Testora: Using Natural Language Intent to Detect Behavioral Regressions

Michael Pradel
michael@binaervarianz.de
CISPA Helmholtz Center for Information Security
Stuttgart, Germany

Abstract

As software is evolving, code changes can introduce regression bugs or affect the behavior in other unintended ways. Traditional regression test generation is impractical for detecting unintended behavioral changes, because it reports all behavioral differences as potential regressions. However, most code changes are intended to change the behavior in some way, e.g., to fix a bug or to add a new feature. This paper presents Testora, the first automated approach that detects regressions by comparing the intentions of a code change against behavioral differences caused by the code change. Given a pull request (PR), Testora queries an LLM to generate tests that exercise the modified code, compares the behavior of the original and modified code, and classifies any behavioral differences as intended or unintended. For the classification, we present an LLM-based technique that leverages the natural language information associated with the PR, such as the title, description, and commit messages – effectively using the natural language intent to detect behavioral regressions. Applying Testora to PRs of complex and popular Python projects, we find 19 regression bugs and 11 PRs that, despite having another intention, coincidentally fix a bug. Out of 13 regressions reported to the developers, 11 have been confirmed and 9 have already been fixed. The costs of using Testora are acceptable for real-world deployment, with 12.3 minutes to check a PR and LLM costs of only $0.003 per PR. We envision our approach to be used before or shortly after a code change gets merged into a code base, providing a way to early on detect regressions that are not caught by traditional approaches.

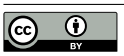
CCS Concepts

• **Software and its engineering → Software defect analysis**;

Keywords

Regression testing, test oracle, natural language processing, large language models, software maintenance

ACM Reference Format:
Michael Pradel. 2026. Testora: Using Natural Language Intent to Detect Behavioral Regressions. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3764527



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3764527

1 Introduction

Practically all successful software projects are continuously evolving. While changing code is necessary to fix bugs, add new features, improve performance, or increase the maintainability of a code base, code changes may also negatively impact the software. For example, while trying to fix a specific bug, a developer may not only remove the buggy behavior, but also accidentally introduce a new bug or modify the behavior in some other unintended way. Prior work shows that behavioral changes are common in practice, leading to regression bugs that often remain unnoticed [41].

As a real-world example of a regression bug, consider Figure 1, which shows a pull request (PR) of the popular Python *scipy* library. The title and description of the PR indicate that the change is intended to add array API support to the `differential_entropy` function. Adding array API support is a larger design change in scipy, which affects how the library internally handles different array types, but it should not change the fundamental behavior of the mathematical functions offered by scipy. However, as exposed by the test case in Figure 1, the output of the `differential_entropy` function changes from 2.3588 to 2.5285 after the PR is applied. That is, the PR causes an unintended behavioral difference, as the output of the function should not change for the same input data just because the internal handling of arrays changes. The regression remained unnoticed by the developers, who merged the PR into the code base without realizing the unintended behavioral change.

As illustrated by this example, regressions may easily remain unnoticed. One reason is that, even in well tested software, the available test suite may not cover the code modified by a code change. Automated regression test generation [5, 18, 46, 50, 55] can partially address this challenge, but lacks a useful test oracle: If a regression test generator finds a test that exposes a behavioral difference between the code before and after a code change, it remains unclear whether this behavioral difference is intended. A naive approach could report any behavioral difference as a regression, but this would lead to many false positives, because most code changes are supposed to change the behavior in some way, e.g., to fix a bug or to add a new feature [5, 41].

This paper presents Testora, an automated technique to detect regressions and other unintended behavioral changes by using natural language information associated with a code change as a test oracle. Our key idea is to compare the intentions of a code change, as provided informally in natural language, with behavioral changes exposed by generated regression tests. The approach checks PRs for unintended behavioral changes by performing three steps: (1) At first, given the code diff associated with the PR, Testora performs a targeted test generation aimed at finding tests that expose behavioral differences between the original and the modified

PR title:
ENH: stats.differential_entropy: add array API support

PR description:
...
Adds array API support to differential_entropy
...

Testora-generated test case:

```
import numpy as np
from scipy.stats import differential_entropy

values = np.array([1, 1, 2, 3, 3, 4, 5, 5, 6, 7, 8, 9, 10, 11])
result = differential_entropy(values)
print(result)
```

Output before PR: 2.3588

Output after PR: 2.5285

Testora's classification: *Unintended behavioral change*
Differential entropy calculations should yield the same results for the same datasets unless there's a specific intention to change the underlying algorithm or methodology. The change in output appears unintended because the pull request does not specify altering the fundamental behavior or calculations of entropy estimators; it only indicates adding array API support.

Figure 1: Motivating example.

code. Motivated by the recent success of large language models (LLMs) in generating effective tests [3, 28, 50, 55, 65, 67], Testora uses an LLM to generate tests that exercise the modified code. (2) Next, the approach compares the behavior of the original and the modified code by executing the generated tests on both versions of the code. (3) Finally, for any test that exposes a behavioral difference, Testora classifies the difference as intended or unintended, and reports differences that are likely unintended to the user. For the classification, Testora also uses an LLM, which is provided with the title, description, and commit messages associated with the PR, and then prompted with multiple questions to determine whether the behavioral difference is intended. To the best of our knowledge, our approach is the first to turn natural language information describing a code change into an oracle for validating behavioral differences.

We envision Testora to be continuously applied to code changes, e.g., in the form of PRs, either while a change is under discussion or shortly after it has been merged into the code base. In this setup, our approach provides several benefits: (i) It detects regressions and other unintended behavioral changes that would otherwise remain unnoticed. (ii) It provides a reproducible test case that exposes the problem, which can later be used to validate a fix and prevent the regression from reoccurring. (iii) It avoids many false positives that a traditional regression testing approach would produce, because it does not simply report any behavioral change as a problem, but compares them to the developer-documented intentions. (iv) It reports problems at a point in time when developers are most receptive to feedback, i.e., while or shortly after a developer is working on a code change. (v) It produces natural language explanations for the detected regressions, which can help developers quickly assess potential problems.

Our evaluation applies Testora to hundreds of real-world PRs from four popular and complex Python projects on GitHub. We show that the approach can detect regressions that have remained unnoticed based on the existing test suite and any other tools run as part of the continuous integration pipelines of the target projects. In total, we find 30 code changes with unintended behavioral changes, including 19 regression bugs and 11 PRs that, despite having another intention, coincidentally fix a bug. Out of 13 regressions reported to the developers, 11 have been confirmed and 9 have also been fixed at this point, with the remaining two issues still being open. The classifier that predicts whether a behavioral change is intended is effective in practice, with a precision of 55% and a recall of 67%. Finally, we find the costs of using Testora to be acceptable for real-world deployment, with 12.3 minutes to check a PR and LLM costs of only $0.003, on average.

In summary, this paper contributes the following:

- *Idea.* We are the first to present the idea of using natural language artifacts associated with a code change as the basis for a regression testing oracle.
- *Technique.* We present a novel technique for automatically detecting regressions and other unintended behavioral changes.
- *Evidence.* We report the results of applying the approach to hundreds of real-world PRs, which shows its effectiveness and cost-efficiency.
- *Dataset and implementation.* We make our dataset, which is the first of its kind, and our implementation publicly available to foster future research.

2 Approach

2.1 Problem Statement

Before presenting our approach, let us define the problem we address. Testora aims to detect regressions in code changes that are introduced via PRs in a software project. The input to the approach is a pull request $pr = (t, d, \Delta, m_c, m_d)$ that contains a title t, a description d of the intended changes, a diff Δ that shows the changes to the code, the commit messages m_c of the code changes, and any messages m_d exchanged among the developers while discussing the PR. Given pr, the goal of Testora is to determine whether the changes in Δ introduce a behavioral difference that does not align with the intention of the PR. That is, our Testora approach yields one of two possible outputs:

$$Testora(pr) = \begin{cases} (\text{“unintended”, c, e}) & \text{if } \Delta \text{ causes behavioral differences that are inconsistent with } pr \\ \text{“intended”} & \text{otherwise} \end{cases}$$

In case the approach yields "unintended", it also provides a test case c that exposes the behavioral difference and a natural language explanation e that describes why the behavioral difference is not in line with the intentions of the PR.

For the motivating example in Figure 1, the input to Testora includes the PR title and description, as shown in the upper part of the figure. Testora yields "unintended", along with the test case

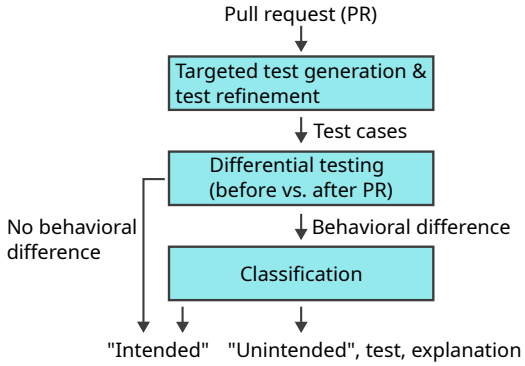


Figure 2: Overview of the approach.

shown in the middle part of the figure and the explanation shown at the bottom.

2.2 Overview

To address the problem defined above, Testora follows the workflow shown in Figure 2. The approach consists of three main steps. Given a PR, the first step is to generate tests aimed at exercising the modified code. We introduce an LLM-based test generation and test refinement technique for this purpose (Section 2.4). Based on the generated test cases, the next step compares the behavior of the original and the modified code via differential testing (Section 2.5). If the differential testing reveals a behavioral difference, the important question is whether this difference is intended, e.g., due to a new feature added by the PR, or unintended, e.g., due to a regression bug introduced by the PR. To answer this question, Testora uses an LLM-based classifier that compares the intention of the PR, as given in the PR title, description, and commit messages, with the observed behavioral difference (Section 2.6). If the classifier determines that the behavioral difference is unintended, Testora provides the test case that exposes the problem, along with a natural language explanation of why the behavior does not match the PR's intention. The remainder of this section provides details on the different components of the approach.

2.3 Selecting Code Changes to Check in Detail

Our approach is designed to automatically check a large number of PRs. To keep the time and monetary costs manageable, we apply two filters to ignore PRs that are out of scope for our approach or that are unlikely to introduce unintended behavioral differences.

The first filter focuses on PRs that modify at least one file in the main source code of the target project, as opposed to modifying only other files, such as a README file, test cases, or configuration files. To identify the main source code files, we consider files with specific extensions, i.e., `.py`, `.pyx`, `.c`, `.cpp`, and `.h`,[1] and that start with either "src/⟨m⟩" or "⟨m⟩", where ⟨m⟩ is the module name of the project. Moreover, we exclude files with a path that contains "test". If a PR does not modify any of the main source code files, we ignore it. We also ignore PRs that modify more than three main source code files, as these PRs tend to be too complex to analyze for Testora.

The second filter ignores documentation-only changes. To this end, we parse both the old and the new version of all source code files modified by a PR into an AST and check if any changes beyond comments are made. If a PR modifies only comments or other documentation, we ignore it. Finally, we exclude PRs where the title starts with "DOC", which is commonly used to indicate documentation-only changes.

2.4 Targeted Generation and Refinement of Regression Tests

The first main step of Testora is to generate tests that exercise the modified code, aiming to expose behavioral differences between the original and modified code. Given the recent success of large language models (LLMs) in generating code and tests [3, 11, 28, 50, 55, 65, 67], we use a combination of lightweight static analysis and LLM prompting for this purpose. The approach consists of three sub-steps: (1) gathering information about the code change, (2) querying an LLM to generate tests, and (3) finding undefined references and asking the LLM to resolve them.

2.4.1 Gathering Information About the Code Change. For an LLM to generate effective tests, it needs to understand the context of the code change. To this end, Testora gathers two kinds of information. First, the approach extracts the names of all functions that are modified by the PR. Providing these names is important for the LLM to understand which parts of the project to exercise in the generated tests. To extract the function names, the approach uses a lightweight static analysis: For each hunk in the code change, the approach parses the new version of the code file into an AST, finds the location of the changes in the AST, and identifies the inner-most function that encloses this location. The approach then concatenates the module name of the modified file and the function name to get the fully qualified function name, which is important for the LLM to understand how to access the modified function. If there is no surrounding function or if a hunk spans multiple functions, then the approach prompts the LLM without mentioning specific function names.

Second, the approach extracts the diff of the code change. Specifically, we extract two versions of the diff. On the one hand, we extract a *full diff*, which contains all modifications made by the PR, including any edits of documentation. On the other hand, the approach extracts a *filtered diff*, which includes only modifications of files that belong to the main source code, as defined in Section 2.3.

2.4.2 Querying an LLM to Generate Tests. Given the gathered information about the code change, the approach queries an LLM to generate tests that exercise the modified code. Our prompt includes the fully qualified names of modified functions and one of the two variants of the diff. The prompt asks the LLM to generate usage examples that expose behavioral differences introduced by the provided diff. We further specify that the usage examples must be self-contained Python code that uses only the publicly exposed APIs of the project and does not depend on non-deterministic behavior, such as random number generation. To enable the approach to understand and compare the runtime behavior triggered by a

[1] We consider C/C++ files because some Python projects have performance-critical parts implemented in C/C++.

usage example, we ask the LLM to include `print` statements for relevant values, such as the return value of an API function. We ask to generate ten usage examples that cover normal usage scenarios and ten usage examples that focus on corner cases. To increase the diversity of the generated tests, we query the LLM twice: once with the full diff and once with the filtered diff. This is based on the observation that asking the LLM in slightly different ways produces different tests.

We clean and de-duplicate tests by removing any tests that invoke "private" functions. The Python language does not have any notion of "private" functions, but most projects follow the convention to prefix functions not intended for client usage with an underscore. Based on this convention, our approach discards tests that call such a function. The rationale is that the tests should only exercise the public API of the project, as this is what users of the project interact with. Finally, the approach discards duplicates, i.e., syntactically identical tests.

2.4.3 Finding and Fixing Undefined References. A common problem we observed in preliminary experiments is that some generated test cases refer to variables or functions that are not defined in the test case itself. This problem is usually easy to fix, e.g., by adding an import statement or by defining a missing variable, but doing this manually for each test case would be too time-consuming. Instead, we use a lightweight static analysis to find any undefined references, and then ask the LLM to refine the test case. The prompt provides the initial test case, a list of identifiers that are referred to, but that are not properly defined, and then asks the LLM to provide a refined test case that resolves the undefined references. Our refinement step is similar to conversational approaches to program repair [64] and test assertion generation [23], but uses a lightweight, AST-based static analysis, instead of executions, for feedback.

2.5 Comparison of Behavior Before and After the Change

Given the set of generated test cases, the next step of Testora is to compare the behavior of the original and the modified code. To this end, the approach starts by building the target project twice: at the commit just before the PR and at the commit with the PR applied. Each commit is built in a containerized environment (based on Docker), providing two environments env_{old} and env_{new}. The motivation for using containers is to isolate these environments from each other and from the environment that Testora itself is running in.

For each generated test case c, the approach executes the test in both the old and the new environment. These executions result in two outputs: $o_{old} = execute(c, env_{old})$ and $o_{new} = execute(c, env_{new})$ The output from a test execution consists of all console output and any stack traces generated by exceptions. Many of the LLM-generated tests contain `print` statements, which enables the approach to inspect the behavior of the code from its output.

Based on the two outputs o_{old} and o_{new}, Testora compares the behavior of the old and the new code. If the outputs are exactly the same, i.e, $o_{old} = o_{new}$, then the test execution has not exposed any behavioral difference, and the approach moves on to the next test. In contrast, if the outputs differ, i.e., $o_{old} \neq o_{new}$, then Testora performs several steps aimed at focusing on relevant differences and at reducing the test case to a minimal example that still exposes the difference:

(1) Check whether both the old and the new version have failed with an exception.[2] If yes, we discard this test case because the test case likely raises a legitimate exception, e.g., caused by using the APIs of the target project incorrectly, which is not a regression.
(2) Validate the difference between o_{old} and o_{new} by re-executing the test with both the old and the new code. This step is to avoid flaky tests, which may cause non-deterministic differences in output. If the difference disappears when re-executing a test, the approach discards the test.
(3) Iteratively reduce the test case by removing one line at a time at the end and by checking if the outputs still differ. This process continues until no difference is found anymore or until the entire test case has been removed. The rationale for simply removing lines at the end, instead of using more sophisticated program reduction techniques [25, 39, 68], is that tests are mostly linear programs without control flow statements. All further steps are performed on the reduced test case.
(4) Check if the difference is still present in the latest commit of the project, to avoid reporting regressions that have already been fixed. To this end, the approach builds the target project into a third environment env_{latest} at the latest commit of the main branch, and re-executes the test case, which results in $o_{latest} = execute(c, env_{latest})$. If $o_{new} \neq o_{latest}$, then the developers have further changed the code affected by the PR after the PR had been applied. In this case, the approach discards the test case, as the observed behavioral difference is not relevant anymore. If Testora is used continuously for every PR before it gets merged, this step can be skipped, as the approach finds regressions before they even get merged into the main branch.

Any behavioral difference that remains after these steps is considered to be potentially unintended, and the tuple (c, o_{old}, o_{new}) is passed on to the final step of Testora for classification.

2.6 Classification of Behavioral Differences

The final step of our approach is to determine, for any behavioral difference found, whether the difference is intended or unintended. This step involves two key challenges: First, not every behavioral difference is a problem. The reason is that many PRs are supposed to modify some behavior, e.g., to fix a bug or to add a new feature. The challenge is to distinguish between intended and unintended differences. For example, the behavioral difference shown in Figure 1 is unintended, as the change in the output of the `differential_entropy` function is not intended by the PR. In contrast, consider the PR shown in Figure 3, which aims to improve the precision of the `logsumexp` function in the `scipy.special` module. The PR also causes a behavioral difference, but unlike the previous example, the difference is intended, as the PR specifically aims to improve the precision of the function. Second, some behavioral differences may not be directly intended by the PR, but at the same

[2] Even if both versions raise the same kind of exception, the concrete outputs o_{old} and o_{new} often differ, e.g., because line numbers in the stack trace may differ.

PR title:
ENH: `special.logsumexp`: improve precision when one element is much bigger than the rest

PR description:
... can lose precision when one element is much bigger than the rest, especially when the exponential of it is close to 1. This improves the precision as described in the issue. ...

Testora-generated test case:

```
import numpy as np
from scipy.special import logsumexp

a = np.array([1.0, 2.0, 3.0])
result = logsumexp(a)
print(result)
```

Output before PR: 3.4076059644443806

Output after PR: 3.40760596444438

Testora's classification: *Intended behavioral change*
While both outputs are very close and essentially represent the same value, the second output has fewer decimal places, which could be seen as a minor change in the representation of the result due to numerical precision improvements. The change is intended by the developer, as the pull request specifically states that it aimed to improve precision for the `logsumexp` function when working with numbers that have a wide range.

Figure 3: Example of an intended behavioral change.

time, are so minor that they are irrelevant in practice. Typical examples include minor differences in error messages, e.g., caused by different object addresses, or minor changes in formatting.

Our key insight is to address these challenges by providing an LLM with the natural language information provided in the PR, such as the PR title, description, and commit messages. The idea is that the intention of the code change is often documented in these natural language artifacts, and that we can use this information to determine whether a behavioral difference is intended or unintended. That is, we use the natural language information as an oracle for regression testing.

To classify whether a behavioral difference is intended, Testora performs two sub-steps, (i) gathering relevant contextual information and (ii) a multi-question, LLM-based classifier, which we describe in the following.

2.6.1 Gathering Relevant Context. To enable the LLM classifier to take an informed decision, we provide it with relevant context about the code change. The context includes the following information:

- The name of the project.
- The fully qualified names of the functions that are modified by the PR.
- The title t of the PR.
- The description d of the PR.
- The diff Δ of the code change.
- The commit message(s) m_c associated with the code diff.
- Any discussion comments m_d associated with the PR.
- The test case c that exposes the behavioral difference.
- The outputs o_{old} and o_{new} of the test execution with the old and the new code.
- The docstrings (if available) of all functions invoked in the test case.

To extract the PR-related information, we query the GitHub API. The docstrings of functions are retrieved using the Language Server Protocol (LSP). Retrieving docstrings is important for the LLM to understand the pre-conditions and the intended behavior of the functions invoked in the test case.

2.6.2 Multi-Question, LLM-Based Classifier. Based on the gathered context, Testora prompts the LLM to determine whether a behavioral difference is intended or unintended. One possible approach, which we call the *single-question classifier*, would be to ask the LLM a single question, e.g., "Is the behavioral difference intended?". In initial experiments, we found that the single-question classifier commonly gives suboptimal answers. For example, the classifier may report a behavioral difference as unintended, but the test case violates a pre-condition of the tested API, which implies that the behavioral difference is irrelevant in practice.

To address this issue, we present a *multi-question classifier* that asks the LLM multiple questions designed to avoid common pitfalls of the single-question classifier. Testora prompts the LLM with the following five questions:

(1) Is the different output a noteworthy change in behavior, such as a completely different value being computed, or is it a minor change, such as a change in a warning/error message or a change in formatting?
(2) Is the different output likely due to non-determinism, e.g., because of random sampling or a non-deterministically ordered set?
(3) Does the usage example refer only to public APIs of the project, or does it use any project-internal functionality?
(4) Does the usage example pass inputs as intended by the API documentation, or does it pass any illegal (e.g., type-incorrect) inputs?
(5) Does the different output match the intent of the developer of the pull request?

We instruct the LLM to answer the questions in a structured, JSON-based format. For each question, the LLM is asked to first provide its thoughts, i.e., the reasoning behind the answer, and then to provide the actual answer. The rationale for this structure is to force the LLM to reason about the question before giving an answer, which we found to improve the quality of the answers in preliminary experiments.

The answers received by the multi-question classifier are a 5-tuple $(a_1, a_2, a_3, a_4, a_5)$. Testora considers a behavioral difference as unintended if the LLM determines the difference to be noteworthy (a_1), deterministic (a_2), triggered via public APIs (a_3) using legal inputs (a_4), and causing an unintended difference in output (a_5). Otherwise, i.e., if any of the five answers suggests the behavioral change to be expected or irrelevant, the approach considers the behavioral difference as intended. If the approach concludes that the difference is unintended, it provides the test case c and the outputs o_{old} and o_{new} to the developer, along with the natural language

Table 1: Projects used in the evaluation.

Name	Application domain	Stars	Closed PRs
keras	Deep learning	62,500	7,752
marshmallow	Serialization and deserialization	7,100	1,198
pandas	Data analysis and manipulation	44,500	33,249
scipy	Scientific computing	13,300	11,572

thoughts e by the LLM explaining why the behavioral difference is unintended.

For example, the approach classifies the behavioral differences in Figures 1 and 3 as unintended and intended, respectively. The bottom of the figures show the natural language explanations provided by the LLM.

3 Evaluation

Our evaluation applies Testora to real-world Python projects to answer the following research questions (RQs):

- *RQ1: Effectiveness at finding real-world problems.* How effective is the approach at detecting regressions in real-world projects?
- *RQ2: Effectiveness of test generation.* How effective is the approach at exercising changed code and at revealing behavioral differences?
- *RQ3: Accuracy of classifier.* How accurate is the approach at distinguishing between intended and unintended behavioral differences?
- *RQ4: Costs.* What are the computational costs of the approach?

3.1 Experimental Setup

Target projects. Even though the approach is mostly language-agnostic, we focus on Python projects in our evaluation. We select four real-world Python projects based on the following criteria:

- Popularity: We discard projects with less than 5,000 stars on GitHub, aiming to target projects that are widely used and well maintained.
- Libraries: To ensure that the projects have a well-defined API that can be tested via generated tests we focus on libraries, i.e., we discard projects that are end-user applications, frameworks, and tutorial-style projects.
- PRs: We consider a project only if it has at least 1,000 closed PRs. This criterion ensures that the targeted projects follow a PR-based development process and have a sufficient number of PRs to evaluate the approach.
- Setup and requirements: Since the approach requires building and installing the target projects at different commits, we create a Docker container for each project. We discard projects where we were unable to create a container that reliably builds and installs the project.
- Diversity: To cover different application domains, we discard projects that are very similar to each other, such as multiple machine learning libraries.

Table 1 provides an overview of the four selected projects. The number of stars and closed PRs shown in the table are as of February 4th, 2025. All four projects are mature, widely used in the Python ecosystem, and provide sufficient complexity to make automated regression testing challenging.

Pull requests. For RQ1, we apply the approach during several testing campaigns to batches of the most recent PRs of the target projects. This process was repeated several times over the course of about six months. During a real-world deployment, Testora would continuously analyze PRs as they are created, but for the evaluation in this paper, we analyze PRs in batches to reduce human time and computational costs. Addressing RQ3 requires a labeled dataset, where each entry consists of a PR, a test case generated by Testora that exposes a behavioral difference, and a label indicating whether the behavioral difference is intended or unintended. We create such a dataset by manually inspecting a randomly sampled subset of the PRs considered in RQ1 where the approach found a behavioral difference. The dataset consists of 164 labeled entries, of which 139 are intended and 25 are unintended behavioral changes. For RQ2 and RQ4, we apply the approach to a systematically gathered range of up to 500 PR numbers per target project. Specifically, we consider a range of consecutive PRs from each target project, starting from a PR number n, where n was the first PR after a specific date (October 2, 2024). For each i in n to $n + 500$, we consider it as a target PR if i is the number of a PR (and not of an issue, as issues use the same numbering scheme) and the PR was merged. This process results in a total of 1,274 PRs across the four projects.

LLMs. The approach interacts with an LLM to generate tests and to classify behavioral differences exposed by these tests. As a default, we use GPT-4o-mini from OpenAI as the LLM. This decision is based on the results of preliminary experiments that showed GPT-4o-mini to provide a good trade-off between effectiveness and costs. For the important tasks of classifying behavioral differences, RQ3 also evaluates Testora with the GPT-4o and DeepSeek-R1 models.

Implementation and hardware. The approach is implemented in Python 3 in about 4,600 lines of non-empty, non-comment Python code. We use the libCST library and the built-in Python AST module to parse and analyze the Python code of the target projects, and also to check for undefined references in generated test cases. To interact with the GitHub API and the git repositories of the target projects, we use the PyGithub and GitPython libraries, respectively. The implementation uses the Jedi Language Server via the multispy library [1] to retrieve the docstrings of functions called in the generated tests. The target projects are installed in separate Docker containers, and we use the Python docker library to interact with the containers. In particular, all test executions are performed in these Docker containers to ensure that the tests run in a clean environment and do not interfere with the host system.

All experiments are performed on a server with 48 Intel Xeon CPUs with 2.2GHz and 256GB of RAM running Ubuntu 22.04. We run multiple experiments in parallel to reduce the time needed to process all PRs. To distribute PR analysis tasks across multiple instances of the approach, we use a MySQL database to store the PRs to check and the results of Testora.

```python
import jax.numpy as jnp
from numpy import nan
from keras.src.backend.jax.math import in_top_k

r = in_top_k(targets=jnp.array([1, 0]),
    predictions=jnp.array([[.1, nan, .5], [.3, .2, .5]]), k=2)
print(r)  # [False  True] vs. [True  True]
```

Figure 4: Regression in keras (#1 in Table 2), leading to incorrect output.

```python
from scipy.fft import hfftn

x = [[[1.0, 1.0], [1.0, 1.0]], [[1.0, 1.0],
      [1.0, 1.0]], [[1.0, 1.0], [1.0, 1.0]]]
hfftn(x)  # no exception vs. exception
```

Figure 5: Regression in scipy (#18 in Table 2), leading to an exception.

3.2 RQ1: Effectiveness at Finding Real-World Problems

To validate the effectiveness of Testora at finding problems in real-world projects, we apply it to PRs of the target projects. When the approach detects a potential regression, we manually inspect the PR and the report generated by our approach. Based on the inspection, each report by Testora belongs to one of three possible outcomes: regression, coincidental fix, or false positive. We discuss the first two cases in the following, and will evaluate false positives in RQ3. Table 2 provides an overview of the results.

3.2.1 Regressions. We categorize a report produced by Testora as a *regression* if the PR introduces a bug that has remained unnoticed by the developers by the time of merging the PR. In this case, we check whether the problem still exists in the latest commit of the project, and if yes, report the problem to the developers. In total, the approach has detected 19 regressions, out of which 13 had still been present in the latest commit of the project by the time that we inspected the PRs. We reported these 13 regressions to the developers. As of July 2025, 11/13 of them have been confirmed, and 9 out of the 11 confirmed problems have been fixed in reaction to our report. 2/13 of the reported regression bugs are still open. 1/13 issue has been confirmed as a bug but eventually closed as "won't fix" by the developers because the problem is unlikely to occur in practice while fixing it would affect a lot of code. The remaining 6/19 PRs had already been fixed independently by the developers by the time that we checked the PRs. If a technique like Testora had been used continuously during development, all 19 regressions could have been detected earlier and fixed before the PR was even merged into the code base.

All detected regressions were missed by the regression test suites that are executed as part of the continuous integration pipelines of the studied projects. This is despite the fact that the existing tests are very comprehensive. For example, for scipy (where Testora finds most regressions), executing all tests takes multiple CPU hours.

Examples. Beyond the example in Figure 1, we provide additional examples of regressions detected by Testora. Figure 4 shows a test case that exposes a regression bug in the keras project. The PR, entitled "Faster `in_top_k` implementation for Jax backend", intends to improve performance for the Jax backend of keras. Unfortunately, the optimized code incorrectly considers `nan` to be a large probability, which causes the `in_top_k` function to return incorrect results. Figure 5 shows a regression bug in scipy, which causes the `fft.hfftn` function to raise an exception when called with a list argument. The PR that causes the bug intends to introduce GPU support for specific computations, which unfortunately, breaks some of the existing functionality.

```python
import pandas as pd

df = pd.DataFrame({'A': ['foo', 'bar', None], 'B': [1, 2, 3]})
grouped = df.groupby('A', dropna=False)
print(len(grouped))  # exception vs. 3
```

Figure 6: Bug in pandas that was coincidentally fixed by a PR meant to improve performance (#13 in Table 2).

3.2.2 Coincidental Fixes. In addition to regressions, Testora also detects PRs intended to improve the code in some way, but that – unknowingly to the developers – also fix a bug. We call such cases *coincidental fixes*. Overall, Testora detects 11 coincidental fixes, as listed in Table 2. Because the bug is fixed by the PR, we generally do not report the problem to the developers. However, we document coincidental fixes in our evaluation to show that the approach can also find bugs that have remained unnoticed by the developers. Project maintainers using a technique like Testora could benefit from knowing about coincidental fixes, e.g., to decide which PRs to backport to older releases or to help with documenting fixed bugs in the release notes of the project.

Examples. Figure 6 shows a coincidental fix in the pandas project. The PR intends to improve performance of the `DataFrame.groupby` function. However, the revised code also fixes a bug that causes the function to raise an exception when grouping by a column that contains `None` values.

Another example (#2 in Table 2) is a PR in the marshmallow project, which is described as "minor refactor", but coincidentally fixes an incorrect output of a utility function that returns the arguments of a given callable.

3.3 RQ2: Effectiveness of Test Generation

In addition to evaluating the end-to-end effectiveness of Testora in RQ1, the following two research questions evaluate the two main components of the approach in more detail. RQ2 focuses on the effectiveness of the test generation component. To this end, we apply Testora to the 1,274 PRs listed in the "Total" column of Table 3. As described in Section 2.3, Testora ignores some PRs, e.g., PRs unlikely to introduce regressions, such as changes to test files or documentation-only changes. This filtering affects 749/1,274 PRs, as shown in the "Ignored" column of Table 3, leaving 525 PRs for Testora to analyze in detail. For the 525 PRs that Testora analyzes in detail, we evaluate the effectiveness of the test generation component. As shown in the last column of Table 3, the approach finds a behavioral difference in 100/525 PRs, i.e., in 19% of the PRs.

Table 2: Real-world regressions and coincidental fixes detected by Testora.

Id	Project	PR	Description	Kind	Issue	Status
1	keras	19814	Incorrect handling of `NaN` in `in_top_k` function in Jax backend	Regression	19995	Confirmed and fixed
2	marshmallow	1399	Refactoring coincidentally fixes incorrectly returned function arguments	Coincidental fix	–	Nothing to report
3	marshmallow	2698	Incorrectly formatted timezone offset	Regression	–	Fixed independently
4	marshmallow	2699	`get_fixed_timezone` raises exception on floating point timezone offset	Regression	–	Fixed independently
5	marshmallow	2700	`missing` argument of `Field` class gets ignored	Regression	–	Fixed independently
6	marshmallow	2701	Passing `missing` argument to constructor of `Field` class raises exception	Regression	–	Fixed independently
7	pandas	55108	Intended to improve performance, but introduced bug that causes wrong output of `Index.difference` function	Regression	58971	Confirmed and fixed
8	pandas	56841	Intended to improve performance, but introduced a bug that causes wrong output of `Index.join` function	Regression	58603	Reported
9	pandas	57034	`Series.combine_first` produces wrong output for series containing `None`	Regression	58977	Confirmed
10	pandas	57046	Intended to fix a bug in `groupby.idxmin` related to extreme values, but also fixes bug triggered by `NaN`	Coincidental fix	–	Nothing to report
11	pandas	57205	Intended to improve performance, but also fixes a bug in handling `None` values in `DataFrame` constructors	Coincidental fix	–	Nothing to report
12	pandas	57399	`interval_range` ignores type of `start` parameter	Regression	58964	Reported
13	pandas	57595	Intended to improve performance, but also fixes a bug where `DataFrame.groupby` returns an invalid value	Coincidental fix	–	Nothing to report
14	pandas	58376	Intended to improve performance, but also fixes a bug related to calling `RangeIndex.searchsorted` with a negative step	Coincidental fix	–	Nothing to report
15	pandas	60461	Intended to improve performance, but also fixes wrong return value of `construct_1d_object_array_from_listlike`	Coincidental fix	–	Nothing to report
16	pandas	60483	Backport of PR 60461 (see above)	Coincidental fix	–	Nothing to report
17	pandas	60538	Adding `DataFrames` with misaligned `MultiIndex` produces `NaN` despite `fill_value=0`	Regression	60903	Confirmed and fixed
18	scipy	19263	`fft.hfftn` fails on list inputs	Regression	21207	Confirmed and fixed
19	scipy	19428	Should raise exception when `stats.levene` is called with a single sample	Regression	–	Fixed independently
20	scipy	19680	`stats.shapiro` raises an error given lists of extreme integers	Regression	21205	Confirmed, won't fix
21	scipy	19776	Intended to vectorize computation, but also fixes bug triggered by passing `inf` to `stats.rankdata`	Coincidental fix	–	Nothing to report
22	scipy	19853	Rewriting of internal functions coincidentally fixes bug that gave incorrect mean value of empty sparse matrix	Coincidental fix	–	Nothing to report
23	scipy	19861	Bug fix also improves robustness of saving matrices into `.mat` files	Coincidental fix	–	Nothing to report
24	scipy	20089	`special.hyp2f1` gives wrong result for extreme inputs	Regression	20988	Confirmed and fixed
25	scipy	20751	Value returned by `stats.bartlett` is negative when exact value would be zero	Regression	21152	Confirmed and fixed
26	scipy	20974	`stats.combine_pvalues` gives result with wrong dimensionality	Regression	21106	Confirmed and fixed
27	scipy	21036	Intended to add array API support, but also fixes a bug that caused `stats.tsem` to incorrectly return `inf`	Coincidental fix	–	Nothing to report
28	scipy	21076	`stats.differential_entropy` with integer dtype gives incorrect result	Regression	21192	Confirmed and fixed
29	scipy	21553	Performance improvement leads to incorrect output of `expm`	Regression	–	Fixed independently
30	scipy	21768	Slicing sparse matrix with `None` gives result different from numpy's matrices	Regression	22458	Confirmed and fixed

Table 4 provides more detailed results of the test generation and testing process. The table show the minimum/average/maximum values per PR, across the 525 checked PRs. At first, we consider the number of generated tests. In principle, the combined answers of the LLM consist of $2 \times 2 \times 10 = 40$ tests (Section 2.4). In practice, there may be fewer tests because the LLM occasionally provides fewer tests than requested and because we de-duplicate tests. During our experiments, we find that the approach generates between 27 and 32 unique test cases, on average per PR.

Next, we study whether the generated tests cover the code changed by the PR, which is a prerequisite for finding behavioral differences. We consider a generated test to "have diff coverage" if

Table 3: Pull requests analyzed for RQ2 and RQ4.

Project	Pull requests			
	Total	Ignored	Checked	Behavioral difference found
keras	271	111	160	0
marshmallow	138	73	65	8
pandas	439	296	143	35
scipy	426	269	157	57
Total	1,274	749	525	100

Table 4: Test generation results (min/avg/max per PRs).

Project	Tests		Test executions	
	Generated	With diff. cov.	Total	Non-failing
keras	2/27/40	0/14/40	4/76/160	0/48/156
marshmallow	9/32/40	0/26/40	18/96/530	0/83/528
pandas	2/31/40	0/18/40	4/107/1098	0/88/1059
scipy	1/30/40	0/18/40	2/135/1481	0/101/1073

it covers at least one line of the changed code. As shown in Table 4, between 14 and 26 tests, on average per PR, have diff coverage. That is, the majority of the generated tests (52%–81%) are successful at exercising changed code.

Finally, the last two blocks of Table 4 show the number of test executions performed by Testora when analyzing a single PR. These numbers are higher than the number of generated tests because the approach re-executes tests while reducing tests that expose a behavioral difference and to filter flaky tests by validating each behavioral difference. Overall, the approach executes between 76 and 135 tests, on average per PR. The majority of these tests (63%–86%) are non-failing, i.e., they do not raise an exception.

3.4 RQ3: Accuracy of Classifier

Besides the test generation component, as evaluated in RQ2, the classifier that distinguishes between intended and unintended behavioral differences is crucial for the effectiveness of Testora. The following evaluates the accuracy of the classifier using different LLMs and prompting techniques. We apply the classifier to the labeled dataset described in Section 3.1. As the dataset is imbalanced in the number of intended and unintended behavioral differences, we evaluate the accuracy of the classifier by measuring precision, recall, and F1 score. Precision here means the percentage of behavioral differences classified as unintended that are actually unintended. Recall means the percentage of unintended behavioral differences that the approach recognizes as unintended. The F1 score is the harmonic mean of precision and recall.

We evaluate the accuracy of the classifier using different LLMs and prompts. As LLMs, we consider two state-of-the-art commercial LLMs, GPT-4o and its more economic variant, GPT-4o-mini, as well as a state-of-the-art open-source LLM, DeepSeek-R1 (671B).

Table 5: Accuracy of classifier (bold = default configuration).

LLM	Single-question classifier			Multi-question classifier		
	Precision	Recall	F1	Precision	Recall	F1
GPT-4o-mini	49%	80%	61%	**55%**	**67%**	**60%**
GPT-4o	80%	64%	71%	71%	42%	53%
DeepSeek-R1	69%	36%	47%	83%	42%	56%

We evaluate the classifier using two different prompts: the multi-question classifier described in Section 2.6.2 and a simpler, single-question classifier that asks the LLM only whether the behavioral difference is intended by the PR.

Table 5 shows the accuracy of the classifier using the different LLMs and prompts. We make three observations. First, and perhaps most importantly, the results are relatively stable across different models and different variants of the classifier. That is, the approach does not depend on a specific LLM or prompting technique, but generalizes well to the considered (and probably also future) models and prompts. Second, no clear winner emerges among the considered LLMs: While the overall best configuration, in terms of F1-score, is GPT-4o with the single-question classifier, both GPT-4o-mini and DeepSeek-R1 outperform GPT-4o when using the multi-question classifier. To keep the costs of using Testora manageable, we use GPT-4o-mini as the default model. Third, when comparing the two prompting techniques, we again see a diverse set of results: The single-question classifier outperforms the multi-question classifier in terms of F1-score for GPT-4o, but the multi-question classifier outperforms the single-question classifier for DeepSeek-R1. For our default model, GPT-4o-mini, the multi-question classifier offers approximately the same F1 score as the single-question classifier (60% vs. 61%), but offers a better balance between precision and recall, which is why we use the multi-question classifier as the default in Testora.

3.5 RQ4: Costs

Our final research question evaluates the computational costs imposed by Testora. We consider two types of costs: monetary costs imposed by LLM queries, and the time taken by the approach. The monetary costs are calculated based on the number of input and output tokens used by the LLM. On average per checked PR, the approach consumes 9,440 tokens, of which 5,818 are input tokens and 3,622 are output tokens. Figure 7 shows a breakdown of the tokens consumed during different steps of Testora. We find that test generation and classification consumes most tokens, whereas test refinement has negligible costs, and test execution (by design) does not consume any tokens. The tokens used by the classifier show several outliers, which are due to long diffs and long discussions of PRs. Yet, despite these outliers, the overall costs remain manageable. Based on the pricing of OpenAI's GPT-4o-mini model as of February 17, 2025, the total token consumption results in monetary costs of $0.003 per checked PR. Given the high human effort caused by regressions that remain unnoticed, we consider this cost acceptable for real-world deployment at a large scale.

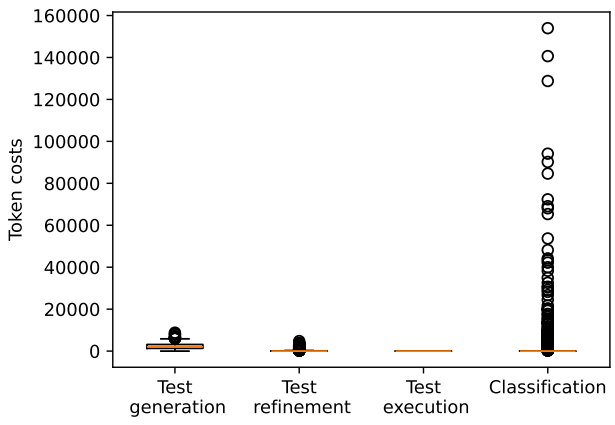


Figure 7: LLM tokens used per PR.

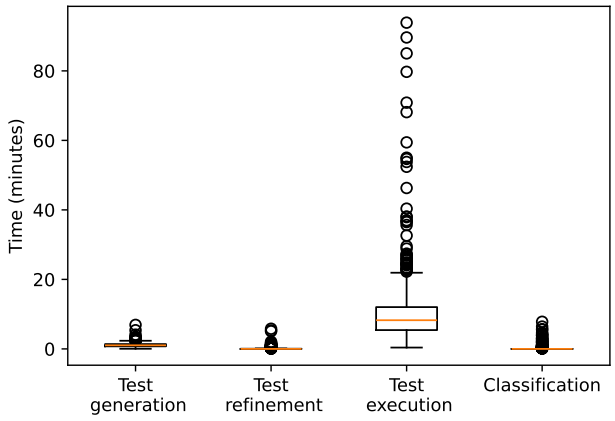


Figure 8: Time taken per PR.

Besides LLM costs, we also evaluate the time taken by the approach. On average per checked PR, the approach takes 12.3 minutes to analyze a PR. As shown in Figure 8, the time is mostly spent on test execution. A relatively large fraction of this time is for compiling the target project at a specific commit, which is a prerequisite for running the generated tests. To put the time of 12.3 minutes in perspective, we compare it to the time taken by the continuous integration (CI) effort performed by an average non-free repository that uses GitHub workflows [9]: Testora's time corresponds to 39.5% of the CI time spent when creating a new pull request and to 43.3% of the CI time spent each time a developer pushes new commits into a repository. While this was not our focus, we believe that the time could be further reduced, e.g., by pre-computing Docker images compiled at specific commits and by parallelizing the execution of different generated tests.

4 Discussion

4.1 Limitations

Testora relies on automated test generation, which is most suitable for projects that have a well-defined API amenable to unit-level testing. Extending our idea to projects with a more complex interface, e.g., a graphical user interface, is left for future work. Another limitation is that very complex code changes, e.g., where the diff spans many files, are likely to be challenging for the approach.

4.2 Threats to Validity

Our evaluation is based on a set of four projects, which may not be representative of all software projects. We mitigate this threat by selecting projects from different domains and by evaluating the approach on hundreds of PRs. Another threat is that different LLMs may provide different results, which we mitigate by evaluating the classifier with three different LLMs. The manual labeling to establish a ground truth for RQ3 is also a potential threat to validity, as it may be biased. To mitigate this threat, the annotator is a senior researcher, who made two passes over all 164 examples: one pass to assign an initial label, and another pass to double-check all labels and ensure consistency across all examples. Another potential bias is that the examples to label are randomly sampled from the set of PRs where Testora found a behavioral difference, and hence, some projects are represented more frequently than others. Finally, our results are limited to Python projects, and while the general idea of Testora is likely to extend beyond Python, we do not have empirical evidence for this.

5 Related Work

Test generation and regression testing. To complement manually written tests, there are various approaches for automated test generation, including feedback-directed, random test generation [46], symbolic and concolic execution [10, 30, 56], and search-based test generation [18]. More recently, the community is focusing increasingly on LLM-based test generation, e.g., in combination with a search-based approach [32], for fuzzing libraries [13] and entire applications [63], for bug reproduction [28], for unit-level test generation [50, 55, 65, 67], and for test augmentation [3]. In principle, these approaches could be integrated into the first step of Testora, but our approach differs by generating tests that target a specific code change, instead of trying to increase overall coverage. Our work also relates to work on selecting and prioritizing regression test cases [15, 22, 66], but differs by generating new tests for the given code change.

Test oracle problem. Effective testing requires not only suitable test inputs, but also a way to identify unexpected behavior, also known as the test oracle problem [7]. One line of work to address this problem infers oracles from API documentation, e.g., in the form of exception oracles [19] or metamorphic relations [8], or based on regular expressions [42]. Another line of work uses deep learning and LLMs to generate likely oracles for a given test prefix [14, 23, 27, 45, 61]. Unlike these approaches, Testora focuses on code changes and proposes the novel idea of exploiting natural language associated with a code change as a basis for a regression test oracle.

Reasoning about code changes and code differences. Differential testing [38] compares the behavior of multiple implementations and has been applied, e.g., to compilers [6], debuggers [31], symbolic execution engines [29], and quantum computing platforms [60]. Our work differs by comparing two versions of the same code, instead of two independently created implementations. To compare two versions of the same code, prior work has explored change-oriented symbolic execution [37] and validating code changes via learning-guided execution [21, 58]. These approaches share the idea of exposing a behavioral difference between an older and a newer version of the code, but unlike Testora, do not address the problem of determining whether a detected difference is intended. Just-in-time defect prediction tries to predict the general likelihood that a commit introduces a bug [26, 40, 59, 69]. Our approach goes further by comparing the behavioral change to the developer's intent, as expressed in the PR description, and by providing a concrete test case that exposes a likely regression. The TraceJIT approach [40] is particularly related due to its dynamic features. A key difference from our work is that Testora is not based on generic features (e.g., the extent to which a dynamic trace changes), but instead checks any behavioral differences against the developer-expressed intent of the PR. Work on untangling commits [34, 49] is related in that it could serve as a pre-processing step before running Testora, enabling our approach to handle more complex, entangled code changes. The DCI approach [12] also uses tests to reason about behavioral changes induced by a code change, but unlike Testora, assumes that a test covering the code change already exists. Once a bug is known, techniques for retrospectively determining the bug-inducing commit have been proposed [4, 57]. Finally, other approaches help reason about code changes without specifically targeting regressions, e.g., by augmenting diffs with runtime information [17] and via a unified data representation of code changes [62].

Natural language and code. Natural language embedded in code has been used to detect bugs [52] and name inconsistencies [44], to infer type annotations [24, 36, 51], and to predict formal specifications of code [16, 70]. Other work summarizes code into comments [2, 20, 43, 72], detects inconsistencies between code and comments [47, 53], automatically updates comments when code changes [48], and propagates comments across related code elements [71]. Summarizing the likely intent of code into natural language can also support automated issue fixing [54]. Others have proposed to detect potential bugs by generating alternative implementations from a natural language specification [33, 35]. Like this work, the above techniques share the observation that natural language associated with code provides a rich resource. Unlike prior work, we leverage this resource as the basis for a regression oracle and integrate it into an automatic regression testing technique.

6 Conclusion

Regression testing is a crucial part of software development, but it is challenged by the need to identify unintended behavioral changes. We present Testora, an approach that (i) generates regression tests targeted at code changed by a PR, (ii) executes these tests to identify behavioral differences introduced by the PR, and (iii) classifies the behavioral differences exposed by the tests as intended or unintended. A key contribution of Testora is to use natural language information associated with a code change as a basis for a regression testing oracle. Our evaluation on real-world projects shows that Testora is effective at finding 19 real-world problems, while imposing acceptable costs of 12.3 minutes of computational time and $0.003 of monetary costs per checked PR. We envision our approach to serve as a useful tool for developers to continuously and early on detect regressions and coincidental bug fixes. Initial feedback from developers is very positive. For example, the developers of scipy asked us to apply Testora to more PRs and are interested in applying Testora to their project in the future.

Data Availability

Code and data associated with this work are available: https://github.com/michaelpradel/Testora

Acknowledgments

This work was supported by the European Research Council (ERC, grant agreements 851895 and 101155832) and by the German Research Foundation within the DeMoCo and QPTest projects. The author also thanks Koushik Sen and Miryung Kim for hosting him during his sabbatical at UC Berkeley und UCLA, respectively, during which parts of this work were conducted.

References

[1] Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu Lahiri, and Sriram Rajamani. 2023. Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 32270–32298. https://proceedings.neurips.cc/paper_files/paper/2023/file/662b1774ba8845fc1fa3d1fc0177ceeb-Paper-Conference.pdf

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL*. 4998–5007. doi:10.18653/v1/2020.acl-main.449

[3] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. In *FSE*, Vol. abs/2402.09171. doi:10.48550/ARXIV.2402.09171 arXiv:2402.09171

[4] Gabin An, Jingun Hong, Naryeong Kim, and Shin Yoo. 2023. Fonte: Finding Bug Inducing Commits from Failures, In ICSE. *arXiv preprint arXiv:2212.06376*.

[5] Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. 2022. Nessie: Automatically Testing JavaScript APIs with Asynchronous Callbacks. In *ICSE*.

[6] Gergö Barany. 2018. Finding missed compiler optimizations by differential testing. In *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*. 82–92.

[7] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525.

[8] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Antonio Carzaniga. 2021. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *Journal of Systems and Software* (2021).

[9] Islem Bouzenia and Michael Pradel. 2024. Resource Usage and Optimization Opportunities in Workflows of GitHub Actions. In *International Conference on Software Engineering (ICSE)*.

[10] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 209–224.

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight,

Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[12] Benjamin Danglot, Martin Monperrus, Walter Rudametkin, and Benoit Baudry. 2020. An approach and benchmark to detect behavioral changes of commits in continuous integration. *Empirical Software Engineering* 25, 4 (2020), 2379–2415.

[13] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, René Just and Gordon Fraser (Eds.). ACM, 423–435. doi:10.1145/3597926.3598067

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

[15] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. 2002. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering* 28, 2 (2002), 159–182.

[16] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions?. In *FSE*.

[17] Khashayar Etemadi, Aman Sharma, Fernanda Madeiral, and Martin Monperrus. 2023. Augmenting Diffs With Runtime Information. *IEEE Trans. Software Eng.* 49, 11 (2023), 4988–5007. doi:10.1109/TSE.2023.3324258

[18] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. 416–419.

[19] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 213–224. doi:10.1145/2931037.2931061

[20] David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. 2020. Code to Comment "Translation": Data, Metrics, Baselining & Evaluation. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 746–757. doi:10.1145/3324884.3416546

[21] Lars Gröninger, Beatriz Souza, and Michael Pradel. 2025. ChangeGuard: Validating Code Changes via Pairwise Learning-Guided Execution. In *International Conference on the Foundations of Software Engineering (FSE)*.

[22] Mary Jean Harrold, David S. Rosenblum, Gregg Rothermel, and Elaine J. Weyuker. 2001. Empirical Studies of a Prediction Model for Regression Test Selection. *IEEE Trans. Software Eng.* 27, 3 (2001), 248–263. doi:10.1109/32.910860

[23] Ishrak Hayet, Adam Scott, and Marcelo d'Amorim. 2024. ChatAssert: LLM-based Test Oracle Generation with External Tools Assistance. *IEEE Transactions on Software Engineering* (2024), 1–15. doi:10.1109/TSE.2024.3519159

[24] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT*. 152–162. doi:10.1145/3236024.3236051

[25] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically Reducing Tree-Structured Test Inputs. In *ASE*.

[26] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*. 34–45. doi:10.1109/MSR.2019.00016

[27] Soneya Binta Hossain and Matthew B. Dwyer. 2025. TOGLL: Correct and Strong Test Oracle Generation with LLMs. In *ICSE*.

[28] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *45th IEEE/ACM International Conference on Software Engineering, ICSE*. 2312–2323. doi:10.1109/ICSE48619.2023.00194

[29] Timotej Kapus and Cristian Cadar. 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 590–600.

[30] J. C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

[31] Daniel Lehmann and Michael Pradel. 2018. Feedback-directed differential testing of interactive debuggers. In *ESEC/SIGSOFT FSE*. 610–620.

[32] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th International Conference on Software Engineering, ser. ICSE*.

[33] Tsz On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 14–26. doi:10.1109/ASE56229.2023.00089

[34] Yi Li, Shaohua Wang, and Tien N Nguyen. 2022. UTANGO: untangling commits with context-aware, graph-based, code change clustering learning model. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 221–232.

[35] Kaibo Liu, Yiyang Liu, Zhenpeng Chen, Jie M. Zhang, Yudong Han, Yun Ma, Ge Li, and Gang Huang. 2024. LLM-Powered Test Case Generation for Detecting Tricky Bugs. arXiv:2404.10304 [cs.SE]

[36] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 304–315. doi:10.1109/ICSE.2019.00045

[37] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: high-coverage testing of software patches.. In *ESEC/SIGSOFT FSE*. 235–245.

[38] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[39] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*. 142–151.

[40] Issei Morita, Yutaro Kashiwa, Masanari Kondo, Jeongju Sohn, Shane McIntosh, Yasutaka Kamei, and Naoyasu Ubayashi. 2024. TraceJIT: Evaluating the Impact of Behavioral Code Change on Just-In-Time Defect Prediction. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 580–591.

[41] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. Experience paper: a study on behavioral backward incompatibilities of Java software libraries. In *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*. 215–225.

[42] Manish Motwani and Yuriy Brun. 2019. Automatically generating precise Oracles from structured natural language specifications. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 188–199. doi:10.1109/ICSE.2019.00035

[43] Fangwen Mu, Xiao Chen, Lin Shi, Song Wang, and Qing Wang. 2023. Developer-Intent Driven Code Comment Generation. In *ICSE*.

[44] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting Natural Method Names to Check Name Consistencies. In *ICSE*.

[45] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion, In ICSE. *arXiv preprint arXiv:2302.10166*.

[46] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *International Conference on Software Engineering (ICSE)*. IEEE, 75–84.

[47] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J. Mooney. 2021. Deep Just-In-Time Inconsistency Detection Between Comments and Source Code. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 427–435. https://ojs.aaai.org/index.php/AAAI/article/view/16119

[48] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond J. Mooney. 2020. Learning to Update Natural Language Comments Based on Code Changes. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 1853–1868. doi:10.18653/v1/2020.acl-main.168

[49] Profir-Petru Pârtachi, Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2020. Flexeme: untangling commits using lexical flows. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 63–74. doi:10.1145/3368089.3409693

[50] Juan Altmayer Pizzorno and Emery D. Berger. 2024. CoverUp: Coverage-Guided LLM-Based Test Generation. arXiv:2403.16218 [cs.SE]

[51] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-based Validation. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. 209–220. https://doi.org/10.1145/3368089.3409715

[52] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *PACMPL* 2, OOPSLA (2018), 147:1–147:25. https://doi.org/10.1145/3276517

[53] Guoping Rong, Yongda Yu, Song Liu, Xin Tan, Tianyi Zhang, Haifeng Shen, and Jidong Hu. 2025. Code Comment Inconsistency Detection and Rectification Using a Large Language Model. In *International Conference on Software Engineering*

(ICSE).

[54] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2024. SpecRover: Code Intent Extraction via LLMs. *arXiv preprint arXiv:2408.02232* (2024).

[55] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A study of Coverage Guided Test Generation in Regression Setting using LLM. In *FSE*.

[56] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 263–272.

[57] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.

[58] Beatriz Souza and Michael Pradel. 2023. LExecutor: Learning-Guided Execution. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. 1522–1534. doi:10.1145/3611643.3616254

[59] Sadia Tabassum, Leandro L. Minku, Danyi Feng, George G. Cabral, and Liyan Song. 2020. An Investigation of Cross-Project Learning in Online Just-In-Time Software Defect Prediction. In *ICSE*.

[60] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. 2021. QDiff: Differential Testing of Quantum Software Stacks. In *ASE*.

[61] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On Learning Meaningful Assert Statements for Unit Test Cases. In *ICSE*.

[62] Xiuheng Wu, Chenguang Zhu, and Yi Li. 2021. DIFFBASE: A Differential Factbase for Effective Software Evolution Management. In *ESEC/FSE*.

[63] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 126:1–126:13. doi:10.1145/3597503.3639121

[64] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 819–831. doi:10.1145/3650212.3680323

[65] Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. 2024. Enhancing LLM-based Test Generation for Hard-to-Cover Branches via Program Analysis. arXiv:2404.04966 [cs.SE]

[66] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Softw. Test., Verif. Reliab.* 22, 2 (2012), 67–120.

[67] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and Improving ChatGPT for Unit Test Generation. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1703–1726. doi:10.1145/3660783

[68] Andreas Zeller. 2002. Isolating Cause-Effect Chains from Computer Programs. In *Symposium on Foundations of Software Engineering*. ACM, 1–10.

[69] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 2021. Deep just-in-time defect prediction: how far are we?. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. 427–438.

[70] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 2020. C2S: Translating Natural Language Comments to Formal Program Specifications. In *FSE*.

[71] Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: Automatically Classifying and Propagating Natural Language Comments via Program Analysis. In *ICSE*.

[72] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based Neural Source Code Summarization. In *ICSE*.