).** Finally, a smaller but qualitatively more consequential class comprises edits that adjust, disambiguate, or complete the operationalised failure predicate itself, and therefore directly influence discrimination behaviour. This class encompasses *overly strict predicate removal* (15 occurrences), *exception mapping fixes* (5 occurrences), *predicate corrections* (2 occurrences), and the extreme case of *placeholder logic replacement* (5 occurrences). These interventions address situations where the generated predicate either (i) mis-specifies the defect signature (e.g., incorrect exception expectations), (ii) encodes an overly narrow or overly aggressive condition that would distort FAILING/PASSING separation on EvoRepair-derived tests, or (iii) remains incomplete as an executable specification. **Math-56** exemplifies the latter: across all five generations, the oracle sketch introduced a helper routine but omitted the core mixed-radix mapping logic, necessitating a human-authored

completion to produce a non-vacuous predicate capable of supporting **RQ2**-style discrimination. More generally, this class reflects the intrinsic reality that oracle construction from bug reports and partial documentation involves underdetermined semantics; consequently, limited human judgement is occasionally required to resolve ambiguity into a concrete, executable, benchmark-aligned failure predicate.

**Implications for scalability.** Two implications follow for pipeline scalability. First, the frequency profile suggests that end-to-end execution is *often* achievable with low-to-moderate overhead: a majority of oracle instances run with no more than minor edits, and moderate edits are dominated by predictable interoperability issues rather than wholesale redesign of the failure predicate. This supports the feasibility of scaling the approach to larger corpora provided that (i) recurring interface and JPype-binding repairs are progressively automated (e.g., systematic casting helpers, overload-resolution utilities, and stricter generation-time scaffolding constraints), and (ii) the post-processing protocol remains standardised and reproducible.

Second, the existence of a small but non-negligible tail of *semantic clarification* cases motivates a *human-in-the-loop* phase as a principled component of oracle engineering rather than an ad-hoc workaround. Oracle generation is, by design, grounded in partially underspecified natural-language artefacts (bug reports, informal documentation, and incomplete contextual cues); consequently, occasional manual clarification should be interpreted as a characteristic of the domain rather than a methodological flaw. Prior work on LLM-assisted test generation and real-world software-repair benchmarks similarly treats manual curation/verification and hybrid workflows as necessary to bridge the gap between generated artefacts and project-specific execution constraints, particularly when moving from syntactically plausible outputs to robust, executable, semantically correct test logic [29, 30, 47]. In this thesis, the key methodological constraint is therefore transparency: interventions are kept minimal, explicitly documented inline, and made reproducible, ensuring that the empirical **RQ2** conclusions reflect oracle behaviour under a clearly specified and auditable post-processing regime.

Overall, the usability analysis indicates that GPT-4o-Mini frequently produces executable oracles with limited engineering effort, but that interoperability and (rarer) semantic-completion cases create a practical boundary for full automation. This strengthens the **RQ2** interpretation: high discrimination performance is achievable in practice, yet sustained scalability requires (i) automation of routine interface/binding repairs, and (ii) a deliberate human-in-the-loop step for the minority of cases where the benchmark-relevant failure predicate cannot be fully recovered from underspecified artefacts alone.

**Failure-mode analysis through the pipeline contract.** To explain *why* discrimination succeeds or fails, the observed error patterns are interpreted through the pipeline contract: (i) the oracle must be *executable* under the harness, (ii) it must bind correctly across the JPype boundary, and (iii) it must operationalise a *failure predicate* whose semantics align with the intended defect signature.

Under this lens, the aggregate confusion totals reveal three qualitatively distinct regimes. The pooled confusion totals underlying these regimes are summarised in Table 26 in Appendix 8.8, which reports the full TP/TN/FP/FN counts and derived FP/FN rates aggregated across all evaluated subjects. Exception-driven oracles are typically robust under execution and binding, and exhibit low pooled false-alarm and miss rates ($\text{FP-rate}_{\text{pooled}} = 0.0093$, $\text{FN-rate}_{\text{pooled}} = 0.0253$), indicating that when the defect manifests as an explicit runtime signal, GPT-4o-Mini often reconstructs a usable failure predicate that supports **RQ2**-style discrimination. Guarded signature oracles similarly show no false alarms in the pooled view ($\text{FP-rate}_{\text{pooled}} = 0$) but a higher miss rate ($\text{FN-rate}_{\text{pooled}} = 0.0788$), consistent with a recurring tendency to under-approximate the guard that characterises bug-triggering inputs. By contrast, return-value predicate oracles are dominated by false alarms ($\text{FP-rate}_{\text{pooled}} = 0.3618$; $\text{FN-rate}_{\text{pooled}} = 0.0041$). This suggests that these strategies frequently encode predicates that are *too eager* and therefore conflate benign executions with bug-indicative behaviour. This directly undermines **RQ2** even when the oracle remains executable. Stateful predicate oracles sit between these extremes ($\text{FP-rate}_{\text{pooled}} = 0.0043$, $\text{FN-rate}_{\text{pooled}} = 0.1244$), reflecting the additional difficulty of observing and comparing pre/post state reliably under the harness contract. For completeness, the exact pooled rates cited in this paragraph are reproduced in Table 26 in Appendix 8.8.

**Exception-driven bugs: mis-specifications at the semantic boundary.** For exception-driven strategies, failure modes largely arise from *semantic mis-specification* rather than harness incompatibility: the oracle runs, but the captured runtime signal is either too weak or too broad. The subject-level profiles illustrate both directions. On the conservative side, FN-dominant behaviour (e.g., LANG-45 with FN-rate 0.1143; LANG-39 with FN-rate 0.0681) is consistent with *missing gating* or an incomplete mapping from the bug condition to the thrown exception (e.g., checking for an exception only under a narrow precondition, or expecting an exception subtype that does not fire for all failing inputs). On the aggressive side, FP-dominant behaviour in LANG-51 (FP-rate 0.75) and LANG-61A (FP-rate 0.0833) aligns with the classic over-generalisation failure mode: treating *any* exception as bug-indicative or failing to restrict the check to the correct exception class (wrong exception fully qualified name (FQN), wrong branch, or wrong failure context), which inflates false alarms on the passing pool. Notably, the project-level aggregation in Table 18 shows that exception oracles in CHART and MATH remain close to ceiling (mean FP-rate 0 in both; mean FN-rate 0.0222 in CHART and 0 in MATH), whereas LANG exhibits markedly higher mean FP-rate for exception strategies (0.2106). This disparity is compatible with the contract-level explanation that LANG defects often surface through API edge cases and diverse error signalling, increasing the likelihood that a plausible-but-incorrect exception predicate is synthesised (feeds **RQ1**) and then over-triggers under execution (degrades **RQ2**).

**Value/predicate bugs: over-eager predicates and weak semantic anchoring.** Return-value predicate strategies exhibit the most systematic failure mode: high false-alarm rates with near-zero miss rates, indicating that the generated predicate often fires on passing behaviour. This is visible both in pooled totals ($\text{FP-rate}_{\text{pooled}} = 0.3618$) and in

the subject profiles: Lang-50a (FP-rate 0.7789), Math-95 (FP-rate 0.5122), and Lang-46 (FP-rate 0.1622) are all FP-dominant with strong recall (FN-rate near 0). Qualitatively, these patterns are consistent with three recurring issues. First, *overly strict bounds* or brittle numeric expectations can label benign variability as failure (an oracle that encodes a boundary as necessary rather than indicative). Second, *checking unrelated properties* (a proxy predicate that correlates weakly with the defect signature) can trigger across broad regions of the passing pool. Third, *redundant or defaulting logic* can yield degenerate behaviour: a predicate that is always true under common inputs (thereby returning `FAILING` too often), or conversely returning `PASSING` by default when the oracle cannot derive a principled expected value (which would manifest as FN inflation, less prominent here). The project-level aggregation reinforces that this is not an isolated artefact: return-value predicate strategies have high mean FP-rate in both Lang (0.4706) and Math (0.2561), despite near-zero mean FN-rate, implying that the principal bottleneck for these oracle strategies is semantic anchoring rather than executability. Importantly, the existence of near-perfect cases (e.g., Math-56 and Math-56's corrected specification yielding FP=FN= 0) indicates that once the intended predicate is made operational and non-vacuous, value-based oracles *can* support **RQ2** discrimination; however, the frequent FP-dominant regime shows that, without strong defect signals, GPT-4o-Mini tends to over-approximate failure conditions, thereby satisfying **RQ1** only weakly and harming **RQ2**.

**Stateful bugs: observational gaps under the harness contract.** Stateful predicate strategies are constrained by what the harness can observe reliably and what the oracle can express without privileged access. The pooled results show low false alarms (FP-rate$_{\text{pooled}}$ = 0.0043) but a substantially elevated miss rate (FN-rate$_{\text{pooled}}$ = 0.1244), which aligns with the hypothesis that stateful defects are more often *under-specified* by an automatically generated oracle. This is stark in Math-22 (FN-rate 0.2473; FN-dominant), where the oracle appears unable to capture the full post-state condition that distinguishes failing from passing executions, even though it does not erroneously fail the passing pool. Several stateful challenges are consistent with this FN-heavy regime: (i) *pre/post state access* may be blocked or non-trivial (private fields, derived state, or state exposed only through side-effectful APIs), (ii) *side-effect observation* may require checking multiple fields or interactions rather than a single return value, and (iii) *object identity versus equality* can invalidate naive comparisons (reference equality in Java, floating-point tolerances, or collection ordering), causing an oracle either to avoid strong checks (leading to missed failures) or to implement checks that are too brittle (which would raise FPs; observed only marginally in Lang-55, FP-rate 0.0083). The contrast between Lang-55 (balanced, low FP/FN) and Math-22 (FN-dominant) further suggests that stateful success depends strongly on whether the relevant state is readily observable and whether the bug signature can be reduced to a small, stable post-condition, precisely the kinds of conditions that determine whether the pipeline contract can be satisfied without extensive human-in-the-loop instrumentation (feeds **RQ1**) and whether discrimination generalises beyond the mined tests (feeds **RQ2**).

| Project | Oracle type | #Subj. | Mean FP-rate | Mean FN-rate |
|---|---|---|---|---|
| Chart | Exception oracle | 2 | 0.0000 | 0.0222 |
| Lang | Exception oracle | 4 | 0.2106 | 0.0456 |
| Lang | Guarded signature oracle | 2 | 0.0000 | 0.0821 |
| Lang | Return-value predicate oracle | 2 | 0.4706 | 0.0106 |
| Lang | Stateful predicate oracle | 1 | 0.0083 | 0.0100 |
| Math | Exception oracle | 2 | 0.0000 | 0.0000 |
| Math | Guarded signature oracle | 1 | 0.0000 | 0.0000 |
| Math | Return-value predicate oracle | 2 | 0.2561 | 0.0000 |
| Math | Stateful predicate oracle | 1 | 0.0000 | 0.2473 |

Table 18: Project-level mean FP and FN rates by oracle category, covering all 17 evaluated Defects4J subjects.

**Variance across generations and its relationship to oracle equivalence.** A useful lens for interpreting the five-generation results for each subject is *cross-run stability*: the extent to which repeated oracle generations preserve (i) *executability* under the pipeline contract, (ii) the *bug-signature strategy* adopted (exception-driven, guarded signature, return-value predicate, or stateful predicate), and (iii) the resulting *verdict distributions* over the EvoRepair-derived passing/failing pools (excluding `UNDEFINED`, which is absent in this corpus). This matters directly for **RQ2**: even where mean discrimination is high, instability across generations implies that empirical effectiveness depends on which concrete oracle instance is selected, and therefore affects the practical reliability of the proposed pipeline.

At the level of *executability*, the post-processing evidence indicates that stability is generally high but not uniform: many subjects exhibit similar edit patterns across generations (often recurring interface or runtime-binding repairs), whereas a minority requires generation-specific interventions (e.g., an isolated JPype interoperability issue or a predicate that crashes under certain inputs). Importantly, this form of instability is largely *syntactic/runtime-binding variance*: it reflects friction at the interface between model-produced Python code and the harness' execution contract (imports, invocation details, JPype casting and member access), rather than a change in the intended failure predicate. Once the oracle is made executable via well-documented, reproducible edits, these cases typically converge to similar empirical behaviour across generations, which is consistent with the observation that many subjects have $\sigma_{\text{Acc}} = 0$ or near-zero despite requiring moderate post-processing.

By contrast, the most operationally meaningful form of non-determinism appears in *verdict distribution* shifts that persist after post-processing, i.e., cases where generations produce materially different TP/TN/FP/FN profiles over identical test pools. Quantitatively, the per-subject standard deviation (e.g., $\sigma_{\text{Acc}}$ in Appendix 8.6) acts as a compact proxy for this instability: subjects with $\sigma_{\text{Acc}} = 0$ are empirically invariant across generations (including near-ceiling cases such as Chart-1, Math-49, Math-

56, MATH-73, MATH-98, and also consistent non-perfect cases such as MATH-22 and MATH-95), whereas larger $\sigma_{\text{Acc}}$ indicates that at least one generation behaves qualitatively differently from the others. In the present corpus, this is most pronounced for LANG-46 and LANG-50A, each of which contains generations with a distinctly different error regime (a single severe FP outlier generation for LANG-46; a block of FP-heavy generations for LANG-50A).

This distinction enables a principled attribution of instability to either *runtime binding variance* or *substantive semantic variance*. Runtime binding variance typically manifests as "can it run?" differences (or minor behavioural artefacts that disappear once conversions and access paths are corrected) and is captured by the edit taxonomy categories such as *type conversion fixes* and *API mismatch (JPype)* repairs. Substantial semantic variance, by contrast, manifests as discrete jumps in FP or FN mass across generations even when all runs are executable, indicating that different generations encode meaningfully different failure predicates (e.g., polarity mismatches, missing gating, or over-broad error conditions). The directional error profiles visible in Appendix 8.7 align with this interpretation: FP-heavy collapses correspond to over-aggressive or inverted predicates on the passing pool, while FN-heavy collapses correspond to under-approximated bug triggers on the failing pool. Cross-run instability is not primarily a gradual metric drift; it is typically a *mode switch* between conservative and over-triggering decision rules.

Relating this to the earlier qualitative analyses, substantial semantic variance is also expected when *reference oracle equivalence* (and, to a lesser extent, *bug-report alignment*) varies across generations. Where the equivalence assessment indicates that some generations encode a non-equivalent predicate (for example, via inverted failure logic or an over-broad failure condition), the empirical evaluation tends to exhibit a corresponding verdict-distribution shift: LANG-46 is a canonical case where the qualitative diagnosis of a inverted failure predicate is mirrored by a one-generation passing-set collapse (TN= 17, FP= 73), which dominates $\sigma_{\text{Acc}}$ and $\sigma_{\text{Prec}}$. Conversely, where equivalence is stable across generations, or where post-processing interventions are *purely mechanical* (interface/runtime-binding repairs rather than semantic corrections), empirical variance is typically small. For example, CHART-5 shows only modest variability consistent with a limited FN increase in two generations (a recall reduction rather than a wholesale predicate change), and LANG-55 shows near-zero variability consistent with only a handful of FP/FN differences.

A final caveat is that bug-report alignment and reference oracle equivalence are informative but not interchangeable predictors of cross-run stability. High bug-report alignment may still yield unstable verdict distributions when the generated oracle is implemented with unsafe assumptions (e.g., overly strict bounds, insufficient gating around exceptions, or brittle checks that over-trigger on passing inputs), while low alignment may still appear empirically stable on a fixed test pool if the oracle nevertheless captures a discriminative signature for that pool. This motivates reporting both the *qualitative stability* of the encoded specification (alignment and equivalence across generations) and the *quantitative stability* of observed verdict distributions (standard deviations and per-generation confusion counts).

Together, these views distinguish subjects where cross-run non-determinism is mainly a matter of *surface form and runtime binding* from those where it reflects genuine *semantic divergence* in the operationalised bug predicate, thereby sharpening the interpretation of the pipeline's effectiveness for **RQ1** and its reliability for **RQ2**.

## 7.2 BugsInPy Generated Oracles

**Feasibility and oracle behaviours in Python: RQ1 and RQ3 support.** Under the BugsInPy/Tests4Py setting, the pipeline's primary feasibility claim for **RQ1** is that GPT-4o-Mini can synthesise *micro-skeleton-compliant* Python oracles that are executable end-to-end within a single-language harness, and that express a non-trivial, runtime-evaluable failure predicate derived from available textual artefacts. Qualitatively, this feasibility is supported for the majority of subjects: the generated artefacts generally adhere to the expected three-valued verdict contract (`PASSING`, `FAILING`, `UNDEFINED`), implement a single-run predicate rather than hard-coding test cases, and (where the defect signature is explicit) adopt a plausible *bug-signature strategy* that mirrors the narrative of the bug report. In terms of observed strategy types, the case study largely reproduces the same high-level split as Defects4J: exception-driven predicates dominate where the defect manifests as a concrete runtime error (PySnooper-2/3), while value/predicate-driven strategies arise where the reported failure concerns output conformance (cookiecutter-3) or transformation semantics (youtube-dl-3). A notable divergence from Defects4J is not the strategy taxonomy itself, but the presence of *differential* reference oracles in Tests4Py (e.g., thefuck-1, youtube-dl-3), which are structurally mismatched to the pipeline's single-execution oracle form; this mismatch becomes an important boundary condition for what can be claimed under **RQ3** when comparing across languages and benchmark conventions.

**Executable micro-skeleton compliance and the nature of "Python-side" effort.** In contrast to the Defects4J setting, the BugsInPy instantiation does not require cross-language execution and therefore avoids integration complexity by design. As a result, oracle executability in this setting is primarily a function of consistent scaffolding and environment assumptions rather than cross-runtime interaction. Accordingly, the BugsInPy component contributes to **RQ1** only at an architectural level, suggesting that single-language oracle execution under a micro-skeleton contract is feasible in principle, while leaving empirical executability and performance outside the scope of this evaluation.

**Alignment with bug reports and available Tests4Py oracles.** The bug-report alignment results indicate a sharp split between subjects with *explicit exception signatures* and those whose defect narratives encode more implicit semantic constraints. For PySnooper-2 and PySnooper-3, the generated oracles are consistently *fully aligned* across all generations: they operationalise the reported bug signature as a decisive exception condition (`TypeError` and `NameError`, respectively), which matches both the defect description and the Tests4Py reference oracle intent. In these cases, alignment and reference oracle equivalence coincide, providing a clean qualitative counterpart to the Defects4J observation that exception-driven defects are the most reliably reconstructed category.

In contrast, COOKIECUTTER-3 is systematically misaligned: most generations implement a return-value membership constraint over the MuT output, whereas the report and Tests4Py oracle intent concern prompt rendering/interaction behaviour observed through the prompt/output channel rather than the final selection return value. This is an instance of a recurring failure mode for **RQ1**: when the correct bug signature depends on a non-return observation channel (stdout text, UI/prompt formatting, or intermediate state), the model may default to "check the return value" even when that is not where the defect is expressed. Finally, YOUTUBE-DL-3 exhibits pronounced cross-generation drift: the defect narrative concerns malformed HTML entity unescaping, but the generated predicates oscillate between (i) overly constrained checks that implicitly assume well-formed entities and therefore miss the reported trigger region, and (ii) broader mismatch checks that are only *partially aligned* because they detect divergence in unescaping behaviour without encoding the specific malformed-entity grammar described in the report. These subject-level outcomes collectively reinforce the Defects4J pattern that high-quality qualitative performance is strongly conditioned on how explicitly the bug signature is expressed in the available artefacts.

**Systematic gaps surfaced by the case study.** Across subjects, three systematic gaps are visible that help explain why qualitative performance degrades outside the exception-driven regime. First, *documentation incompleteness* materially constrains the model's ability to infer intended behaviour: where method-level and helper-level contracts are missing and must be synthesised, the model has fewer anchors for deriving a correct predicate, increasing the likelihood of proxy checks that are plausible but mis-targeted. Second, *noisy or underspecified bug reports* lead to predicate under-determination: the model may capture a symptom (e.g., "unescaping mismatch") without capturing the defining condition that distinguishes bug-triggering inputs from normal inputs (e.g., malformed/truncated entity sequences). Third, the model shows a tendency to *mirror observed outputs rather than intended behaviour* when the defect is naturally described as a transformation constraint; this manifests as ad hoc string replacements or narrow example-based rules that may detect some failures but do not encode the general bug predicate. These gaps explain why COOKIECUTTER-3 remains non-equivalent to the Tests4Py oracle despite multiple generations, and why YOUTUBE-DL-3 yields only partial alignment: in both cases, the limiting factor is not micro-skeleton compliance but predicate grounding and correct selection of the observation channel.

**Cross-language comparison and how it supports RQ3.** The BugsInPy case study provides additional qualitative support for **RQ3** by making it possible to separate behaviours that are plausibly *model-level* (and therefore expected to transfer across languages) from behaviours that are *language- or runtime-dependent*. At a model level, the same high-level strategy is observed in both settings: when the defect is described as a concrete runtime error, GPT-4o-Mini tends to encode the bug signature as an *exception-driven* predicate; when the defect is defined through implicit invariants, interaction traces, or non-local transformation constraints, the model more frequently produces *weaker or mis-scoped semantic predicates* and exhibits higher prompt- and generation-sensitivity. This supports the interpretation that cross-generation non-determinism primarily affects

cases where the oracle must reconstruct a semantic predicate, whereas exception-driven signatures are comparatively stable.

At the same time, the cross-language comparison highlights that the *sources of failure* differ materially between Java and Python once the predicate intent has been chosen. In Defects4J, a substantial portion of the observed friction stems from runtime embedding and binding: correct intent can still yield a non-executable oracle due to integration constraints, and some post-processing effort is therefore attributable to interface compliance rather than predicate semantics. In BugsInPy, the absence of cross-language execution removes this class of integration concerns by construction, shifting practical fragility towards project-level assumptions and execution context. This distinction is important for **RQ3**: it indicates that some barriers to language-generalised oracle generation are not inherent to oracle reasoning, but instead arise from the mechanics of crossing runtime boundaries and normalising observations.

A further language-dependent contrast concerns how precisely failure conditions can be represented. In Java-centric oracles, exception identity and method binding are often expressed in a highly specific manner (e.g., fully qualified exception types and signature-consistent call sites), which encourages narrow predicates but can be brittle under interop. In Python-centric oracles, dynamic typing and broader exception hierarchies make it easier to write syntactically permissive checks, but also make it harder to enforce a narrow failure boundary without strong prompt constraints, increasing the risk of over-generalised catch patterns. Taken together, these observations refine **RQ3** by distinguishing (i) cross-language consistencies in how bug signatures are *selected* from (ii) language-dependent constraints that affect how those signatures can be *operationalised* and executed.

**What can and cannot be claimed from BugsInPy under RQ3.** Given that Defects4J is evaluated empirically over mined passing/failing pools whereas BugsInPy is primarily assessed qualitatively (with selective execution evidence only where comparable), the cross-language conclusions must be delimited. The BugsInPy results can support claims about *feasibility of generating executable, micro-skeleton-compliant Python oracles* and about *qualitative tendencies in predicate selection* (e.g., exception preference; mis-targeting of observation channels), but they do not license strong claims about *comparative discrimination effectiveness* across languages, because the evaluation modalities and test-pool structures differ and the BugsInPy sample size is small. In particular, the presence of *differential* Tests4Py reference oracles highlights an evaluation-contract mismatch: a pipeline that generates single-version predicates cannot be expected to reproduce a reference oracle that classifies by buggy-versus-fixed divergence without additional instrumentation. Consequently, the BugsInPy case study should be interpreted as a complementary "case study lens" for **RQ1** and **RQ3**: it corroborates the same core strengths and weaknesses observed in Defects4J at the level of bug-signature strategy and qualitative alignment, while making explicit which practical obstacles are intrinsic to the language/runtime embedding versus intrinsic to oracle construction from underspecified natural-language artefacts.

## 7.3 Threats to Validity

**Threats to validity.** This section summarises the principal threats to validity of the empirical and qualitative evaluations. The discussion follows standard software engineering reporting practice by distinguishing construct, internal, external, and conclusion validity [9]; and additionally highlights the residual risk of benchmark contamination.

**Construct validity.** A first construct threat is the use of EvoRepair-derived *passing* versus *failing* test pools as a proxy for *non-bug-triggering* versus *bug-triggering* behaviour. Although EvoRepair's labels provide a pragmatic and reproducible partition aligned with benchmark conventions, the resulting test pools may not fully characterise the behavioural boundary of the defect region (e.g., they may under-sample corner cases or reflect generator biases). This threat is mitigated by reporting pool sizes and class imbalance per subject and by complementing execution-based discrimination (**RQ2**) with specification-level analyses (bug-report alignment and reference oracle equivalence) that do not depend on test distribution assumptions; nevertheless, the evaluation remains conditional on the representativeness of the mined pools.

**Internal validity.** A primary internal validity threat is that *manual post-processing* could introduce bias by steering generated oracles towards improved executability or behaviour, potentially inflating discrimination performance. This is mitigated by (i) operationalising executability explicitly (severity levels and edit taxonomy), (ii) documenting every intervention inline in the oracle artefacts, and (iii) distinguishing repairs that are plausibly *mechanical* (interface/runtime binding) from those that require *semantic clarification* (e.g., placeholder completion for MATH-56). However, the possibility remains that even small semantic choices can affect borderline cases, and thus post-processing constitutes an explicit limitation on fully automated scalability. A second internal threat is that *test mining, translation, and refinement* errors may distort measured oracle performance (e.g., by mis-labelling, altering inputs, or introducing spurious failures). This is mitigated by filtering noisy tests, reporting final pool sizes, and using both macro- and micro-averaging to reduce the impact of small, error-prone pools; nonetheless, residual translation artefacts cannot be fully excluded. A third threat concerns *harness and runtime environment differences* (JVM/JDK versions, dependency resolution, library configuration), which can change exception behaviour, numerical edge cases, or I/O formatting. This is mitigated by running all experiments in a controlled environment and by treating harness failures as part of the executability analysis rather than silently discarding them, but cross-environment reproducibility remains a limitation that future replications should validate. Finally, the choice of *one-shot exemplars* (TIME-4 and THEFUCK-1) may steer oracle style and bug-signature strategy across subsequent generations. This is mitigated by explicitly framing exemplars as structural conditioning rather than semantic templates, and by evaluating multiple independent generations; nevertheless, exemplar-induced stylistic bias remains a plausible contributor to cross-run uniformity, especially in the BugsInPy case study.

**External validity.** External validity is constrained by the subject selection and benchmark scope. On Defects4J, the evaluation covers only those subjects for which usable EvoRepair passing/failing pools were available and successfully translated, and is further limited to specific project families (CHART, LANG, MATH). This is mitigated by reporting results per project and per subject, but generalisation to other Defects4J projects, other bug types, or other distributions of defect signatures remains uncertain. In BugsInPy, the sample size is small and the evaluation is primarily qualitative, which limits generalisability and prohibits strong population-level performance claims. This is mitigated by positioning BugsInPy as a case study lens for **RQ1**/**RQ3** rather than as an empirical counterpart to the Defects4J **RQ2** evaluation. More broadly, generalisation beyond Java and Python to other languages/runtimes depends on the availability and fidelity of adapters and on runtime binding semantics; the present results therefore demonstrate feasibility under two concrete embeddings rather than universal applicability.

**Conclusion validity.** A key conclusion validity threat is variance induced by *LLM non-determinism*: even with fixed prompting and repeated generations, different oracle predicates can emerge, and performance may hinge on a small number of semantically divergent variants. This is mitigated by using five generations per subject, reporting dispersion (standard deviations), and analysing generation-level variance patterns; however, the observed outliers (e.g., FP-heavy blocks) indicate that stability cannot be assumed from single samples. A second threat concerns *reporting and aggregation choices*, particularly the use of macro versus micro averages. This is mitigated by reporting both averaging schemes and by interpreting them explicitly, but alternative aggregation choices could change the apparent ranking of projects or oracle categories, especially under strong class imbalance.

**Data leakage and benchmark contamination.** Because Defects4J, EvoRepair, BugsInPy, and Tests4Py are public, there is an inherent risk that the LLM has been exposed to benchmark artefacts or closely related descriptions during pre-training, potentially inflating apparent performance. The study design mitigates (but cannot eliminate) this risk by focusing on *behavioural* outcomes under execution (**RQ2**), by evaluating multiple independently generated oracles per subject, and by analysing failures and mis-specifications that would be unlikely under simple memorisation. Nonetheless, contamination cannot be ruled out a priori, and the results should be interpreted as evidence of practical capability under realistic public-benchmark conditions rather than as a guarantee of identical performance on fully novel, private industrial defects.

Overall, the evaluation indicates that GPT-4o-Mini can produce correct and practically useful test oracles for a substantial subset of defects, with the strongest evidence in exception- and guard-driven settings and clear limitations for stateful and return-value specifications. These findings therefore support the thesis that LLM-driven oracle generation is feasible and often effective, but that robustness and scalability depend on predictable post-processing and careful treatment of under-specified behavioural predicates.

# 8 Conclusion & Future Work

This thesis examined whether large language models can synthesise executable and discriminative test oracles from method-level artefacts under a language-generalised prompt contract. The proposed pipeline renders subject prompts from method under test (MuT) source, available documentation, and bug-report-style defect descriptions, and produces Python-based oracle implementations that adhere to a stable three-valued verdict interface (`PASSING`, `FAILING`, `UNDEFINED`). Defects4J/EvoRepair provided an execution-based setting with mined passing/failing pools, while BugsInPy/Tests4Py served as a complementary case study emphasising qualitative traceability and cross-run variance, thereby grounding the answers to **RQ1**–**RQ3**.

The work contributes (i) a unified oracle-generation workflow that separates language-generalised specification elicitation from language-specific execution adapters, (ii) dual instantiations for Java and Python under the same oracle micro-skeleton interface, and (iii) a mixed-method evaluation design combining qualitative equivalence/alignment analysis with execution-based discrimination measurement where benchmark artefacts permit. Together, these contributions operationalise LLM-based oracle generation as a reproducible workflow rather than an ad hoc prompting exercise.

The results show that the model can often produce oracles that are close to micro-skeleton compliant and that encode plausible bug-signature predicates, particularly when defect descriptions expose an explicit runtime signal (e.g., exceptions or guards). Feasibility is nonetheless conditional on engineering effort that clusters into (i) interface repairs to satisfy scaffolding and structural conventions, (ii) runtime binding repairs induced by the execution embedding (most visibly for Java interop), and (iii) occasional semantic clarification when the artefacts under-specify the intended failure predicate.

On subjects with usable EvoRepair-derived passing/failing pools, many generated oracles discriminate strongly between bug-triggering and non-bug-triggering executions, while failures concentrate into directional error modes: false positive (FP) heavy collapse for over-broad predicate oracles and false negative (FN) heavy behaviour when exception-driven predicates lack correct gating or identity checks. Crucially, reliable executability does not guarantee a faithful bug signature, and conversely, semantically strong predicates may require only small, well-scoped repairs to meet the pipeline contract. As such, the empirical evidence supports a qualified effectiveness claim: the pipeline can yield practically useful oracles, but robustness depends on both predicate correctness and the stability of the runtime embedding.

Across Java and Python settings, the model exhibits consistent strategy preferences (notably exception- and guard-driven signatures) and recurring weak-oracle pathologies (vacuous defaults, redundant checks, or predicates over the wrong observable). At the same time, key differences arise from the execution context: Java introduces substantial binding and overload-selection complexity, whereas the Python setting more often exposes import/package fragility and ambiguity about which observation channel (return value versus stdout/stderr) carries the bug signal.

The findings collectively yield a structured answer to the thesis research questions. For **RQ1**, the evidence shows that GPT-4o-Mini can generate *correct* and *non-trivial*

oracle predicates from method-level artefacts to a substantial extent, but that success is strongly conditioned on defect observability and predicate complexity: exception- and guard-driven defect narratives produce the most consistently aligned predicates, whereas stateful and return-value specifications remain more prone to proxy checks, over-approximation, and example-bound constraints. For **RQ2**, the execution-based results demonstrate that, once brought into a stable executable form, many generated oracles do act as effective discriminators on benchmark-derived passing/failing pools, with the dominant practical risk being *false alarms* from overly eager value predicates rather than systematic failure to detect bug-triggering executions. For **RQ3**, the comparative analysis indicates that the model's *choice of oracle strategy* is broadly stable across languages, while the main sources of divergence are operational: differences in runtime embedding, exception representation, and observable channels shape which repair effort is required and which failure modes surface in practice.

Overall, the study indicates that LLM-based oracle generation is feasible and can be empirically discriminative when constrained by a disciplined prompt contract and a stable micro-skeleton interface. The main scalability boundary is that automation is limited not only by model capability, but by cumulative contract repairs and occasional semantic underspecification in the artefacts.

**Future work.** Several directions follow directly from the observed failure modes. A first priority is reducing runtime binding repairs through stronger adapter abstractions: typed wrappers for common Java value classes, systematic overload selection, and defensive normalisation of exceptions and attribute access could substantially reduce JPype-related friction and improve executability consistency. A second direction is improving the model's ability to select the correct observation channel and avoid weak predicates: prompt-level constraints could be complemented with lightweight static checks (detecting redundancies, unconditional `PASSING` defaults, or predicates unrelated to MuT outputs) and with self-consistency sampling that rejects outlier generations whose verdict distributions collapse.

A third direction is expanding beyond single-execution predicates where benchmarks permit: differential oracle styles (as seen in Tests4Py) suggest a richer class of specifications that compare behaviours across versions or configurations; integrating these into the pipeline would require extending the micro-skeleton contract to support controlled multi-run comparisons while preserving reproducibility. Beyond these immediate extensions, future work could explore adaptive prompting strategies that condition on prior generation failures, as well as tighter integration with program analysis techniques (e.g., lightweight symbolic execution or invariant inference) to ground generated predicates more firmly in program semantics. Finally, broader empirical validation on private or industrial defect corpora would be valuable to assess robustness under non-public artefacts and to further reduce concerns about benchmark exposure.

# References


[1] C. Augusto, A. Bertolino, G. D. Angelis, F. Lonetti, and J. Morán. Large language models for software testing: A research roadmap, 2025.

[2] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, pages 507–525, May 2015.

[3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners, 2020.

[4] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, and A. Jangda. Multipl-e: A scalable and extensible approach to benchmarking neural code generation, Dec. 2022.

[5] T. Y. Chen. Metamorphic testing: A simple approach to alleviate the oracle problem. In *2010 Fifth IEEE International Symposium on Service Oriented System Engineering*, pages 1–2, 2010.

[6] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: A new approach for generating next test cases, 2020.

[7] B. Chu, Y. Feng, K. Liu, Z. Nan, Z. Guo, and B. Xu. Large language models for unit test generation: Achievements, challenges, and the road ahead, 2025.

[8] CommonMark. Commonmark spec 0.31.2: Fenced code blocks. CommonMark Specification (version 0.31.2), 2024. Accessed: 2025-12-18.

[9] T. D. Cook and D. T. Campbell. *Quasi-Experimentation: Design & Analysis Issues for Field Settings*. Houghton Mifflin, Boston, 1979.

[10] E. Dinella, G. Ryan, and T. Mytkowicz. Toga: A neural method for test oracle generation. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 2130–2141, 2021.

[11] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. A comprehensive survey of trends in oracles for software testing. 2013.

[12] S. B. Hossain and M. Dwyer. Togll: Correct and strong test oracle generation with llms. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 635–635. IEEE Computer Society, May 2025.

[13] S. B. Hossain, R. Taylor, and M. B. Dwyer. Doc2oracle: Investigating the impact of javadoc comments on test oracle generation. *CoRR*, 2024.

[14] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang. Large language models for software engineering: A systematic literature review, 2024.

[15] S. Jiang, C. Zhu, and S. Khurshid. Generating executable oracles to check conformance of client code to requirements of jdk javadocs using llms. *ArXiv*, 2024.

[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*, page 437–440. Association for Computing Machinery, 2014.

[17] U. Kanewala and J. M. Bieman. Testing scientific software: A systematic literature review. *Information and Software Technology*, 56(10):1219–1232, Oct. 2014. PMID: 25125798; PMCID: PMC4128280.

[18] S. Kang, J. Yoon, and S. Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *Proceedings of the 45th International Conference on Software Engineering*, page 2312–2323. IEEE Press, 2023.

[19] M. Konstantinou, R. Degiovanni, and M. Papadakis. Do llms generate test oracles that capture the actual or the expected program behaviour?, 2024.

[20] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931, 2023.

[21] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey, 2019.

[22] D. Molinelli, L. D. Grazia, A. Martin-Lopez, M. D. Ernst, and M. Pezze. Do llms generate useful test oracles? an empirical study with an unbiased dataset. *IEEE/ACM International Conference on Automated Software Engineering 2025*, Nov. 2025.

[23] P. Nie, R. Banerjee, J. J. Li, R. J. Mooney, and M. Gligoric. Learning deep semantics for test completion, Mar. 2023.

[24] OpenAI. Completions. `https://platform.openai.com/docs/api-reference/completions/create?locale=en&utm_source=chatgpt.com`. [Accessed 20-11-2025].

[25] OpenAI. Prompt engineering. OpenAI API Documentation, 2025. Accessed: 2025-12-17.

[26] W. C. Ouedraogo, K. Kabore, H. Tian, Y. Song, A. Koyuncu, J. Klein, D. Lo, and T. F. Bissyande. Llms and prompting for unit test generation: A large-scale evaluation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 2464–2465, New York, NY, USA, 2024. Association for Computing Machinery.

[27] A. Panwar. Anveshana's international journal of research in engineering and applied sciences. *SSRN Electron. J.*, 2024.

[28] Q. Peng, Y. Chai, and X. Li. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization, Mar. 2024.

[29] A. Poth, O. Rrjolli, and H. Wang. Benchmarking ai-facilitated ui test-script generation: A reproducible evaluation framework. In M. Yilmaz, P. Clarke, A. Riel, R. Messnarz, M. Zelmenis, and I. A. Buce, editors, *Systems, Software and Services Process Improvement*, pages 143–157, Cham, Aug. 2025. Springer Nature Switzerland.

[30] A. Poth, O. Rrjolli, H. Wang, and K. Schmid. Baseline evaluation of llm-facilitated ui test-case generation from gherkin specifications. In M. Yilmaz, P. Clarke, A. Riel, R. Messnarz, M. Zelmenis, and I. A. Buce, editors, *Systems, Software and Services Process Improvement*, 2025.

[31] H. Ruan, H. L. Nguyen, R. Shariffdeen, Y. Noller, and A. Roychoudhury. Evolutionary testing for program repair. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 105–116, May 2024.

[32] S. Schulhoff". The Sandwich Defense: Strengthening AI Prompt Security — learnprompting.org. `https://learnprompting.org/docs/prompt_hacking/defensive_measures/sandwich_defense?srsltid=AfmBOoqYbHQ7q1emPKIk0u5fMbIkL2y9hMYhaq29a5CqL0QwVa_AfAFD`. [Accessed 22-07-2025].

[33] S. Schulhoff, M. Ilie, N. Balepur, K. Kahadze, A. Liu, C. Si, Y. Li, A. Gupta, H. Han, S. Schulhoff, P. S. Dulepet, S. Vidyadhara, D. Ki, S. Agrawal, C. Pham, G. Kroiz, F. Li, H. Tao, A. Srivastava, H. D. Costa, S. Gupta, M. L. Rogers, I. Goncearenco, G. Sarli, I. Galynker, D. Peskoff, M. Carpuat, J. White, S. Anadkat, A. Hoyle, and P. Resnik. The prompt report: A systematic survey of prompt engineering techniques, 2025.

[34] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip. An empirical evaluation of using large language models for automated unit test generation, 2023.

[35] D. Seca. A review on oracle issues in machine learning, 2021.

[36] M. Smytzek, M. Eberlein, B. Serçe, L. Grunske, and A. Zeller. Tests4py: A benchmark for system testing. page 557–561. Association for Computing Machinery, 2024.

[37] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan. Unit test case generation with transformers and focal context, 2021.

[38] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, Aug. 2012.

[39] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need, 2023.

[40] V. Vikram, C. Lemieux, J. Sunshine, and R. Padhye. Can large language models write good property-based tests?, 2024.

[41] Y. Wang, C. Xia, W. Zhao, J. Du, C. Miao, Z. Deng, P. S. Yu, and C. Xing. Projecttest: A project-level llm unit test generation benchmark and impact of error fixing mechanisms, 2025.

[42] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[43] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, pages 465–470, 11 1982.

[44] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2020.

[45] R. Xu, J. Cao, Y. Lu, M. Wen, H. Lin, X. Han, B. He, S.-C. Cheung, and L. Sun. Cruxeval-x: A benchmark for multilingual code reasoning, understanding and execution, May 2025.

[46] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang, and J. Chen. On the evaluation of large language models in unit test generation, 2024.

[47] L. Yang, C. Yang, G. Shutao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang, and J. Chen. An empirical study of unit test generation with large language models, 06 2024.

[48] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, Oct. 2012.

[49] J. M. Zhang, M. Harman, L. Ma, and Y. Liu. Machine learning testing: Survey, landscapes and horizons. 48(1):1–36, Jan. 2022.

# Appendix

## 8.1 System prompt template

```
[TASK]
You are a software testing engineer.
Your task is to generate a Python test oracle for the method-under-test (MuT) described in the prompt.
. The oracle must follow the provided micro-skeleton: it calls the MuT once, then returns exactly one of:
- TestResult.FAILING
- TestResult.PASSING
- TestResult.UNDEFINED

Use the provided documentation and method source code to reason about what constitutes correct behaviour. Use the provided bug report to aid in determining what incorrect behaviour may look like.
Treat any values shown in the bug report as illustrative only. Use the structure to generate the oracle in a standardised format. The oracle should be able to classify any range of inputs as PASSING, FAILING, or UNDEFINED.
Derive a general predicate for the bug signature from the report/spec and compute it from the inputs at runtime.

[WHAT 'FAILING / UNDEFINED / PASSING' MEAN]
- FAILING: The bug report's conditions for a failure are observed (by exception type for specific inputs, or by value/state conditions). This is a **bug-signature**.
- UNDEFINED: An unexpected exception occurred (not listed as a bug-signature), or required pass checks cannot be confirmed.
- PASSING: No failure condition triggered and no unexpected exception.

[STYLE & SCOPE]
- Keep the oracle lean. Only fill the marked 'LLM-FILL' spots in the skeleton.
- Use **fully qualified** exception names.
- Do not re-implement helpers; if needed, call them via the same mechanism.
```

Listing 29: System prompt template.

## 8.2 Generic oracle micro-skeleton

```
# === GENERIC ORACLE MICRO-SKELETON =========
# Shared core between the Python-native and Java-via-bridge variants.
# Language-specific details live only in CALL_MUT(...) and exception-id normalisation.
FAIL_EXCEPTIONS = {
    # "<ExceptionId>": "bug-signature reason",
}

def CALL_MUT(receiver_or_runtime, *args, **kwargs):
    # LLM-FILL (ONE line): call the MuT exactly once (direct, bridge, or subprocess)
    raise NotImplementedError

def oracle_<MUT_NAME>(receiver_or_runtime, *args, **kwargs):
    """
    Return (verdict, message, details) where verdict {"PASSING","FAILING","UNDEFINED"}.
    Keep structure unchanged.
    """
    try:
        ret = CALL_MUT(receiver_or_runtime, *args, **kwargs)
    except Exception as e:
        et = EXCEPTION_ID(e) # language-specific normalisation (e.g., FQN or type name)
        if et in FAIL_EXCEPTIONS:
            return "FAILING", f"{et}: {FAIL_EXCEPTIONS[et]}", {"exception": et, "args": args}
        return "UNDEFINED", f"{et}: {str(e)}", {"exception": et, "args": args}
    # LLM-FILL: bug-signature checks on the non-exception path
    # if <COND on ret/args/state>:
    # return "FAILING", "bug-signature condition observed", {"ret": str(ret), "args": args}
    # (optional) LLM-FILL: explicit pass confirmations; otherwise default PASSING
    # if not <PASS_COND>:
    # return "UNDEFINED", "pass condition not confirmable", {"ret": str(ret), "args": args}
    return "PASSING", "OK", {"ret": str(ret), "args": args}
# ===
```

Listing 30: General microskeleton structure.

## 8.3 Initial EvoRepair test mining totals

| Bug ID | Passing total | Failing total |
|---|---|---|
| Chart-1 | 26 | 100 |
| Chart-12 | 100 | 0 |
| Chart-5 | 100 | 100 |
| Lang-16 | 100 | 100 |
| Lang-20 | 100 | 100 |
| Lang-22 | 0 | 0 |
| Lang-35a | 0 | 0 |
| Lang-35b | 0 | 0 |
| Lang-39 | 100 | 100 |
| Lang-41 | 100 | 7 |
| Lang-43 | 0 | 0 |
| Lang-45 | 100 | 100 |
| Lang-46 | 100 | 100 |
| Lang-50b | 100 | 100 |
| Lang-50a | 100 | 100 |
| Lang-51 | 100 | 100 |
| Lang-55 | 100 | 100 |
| Lang-59 | 100 | 100 |
| Lang-61a | 100 | 25 |
| Lang-61b | 100 | 14 |
| Lang-7 | 100 | 100 |
| Math-103 | 0 | 0 |
| Math-2 | 100 | 100 |
| Math-20 | 100 | 0 |
| Math-22 | 100 | 100 |
| Math-39 | 0 | 0 |
| Math-49 | 100 | 100 |
| Math-50 | 100 | 100 |
| Math-53 | 100 | 100 |
| Math-56 | 32 | 100 |
| Math-58 | 0 | 100 |
| Math-60 | 0 | 0 |
| Math-70 | 0 | 0 |
| Math-73 | 55 | 100 |
| Math-74 | 0 | 0 |
| Math-8 | 0 | 0 |
| Math-81 | 100 | 0 |
| Math-95 | 100 | 100 |
| Math-98 | 100 | 100 |
| Time-11 | 0 | 0 |
| Time-14 | 0 | 0 |

Table 19: Initial mined passing/failing test totals per Defects4J subject from EvoRepair experimental results (prior to translation, refinement, and noise removal).

## 8.4 Defects4J discrimination performance overall

| **Aggregation** | **Acc.**$_\mu$ | **Prec.**$_\mu$ | **Rec.**$_\mu$ | **F1**$_\mu$ | **Acc.**$_\sigma$ | **Prec.**$_\sigma$ | **Rec.**$_\sigma$ | **F1**$_\sigma$ | $n$ |
|---|---|---|---|---|---|---|---|---|---|
| Macro | 0.933494 | 0.944693 | 0.960615 | 0.942956 | 0.105340 | 0.121994 | 0.068062 | 0.078986 | 17 |
| Micro | 0.924314 | 0.899290 | 0.955381 | 0.926487 | 0.127080 | 0.143360 | 0.075756 | 0.096525 | 85 |

Table 20: Overall macro- and micro-aggregated discrimination performance across all Defects4J subjects and generations.

## 8.5 Defects4J discrimination performance by project

| **Project** | **Acc.**$_\mu$ | **Acc.**$_\sigma$ | **Prec.**$_\mu$ | **Prec.**$_\sigma$ | **Rec.**$_\mu$ | **Rec.**$_\sigma$ | **F1**$_\mu$ | **F1**$_\sigma$ | $n$ **subj.** |
|---|---|---|---|---|---|---|---|---|---|
| Chart | 0.988776 | 0.015874 | 1.000000 | 0.000000 | 0.977778 | 0.031427 | 0.988636 | 0.016071 | 2 |
| Lang | 0.920409 | 0.122115 | 0.917716 | 0.155052 | 0.958023 | 0.057145 | 0.928187 | 0.091468 | 9 |
| Math | 0.934693 | 0.110844 | 0.940171 | 0.146551 | 0.958781 | 0.100965 | 0.940024 | 0.096104 | 6 |
| **ALL** | **0.933494** | **0.108582** | **0.935322** | **0.139421** | **0.960615** | **0.070157** | **0.939477** | **0.086368** | **17** |

Table 21: Macro-averaged discrimination performance across empirically evaluated subjects, grouped by Defects4J project family (and overall).

| **Project** | **Acc.**$_\mu$ | **Acc.**$_\sigma$ | **Prec.**$_\mu$ | **Prec.**$_\sigma$ | **Rec.**$_\mu$ | **Rec.**$_\sigma$ | **F1**$_\mu$ | **F1**$_\sigma$ | $n$ **subj.** |
|---|---|---|---|---|---|---|---|---|---|
| Chart | 0.985760 | 0.02245 | 1.000000 | 0.000000 | 0.964516 | 0.044444 | 0.981938 | 0.023529 | 2 |
| Lang | 0.912835 | 0.150715 | 0.884954 | 0.161585 | 0.961008 | 0.068291 | 0.921414 | 0.108710 | 9 |
| Math | 0.921021 | 0.101186 | 0.898551 | 0.133782 | 0.941772 | 0.092168 | 0.919654 | 0.087730 | 6 |
| **ALL** | **0.9243134** | **0.127079** | **0.899290** | **0.143360** | **0.95538** | **0.075756** | **0.926487** | **0.096525** | **17** |

Table 22: Micro-averaged discrimination performance across empirically evaluated subjects, grouped by Defects4J project family (and overall).

## 8.6 Defects4J discrimination performance by subject

| Subject | $\text{Acc.}_{\mu}$ | $\text{Acc.}_{\sigma}$ | $\text{Prec.}_{\mu}$ | $\text{Prec.}_{\sigma}$ | $\text{Rec.}_{\mu}$ | $\text{Rec.}_{\sigma}$ | $\text{F1}_{\mu}$ | $\text{F1}_{\sigma}$ |
|---|---|---|---|---|---|---|---|---|
| Chart_1 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 |
| Chart_5 | 0.977551 | 0.027494 | 1.000000 | 0.000000 | 0.955556 | 0.054433 | 0.976471 | 0.028818 |
| Lang_16 | 0.925128 | 0.052758 | 1.000000 | 0.000000 | 0.846316 | 0.108293 | 0.913110 | 0.062259 |
| Lang_39 | 0.961326 | 0.011050 | 0.990805 | 0.004598 | 0.931868 | 0.026374 | 0.960168 | 0.012248 |
| Lang_45 | 0.957252 | 0.021374 | 1.000000 | 0.000000 | 0.885714 | 0.057143 | 0.938462 | 0.030769 |
| Lang_46 | 0.886131 | 0.213139 | 0.877311 | 0.245378 | 0.978723 | 0.000000 | 0.902241 | 0.174012 |
| Lang_50a | 0.608466 | 0.190476 | 0.600134 | 0.189516 | 1.000000 | 0.000000 | 0.735038 | 0.127218 |
| Lang_51 | 0.970000 | 0.000000 | 0.969697 | 0.000000 | 1.000000 | 0.000000 | 0.984615 | 0.000000 |
| Lang_55 | 0.990816 | 0.008163 | 0.992308 | 0.015385 | 0.990000 | 0.012649 | 0.991013 | 0.007931 |
| Lang_61a | 0.989796 | 0.000000 | 0.988506 | 0.000000 | 1.000000 | 0.000000 | 0.994220 | 0.000000 |
| Lang_7 | 0.994764 | 0.000000 | 1.000000 | 0.000000 | 0.989583 | 0.000000 | 0.994764 | 0.000000 |
| Math_22 | 0.875676 | 0.000000 | 1.000000 | 0.000000 | 0.752688 | 0.000000 | 0.858896 | 0.000000 |
| Math_49 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 |
| Math_56 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 |
| Math_73 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 |
| Math_95 | 0.732484 | 0.000000 | 0.641026 | 0.000000 | 1.000000 | 0.000000 | 0.781250 | 0.000000 |
| Math_98 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 |

Table 23: Macro-averaged discrimination performance across empirically evaluated subjects.

| Subject | $\text{Acc.}_{\mu}$ | $\text{Acc.}_{\sigma}$ | $\text{Prec.}_{\mu}$ | $\text{Prec.}_{\sigma}$ | $\text{Rec.}_{\mu}$ | $\text{Rec.}_{\sigma}$ | $\text{F1}_{\mu}$ | $\text{F1}_{\sigma}$ |
|---|---|---|---|---|---|---|---|---|
| Chart_1 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 |
| Chart_5 | 0.977551 | 0.027494 | 1.000000 | 0.000000 | 0.955556 | 0.054433 | 0.977273 | 0.028818 |
| Lang_16 | 0.925128 | 0.052758 | 1.000000 | 0.000000 | 0.846316 | 0.108293 | 0.916762 | 0.062259 |
| Lang_39 | 0.961326 | 0.011050 | 0.990654 | 0.004598 | 0.931868 | 0.026374 | 0.960362 | 0.012248 |
| Lang_45 | 0.957252 | 0.021374 | 1.000000 | 0.000000 | 0.885714 | 0.057143 | 0.939394 | 0.030769 |
| Lang_46 | 0.886131 | 0.213139 | 0.759076 | 0.245378 | 0.978723 | 0.000000 | 0.855019 | 0.174012 |
| Lang_50a | 0.608466 | 0.190476 | 0.559524 | 0.189516 | 1.000000 | 0.000000 | 0.717557 | 0.127218 |
| Lang_51 | 0.970000 | 0.000000 | 0.969697 | 0.000000 | 1.000000 | 0.000000 | 0.984615 | 0.000000 |
| Lang_55 | 0.990816 | 0.008163 | 0.991984 | 0.015385 | 0.990000 | 0.012649 | 0.990991 | 0.007931 |
| Lang_61a | 0.989796 | 0.000000 | 0.988506 | 0.000000 | 1.000000 | 0.000000 | 0.994220 | 0.000000 |
| Lang_7 | 0.994764 | 0.000000 | 1.000000 | 0.000000 | 0.989583 | 0.000000 | 0.994764 | 0.000000 |
| Math_22 | 0.875676 | 0.000000 | 1.000000 | 0.000000 | 0.752688 | 0.000000 | 0.858896 | 0.000000 |
| Math_49 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 |
| Math_56 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 |
| Math_73 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 |
| Math_95 | 0.732484 | 0.000000 | 0.641026 | 0.000000 | 1.000000 | 0.000000 | 0.781250 | 0.000000 |
| Math_98 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 |

Table 24: Micro-averaged discrimination performance across empirically evaluated subjects.

## 8.7 Defects4J discrimination performance per generation

Table 25: Per-generation discrimination results for Defects4J targets with usable EvoRepair-derived passing/failing tests. Metrics are computed over this combined set.

| Subject | Gen | TP | TN | FP | FN | #Tests | Acc. | Prec. | Rec. | F1 |
|---|---|---|---|---|---|---|---|---|---|---|
| Chart_1 | 0 | 25 | 88 | 0 | 0 | 113 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Chart_1 | 1 | 25 | 88 | 0 | 0 | 113 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Chart_1 | 2 | 25 | 88 | 0 | 0 | 113 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Chart_1 | 3 | 25 | 88 | 0 | 0 | 113 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Chart_1 | 4 | 25 | 88 | 0 | 0 | 113 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Chart_5 | 0 | 99 | 97 | 0 | 0 | 196 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Chart_5 | 1 | 99 | 97 | 0 | 0 | 196 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Chart_5 | 2 | 88 | 97 | 0 | 11 | 196 | 0.943878 | 1.000000 | 0.888889 | 0.941176 |
| Chart_5 | 3 | 99 | 97 | 0 | 0 | 196 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Chart_5 | 4 | 88 | 97 | 0 | 11 | 196 | 0.943878 | 1.000000 | 0.888889 | 0.941176 |
| Lang_16 | 0 | 93 | 100 | 0 | 2 | 195 | 0.989744 | 1.000000 | 0.978947 | 0.989362 |
| Lang_16 | 1 | 93 | 100 | 0 | 2 | 195 | 0.989744 | 1.000000 | 0.978947 | 0.989362 |
| Lang_16 | 2 | 72 | 100 | 0 | 23 | 195 | 0.882051 | 1.000000 | 0.757895 | 0.862275 |
| Lang_16 | 3 | 72 | 100 | 0 | 23 | 195 | 0.882051 | 1.000000 | 0.757895 | 0.862275 |
| Lang_16 | 4 | 72 | 100 | 0 | 23 | 195 | 0.882051 | 1.000000 | 0.757895 | 0.862275 |
| Lang_39 | 0 | 86 | 89 | 1 | 5 | 181 | 0.966851 | 0.988506 | 0.945055 | 0.966292 |
| Lang_39 | 1 | 86 | 89 | 1 | 5 | 181 | 0.966851 | 0.988506 | 0.945055 | 0.966292 |
| Lang_39 | 2 | 80 | 90 | 0 | 11 | 181 | 0.939227 | 1.000000 | 0.879121 | 0.935673 |
| Lang_39 | 3 | 86 | 89 | 1 | 5 | 181 | 0.966851 | 0.988506 | 0.945055 | 0.966292 |
| Lang_39 | 4 | 86 | 89 | 1 | 5 | 181 | 0.966851 | 0.988506 | 0.945055 | 0.966292 |
| Lang_45 | 0 | 42 | 82 | 0 | 7 | 131 | 0.946565 | 1.000000 | 0.857143 | 0.923077 |
| Lang_45 | 1 | 42 | 82 | 0 | 7 | 131 | 0.946565 | 1.000000 | 0.857143 | 0.923077 |
| Lang_45 | 2 | 42 | 82 | 0 | 7 | 131 | 0.946565 | 1.000000 | 0.857143 | 0.923077 |
| Lang_45 | 3 | 42 | 82 | 0 | 7 | 131 | 0.946565 | 1.000000 | 0.857143 | 0.923077 |
| Lang_45 | 4 | 49 | 82 | 0 | 0 | 131 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Lang_46 | 0 | 46 | 90 | 0 | 1 | 137 | 0.992701 | 1.000000 | 0.978723 | 0.989247 |
| Lang_46 | 1 | 46 | 90 | 0 | 1 | 137 | 0.992701 | 1.000000 | 0.978723 | 0.989247 |
| Lang_46 | 2 | 46 | 17 | 73 | 1 | 137 | 0.459854 | 0.386555 | 0.978723 | 0.554217 |
| Lang_46 | 3 | 46 | 90 | 0 | 1 | 137 | 0.992701 | 1.000000 | 0.978723 | 0.989247 |
| Lang_46 | 4 | 46 | 90 | 0 | 1 | 137 | 0.992701 | 1.000000 | 0.978723 | 0.989247 |
| Lang_50a | 0 | 94 | 93 | 2 | 0 | 189 | 0.989418 | 0.979167 | 1.000000 | 0.989474 |
| Lang_50a | 1 | 94 | 3 | 92 | 0 | 189 | 0.513228 | 0.505376 | 1.000000 | 0.671429 |
| Lang_50a | 2 | 94 | 3 | 92 | 0 | 189 | 0.513228 | 0.505376 | 1.000000 | 0.671429 |
| Lang_50a | 3 | 94 | 3 | 92 | 0 | 189 | 0.513228 | 0.505376 | 1.000000 | 0.671429 |
| Lang_50a | 4 | 94 | 3 | 92 | 0 | 189 | 0.513228 | 0.505376 | 1.000000 | 0.671429 |
| Lang_51 | 0 | 96 | 1 | 3 | 0 | 100 | 0.970000 | 0.969697 | 1.000000 | 0.984615 |
| Lang_51 | 1 | 96 | 1 | 3 | 0 | 100 | 0.970000 | 0.969697 | 1.000000 | 0.984615 |
| Lang_51 | 2 | 96 | 1 | 3 | 0 | 100 | 0.970000 | 0.969697 | 1.000000 | 0.984615 |
| Lang_51 | 3 | 96 | 1 | 3 | 0 | 100 | 0.970000 | 0.969697 | 1.000000 | 0.984615 |
| Lang_51 | 4 | 96 | 1 | 3 | 0 | 100 | 0.970000 | 0.969697 | 1.000000 | 0.984615 |
| Lang_55 | 0 | 97 | 96 | 0 | 3 | 196 | 0.984694 | 1.000000 | 0.970000 | 0.984772 |
| Lang_55 | 1 | 100 | 96 | 0 | 0 | 196 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Lang_55 | 2 | 100 | 96 | 0 | 0 | 196 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Lang_55 | 3 | 100 | 92 | 4 | 0 | 196 | 0.979592 | 0.961538 | 1.000000 | 0.980392 |
| Lang_55 | 4 | 98 | 96 | 0 | 2 | 196 | 0.989796 | 1.000000 | 0.980000 | 0.989899 |
| Lang_61a | 0 | 86 | 11 | 1 | 0 | 98 | 0.989796 | 0.988506 | 1.000000 | 0.994220 |
| Lang_61a | 1 | 86 | 11 | 1 | 0 | 98 | 0.989796 | 0.988506 | 1.000000 | 0.994220 |
| Lang_61a | 2 | 86 | 11 | 1 | 0 | 98 | 0.989796 | 0.988506 | 1.000000 | 0.994220 |
| Lang_61a | 3 | 86 | 11 | 1 | 0 | 98 | 0.989796 | 0.988506 | 1.000000 | 0.994220 |
| Lang_61a | 4 | 86 | 11 | 1 | 0 | 98 | 0.989796 | 0.988506 | 1.000000 | 0.994220 |
| Lang_7 | 0 | 95 | 95 | 0 | 1 | 191 | 0.994764 | 1.000000 | 0.989583 | 0.994764 |
| Lang_7 | 1 | 95 | 95 | 0 | 1 | 191 | 0.994764 | 1.000000 | 0.989583 | 0.994764 |
| Lang_7 | 2 | 95 | 95 | 0 | 1 | 191 | 0.994764 | 1.000000 | 0.989583 | 0.994764 |
| Lang_7 | 3 | 95 | 95 | 0 | 1 | 191 | 0.994764 | 1.000000 | 0.989583 | 0.994764 |

*Continued on next page*

| Subject | Gen | TP | TN | FP | FN | #Tests | Acc. | Prec. | Rec. | F1 |
|---|---|---|---|---|---|---|---|---|---|---|
| Lang_7 | 4 | 95 | 95 | 0 | 1 | 191 | 0.994764 | 1.000000 | 0.989583 | 0.994764 |
| Math_22 | 0 | 70 | 92 | 0 | 23 | 185 | 0.875676 | 1.000000 | 0.752688 | 0.858896 |
| Math_22 | 1 | 70 | 92 | 0 | 23 | 185 | 0.875676 | 1.000000 | 0.752688 | 0.858896 |
| Math_22 | 2 | 70 | 92 | 0 | 23 | 185 | 0.875676 | 1.000000 | 0.752688 | 0.858896 |
| Math_22 | 3 | 70 | 92 | 0 | 23 | 185 | 0.875676 | 1.000000 | 0.752688 | 0.858896 |
| Math_22 | 4 | 70 | 92 | 0 | 23 | 185 | 0.875676 | 1.000000 | 0.752688 | 0.858896 |
| Math_49 | 0 | 98 | 61 | 0 | 0 | 159 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_49 | 1 | 98 | 61 | 0 | 0 | 159 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_49 | 2 | 98 | 61 | 0 | 0 | 159 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_49 | 3 | 98 | 61 | 0 | 0 | 159 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_49 | 4 | 98 | 61 | 0 | 0 | 159 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_56 | 0 | 25 | 94 | 0 | 0 | 119 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_56 | 1 | 25 | 94 | 0 | 0 | 119 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_56 | 2 | 25 | 94 | 0 | 0 | 119 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_56 | 3 | 25 | 94 | 0 | 0 | 119 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_56 | 4 | 25 | 94 | 0 | 0 | 119 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_73 | 0 | 7 | 18 | 0 | 0 | 25 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_73 | 1 | 7 | 18 | 0 | 0 | 25 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_73 | 2 | 7 | 18 | 0 | 0 | 25 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_73 | 3 | 7 | 18 | 0 | 0 | 25 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_73 | 4 | 7 | 18 | 0 | 0 | 25 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_95 | 0 | 75 | 40 | 42 | 0 | 157 | 0.732484 | 0.641026 | 1.000000 | 0.781250 |
| Math_95 | 1 | 75 | 40 | 42 | 0 | 157 | 0.732484 | 0.641026 | 1.000000 | 0.781250 |
| Math_95 | 2 | 75 | 40 | 42 | 0 | 157 | 0.732484 | 0.641026 | 1.000000 | 0.781250 |
| Math_95 | 3 | 75 | 40 | 42 | 0 | 157 | 0.732484 | 0.641026 | 1.000000 | 0.781250 |
| Math_95 | 4 | 75 | 40 | 42 | 0 | 157 | 0.732484 | 0.641026 | 1.000000 | 0.781250 |
| Math_98 | 0 | 97 | 81 | 0 | 0 | 178 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_98 | 1 | 97 | 81 | 0 | 0 | 178 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_98 | 2 | 97 | 81 | 0 | 0 | 178 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_98 | 3 | 97 | 81 | 0 | 0 | 178 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| Math_98 | 4 | 97 | 81 | 0 | 0 | 178 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |

## 8.8 Supplementary Failure-Mode Tables

| **Oracle type** | **TP** | **TN** | **FP** | **FN** | $n_{\textbf{pass}}$ | $n_{\textbf{fail}}$ | **FP-rate**$_{\textbf{pooled}}$ | **FN-rate**$_{\textbf{pooled}}$ |
|---|---|---|---|---|---|---|---|---|
| Exception oracle | 3124 | 2551 | 24 | 81 | 2575 | 3205 | 0.0093 | 0.0253 |
| Guarded signature oracle | 912 | 1065 | 0 | 78 | 1065 | 990 | 0.0000 | 0.0788 |
| Return-value predicate oracle | 1200 | 1152 | 653 | 5 | 1805 | 1205 | 0.3618 | 0.0041 |
| Stateful predicate oracle | 845 | 936 | 4 | 120 | 940 | 965 | 0.0043 | 0.1244 |

Table 26: Pooled confusion totals and derived FP/FN rates by oracle category across all evaluated Defects4J subjects.

| **Oracle type** | **#Subj.** | **Near-perfect (%)** | **Balanced (%)** | **FP-dominant (%)** | **FN-dominant (%)** |
|---|---|---|---|---|---|
| Exception oracle | 8 | 37.5 | 12.5 | 25.0 | 25.0 |
| Guarded signature oracle | 3 | 33.3 | 33.3 | 0.0 | 33.3 |
| Return-value predicate oracle | 4 | 25.0 | 0.0 | 75.0 | 0.0 |
| Stateful predicate oracle | 2 | 0.0 | 50.0 | 0.0 | 50.0 |

Table 27: Distribution of dominant error regimes by oracle category by percentage of subjects.