TestMiner: Software Testing Analysis for GitHub Repositories

Andre Hora
Department of Computer Science, UFMG
Belo Horizonte, Brazil
andrehora@dcc.ufmg.br

José Miguel Rojas
University of Sheffield
Sheffield, UK
j.rojas@sheffield.ac.uk

Romain Robbes
University of Bordeaux, CNRS
Bordeaux, France
romain.robbes@labri.fr

***Abstract*—Software systems have unique testing characteristics. Some projects can emphasize unit tests, while others may focus on end-to-end testing. Test organization may vary across ecosystems: in languages like Python and Java, tests are typically placed in dedicated folders, whereas Go and Rust projects commonly co-locate tests with source code. These distinctions make it harder to understand how a project approaches testing. In this paper, we present TestMiner, a tool for exploring software testing in GitHub repositories. TestMiner provides an overview of a project's testing practices, including test statistics, test location, test metrics across releases, and dependencies related to testing. We used TestMiner in an undergraduate Software Testing course, where 50 students explored the testing practices of real-world GitHub repositories. Overall, students expressed positive feedback regarding TestMiner. They were able to critically explore a variety of testing practices, including test organization, test evolution, test fixtures, mocking, and edge-case testing. TestMiner is available at: https://andrehora.github.io/testminer. Screencast: https://youtu.be/w1sBgLTq-7Y.**

***Index Terms*—Software testing, Software evolution, Mining software repositories, GitHub, SBOM**

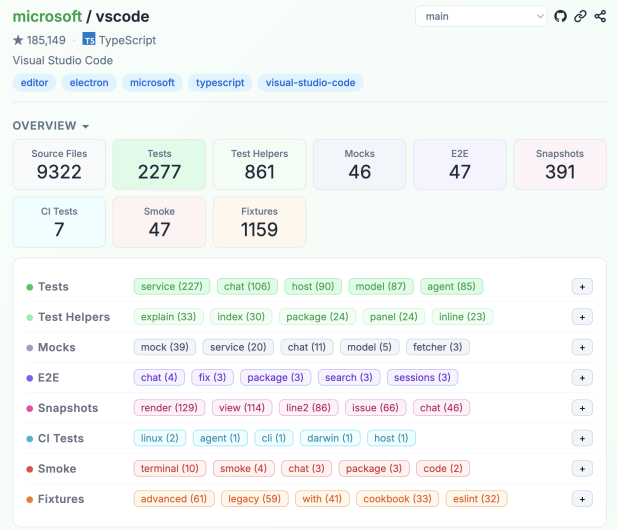

Fig. 1: TestMiner overview for microsoft/vscode.

I. Introduction

Software testing is a key activity in modern software development, and a good test suite is fundamental to ensuring software quality and sustainable evolution [1], [2].

Software systems have unique testing characteristics. While some projects lean heavily on unit tests, others may focus more on end-to-end testing. Projects may also adopt specific techniques, such as mocking, or integrate testing into CI/CD pipelines. Test organization also varies across ecosystems: in languages like Python and Java, tests are typically placed in dedicated folders, whereas Go and Rust projects commonly co-locate tests with source code. Additionally, projects may rely on multiple dependencies to support testing.

In practice, these distinctions make it harder to understand how a project approaches testing. Developers onboarding to a project must explore the repository manually to understand its testing practices, a costly process that scales poorly as codebases grow. Students seeking to understand real-world software testing face the same obstacle: testing practices are present in the repository, but they are not visible without an in-depth analysis. This gap motivates our work.

This paper presents TestMiner, a tool for exploring software testing practices in GitHub repositories. TestMiner provides an overview of a project's testing practices, including test statistics, test location within the repository, test metrics across releases, and dependencies related to testing. TestMiner is a web application and relies on GitHub and jsDelivr APIs to retrieve repository metadata and file-level information. Users can enter any public GitHub repository, owner, or topic, and TestMiner processes the input to generate multiple testing reports, as the one presented in Figure 1 for microsoft/vscode.

We used TestMiner in an undergraduate Software Testing course, where 50 students explored the testing practices of real-world GitHub repositories [3]. Overall, students expressed positive feedback regarding TestMiner to understand software testing. They were able to critically explore a variety of testing practices, including test organization, test evolution, test fixtures, mocking, and edge-case testing.

> **TestMiner**: https://andrehora.github.io/testminer
> **Screencast**: https://youtu.be/w1sBgLTq-7Y

Novelty. General repository mining frameworks [4]–[9] provide no test-specific functionality out of the box, existing testing platforms, such as Codecov [10] and SonarQube [11], require CI instrumentation, and test smell detectors, such as

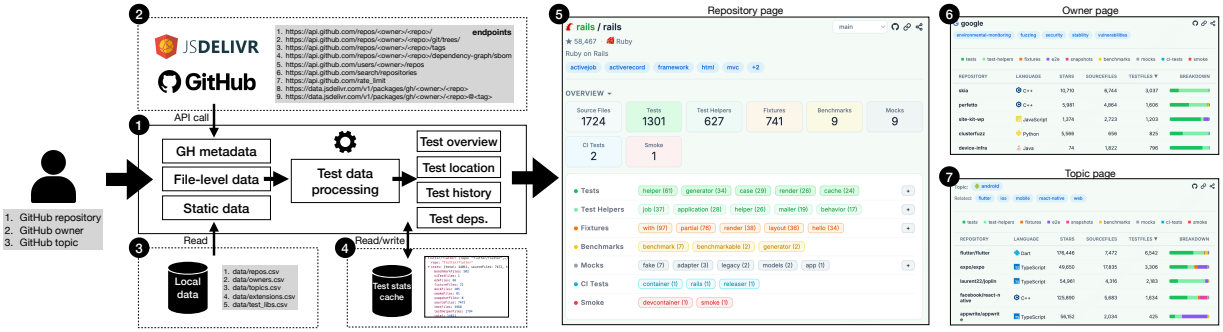

Fig. 2: Overview of TestMiner.

TSDetect [12] and PyNose [13], provide per-test analyses only and require local setups. The novelty of TestMiner lies in its ability to combine multiple testing analyses in a zero-installation, browser-based interface, allowing users to analyze any public GitHub repository without cloning or pipeline configuration. Our classroom evaluation provides initial evidence that this design lowers the effort required to understand how real-world open-source systems are tested.

II. TestMiner

A. Overview

TestMiner is a web application that runs in the browser–Figure 2 depicts a design overview. (1) Users can enter a GitHub repository URL, GitHub owner, or GitHub topic. The input is processed by TestMiner, which generates multiple testing reports. (2) TestMiner relies on GitHub and jsDelivr APIs to retrieve repository metadata and file-level information. (3) Other data, such as autocomplete suggestions, comes from local sources. (4) To avoid frequent requests to the external APIs, TestMiner caches selected testing statistics. Depending on the user input, TestMiner generates three types of report pages: (5) repository, (6) owner, and (7) topic pages.

B. Main Features

1) Repository View: It provides a detailed testing analysis for a single repository and includes four sections: test overview, test location, test history, and test dependencies.

The *Test Overview* section presents testing statistics for the analyzed repository. To compute them, TestMiner retrieves the repository's file list via the jsDelivr API and classifies each file into one of ten categories: tests, test-helpers, fixtures, e2e, snapshots, benchmarks, mocks, ci-tests, smoke, and source code. TestMiner classifies each file by applying a series of filename- and path-based rules. Figures 1 and 2(5) show examples of the *Test Overview* section TestMiner generated for microsoft/vscode[1] and rails/rails,[2] respectively.

The *Test Location* section provides a file tree visualization showing where tests are located within the repository. This visualization is inspired by Distribution Map, a generic technique to reason about the result of software analysis and to help

Fig. 3: Test location for prisma/prisma.

to understand how a given phenomenon is distributed across a software system [14]–[16]. It enables users to easily identify test locations, even in complex projects with deeply nested folder structures. Users can also filter by test category, sort results, and adjust the depth of the folder structure displayed in the visualization. Figure 3 presents the *Test Location* section for the project prisma/prisma.[3]

The *Test History* section presents a chart showing testing metrics across three releases: the initial release, a middle release, and the most recent release. It allows users to track the evolution of testing over time. Figure 4 shows the *Test History* section for project fastapi/fastapi, which increases from close to zero test files in the initial release to 300 in the middle release and nearly 600 in the most recent release.[4]

Finally, the *Test Dependencies* section details dependencies related to testing. In this case, the dependencies are extracted from the GitHub Software Bill of Materials (SBOM), which lists the open-source and proprietary components that constitute a software product [17], [18]. GitHub's SBOM computation supports the analysis of dependencies in multiple ecosys-

[1] https://andrehora.github.io/testminer/#microsoft/vscode
[2] https://andrehora.github.io/testminer/#rails/rails
[3] https://andrehora.github.io/testminer/#prisma/prisma
[4] https://andrehora.github.io/testminer/#fastapi/fastapi

Fig. 4: Test history for fastapi/fastapi.

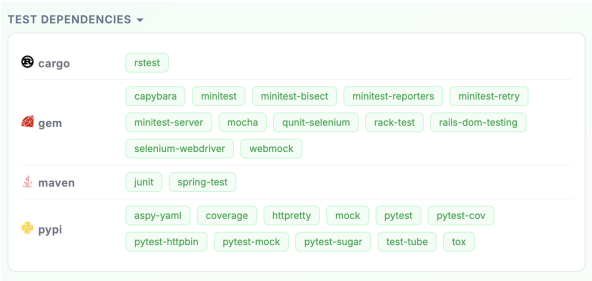

Fig. 5: Test dependencies for github/linguist.

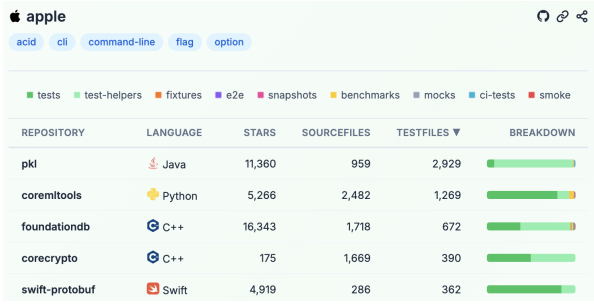

Fig. 6: Owner page for Apple.

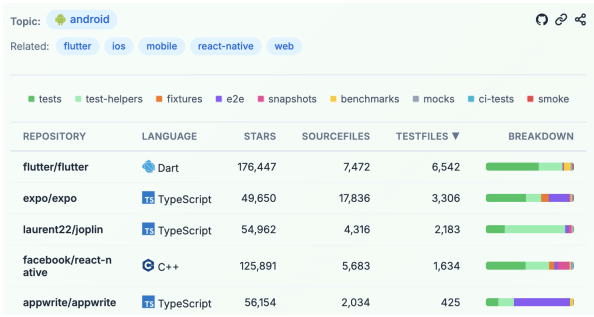

Fig. 7: Topic page for Android.

tems, such as Python, JavaScript, TypeScript, Java, Ruby, Go, Rust, and .NET. Figure 5 shows the *Test Dependencies* section for the github/linguist, which displays the test dependencies for Rust (Cargo), Ruby (Gem), Java (Maven), and Python (PyPI).[5]

2) Owner and Topic Views: While the Repository View focuses on the analysis of a single repository, the Owner and Topic views enable the analysis of multiple repositories. Users can specify a GitHub owner (organization or user) or topic. TestMiner then provides an overview of multiple repositories in a table, enabling users to easily compare them and identify a target repository for further analysis. Users can also filter results by test category. Figure 6 presents the Owner page for Apple[6] and Figure 7 shows the Topic page for Android.[7]

3) Usability and Navigation Features: Due to space constraints, we can only briefly describe further usability and navigation features of TestMiner: *(i)* The *Test Overview* provides complementary data showing test categories grouped by terms, allowing users to easily select a desired test; *(ii)* Test categories can be mapped to their corresponding source code on GitHub, so by clicking a link in the *Test Overview* or *Test Location*, users can navigate to the corresponding source code on GitHub to inspect the actual test implementation;[8] *(iii)* In the *Repository View*, users can navigate to a specific release using the dropdown menu;[9] *(iv)* When navigating in TestMiner, the URL is updated, allowing users to share the exact search state;[10] *(v)* Users can provide a read-only GitHub token to increase the underlying GitHub API limit to 5,000 requests/hour instead of the baseline 60 requests/hour.

[5]https://andrehora.github.io/testminer/#github/linguist
[6]https://andrehora.github.io/testminer/#apple
[7]https://andrehora.github.io/testminer/#topic:android
[8]e.g., https://andrehora.github.io/testminer/#prisma/prisma:cat:tests
[9]e.g., https://andrehora.github.io/testminer/#prisma/prisma@7.8.0

C. Implementation Notes

TestMiner is a Single Page Application (SPA) built with React for the frontend and Vite for the build process. The source code is organized into React components, React hooks, and vanilla JavaScript functions. Test data is fetched from GitHub and jsDelivr APIs. TestMiner also loads other data, such as repositories for autocomplete suggestions and lists of testing libraries used in SBOM analysis, from local sources. In this case, the list of repositories is obtained from the SEART GitHub Search Engine (seart-ghs) [6]. Moreover, to avoid frequent requests to the GitHub and JSDelivr APIs, TestMiner caches selected testing statistics for the Owner and Topic pages in `localStorage`. Finally, TestMiner includes hundreds of unit, component, and end-to-end tests written with Vitest, React Testing Library, and Cypress.

III. Evaluation

A. Overview

We envision TestMiner as a resource for developers, researchers, or contributors seeking to understand a software project's approach to testing. Computer Science students exploring real-world projects represent a particularly relevant user group and a natural validation context. In this context, we conducted a classroom evaluation of TestMiner's usefulness. We tasked 50 undergraduate students in a Software Testing

[10]e.g., https://andrehora.github.io/testminer/#prisma/prisma@2.0.1:cat:tests

TABLE I: Summary of testing practices.

Category	Count	Description
Test organization	10	Test organization, location, or structure
Test evolution	9	Test changes over time
Test content	6	Inner workings of tests
Fixtures	5	Test fixtures (e.g, setups, teardowns)
Mocking	5	Test double practices (e.g., mock, fake)
Test type	5	Unit, integration, or e2e tests
Doc test	4	Documentation testing
Edge cases	3	Edge and exceptional cases
Test dependencies	3	Dependencies to test libs and frameworks

course to (1) select a real-world GitHub repository, (2) explore testing practices in the selected project with TestMiner, and (3) explain a testing practice in an open-ended response.

B. Qualitative Analysis

The data collected is publicly available on GitHub [3] (each *fork* in the repository corresponds to a student's response). Table 1 summarizes the testing practices discussed by participants across the 50 responses (the practices were extracted via thematic analysis). Next, we describe each category and provide representative examples.

1) Test organization: This category appeared in 20% of the responses (10 out of 50). In this case, students focused on the organization, location, or structure of the analyzed tests. For example, a student who analyzed streamlit/streamlit mentioned the overall organization of tests: "*I chose the Streamlit project because I use it a lot in my daily life and was curious to see how such a popular project organizes its tests. In the TEST LOCATION visualization, it is clear that the highest concentration of tests is in the e2e/ and lib/ folders, and there is also a significant amount in frontend/. This makes a lot of sense, as Streamlit has both a Python part and a heavy web interface*". Another student detailed how project facebook/lexical structure regression tests when creating bug-reproducing tests [19]: "*These tests typically follow the format .../**tests**/regression/<issue_number>-<issue_subject>.mjs, as the file packages/lexical-playground/**tests**/regression/4876-unmerge-cell.spec.mjs. Creating test cases for every bug fix is important to prevent regressions and ensure the fix is effective, as the test should fail before the fix and pass after the commit*".

2) Test evolution: Test evolution occurred in 18% of the responses (9/50) and refers to focus on how tests evolve and change. A student described how project JamesNK/Newtonsoft.Json increased its number of test files from 7 to 200 over time: "*The most interesting data observed in TestMiner for the Newtonsoft.Json project is the constant and linear growth in the number of test files throughout the project's history, from version 1.3.1 to 13.0.4. The project went from 7 test files in v1.3.1 to 200 files in v13.0.4, without any sharp jumps or drops in any version*". Similarly, another student observed test growth in psf/requests, noting an increase from 1 to 10 test files: "*The project started with only 1 test file and reached 10 in the current version. The CI Test (run) only appears from version 2.33.1 onwards*".

3) Test content: In 12% of cases (6/50), the responses focused on test content, i.e., the inner workings of tests. A student analyzed project nvbn/thefuck (a tool that suggests fixes for incorrect terminal commands) and noted that its tests emphasize Git commands, which are often confusing for users: "*When observing the test statistics in TestMiner, it was possible to see that a large part of the repository's test files served exclusively to test command fixes for the Git version control system. This focus on Git occurs because, besides being a widely used system, its commands have specificities that can be frustrating for many developers. Thefuck took the opportunity to ensure that most Git commands were automatically corrected and extremely well tested*".

4) Fixtures: In 10% of the responses (5/50), students discussed test fixtures, such as `setUp` and `tearDown` methods commonly found in testing frameworks. A student highlighted the number of test fixtures in fastapi/fastapi and how they contribute to test quality: "*What caught my attention most was the use of fixtures, with a total of 53 in the project. Fixtures are used to reuse data and configurations in tests, avoiding code repetition. Instead of manually configuring each test, the project defines reusable structures that can be shared among several tests. I found this practice interesting because it makes the tests more organized, easier to maintain, and less repetitive. Additionally, it facilitates writing new tests since many configurations are already ready for use. This contributes to improving software quality by encouraging the efficient creation of more tests*". Another student noted that thousands of fixture files were introduced in a specific version of facebook/react: "*With TestMiner, I noticed that from version 16 to 19, there was a significant increase in the number of tests and especially in the number of fixtures; while version 16 had no fixtures, version 19 has more than 3,500*".

5) Mocking: In 10% of the responses (5/50), students explored test double practices, where real components are replaced with simulated versions. A student mentioned how soxoj/maigret mocks external requests in its tests. "*Since it makes thousands of external requests, running real tests on all sites would be slow and unstable. The presence of pytest-httpserver and mock indicates that the repository uses Test Doubles. Therefore, the tests simulate site responses to verify that the application can interpret them correctly without needing actual internet access*". For project apache/dubbo, a student mentioned: "*The 69 mocks mainly cover simulations of service communication channels and service registries. This makes sense for a distributed communication framework, where testing network behavior directly would be slow, fragile, and difficult to reproduce*". Another student mentioned that there are tests for mocks themselves in scikit-learn/scikit-learn: "*In particular, the use of mocks with the objective of mocking classifiers caught my attention. There is also a battery of tests that verify the functioning of this mock itself*".

6) Test type: In 10% of the responses (5/50), students discussed the test types, such as unit, integration, and end-to-end (e2e) tests. For project nestjs/nest, a student highlighted the large number of e2e tests: "*Something I found very interesting*

and even counter-intuitive was the large number of e2e tests, even compared to unit tests. After doing some research, I concluded that NestJS has many integration tests because its architecture is based on modules and dependency injection, where correct behavior depends on the interaction between several parts (controllers, services, guards, pipes). Unit tests isolate these parts but do not guarantee that they work well together. Therefore, there is a strong focus on integration tests: they simulate an environment closer to reality and validate the complete flow of the application". A student also mentioned the large number of e2e tests in stenciljs/core: "*The robust multi-platform and end-to-end (E2E) testing strategy stands out. The presence of 32 E2E test suites, combined with the use of tools like Puppeteer and Playwright (indicated by the dependencies), demonstrates that the team prioritizes validating component behavior in real browsers*".

7) Other cases: Documentation testing, edge cases and test dependencies were mentioned less frequently. A student who analyzed fastapi/fastapi mentioned the use of tests as a means of documentation, mitigating the risk of outdated documentation: "*What caught my attention most was the density of the tests/test_tutorial directory, which leads with 221 files. The fact that the highest concentration of tests is in the test_tutorial folder reveals that the code snippets and examples in the official documentation and tutorials are actually the framework's own automated tests. This significantly reduces the risk of outdated documentation, as any update that breaks a feature will cause the tutorial example to fail. It is an interesting practice that makes perfect sense*".

Summary: Students critically explored a diverse range of real-world testing practices, drawing conclusions that would have required substantial manual effort without tool support. Qualitative responses suggest that TestMiner lowers the barrier to engaging with unfamiliar codebases, supporting its potential as both a practitioner and educational tool.

IV. Limitations

TestMiner relies on the GitHub API to collect repository metadata and is therefore subject to a limit of 60 requests/hour. If users reach this limit, some features may become partially unavailable. TestMiner mitigates this limitation by accepting a read-only GitHub token, which increases the rate limit to 5,000 requests/hour; this token is stored in the browser's `localStorage`. Second, GitHub links in the *Test Overview* and *Test Location* sections may occasionally not work due to synchronization inconsistencies between the data returned by the API and the actual codebase on GitHub, particularly in the `main` and `master` branches. Third, the API may return partial information for very large repositories, which can lead to incomplete test statistics. In such cases, TestMiner displays a message informing users that this is an API limitation.

V. Related Work

The literature provides multiple tools for analyzing Git and GitHub repositories, e.g., [4]–[9]. Commonly, these tools allow users to mine repository metadata and source code, typically via command-line and live datasets, but without specific emphasis on testing practices. Commercial tools such as CodeCov [10] and SonarQube [11] are powerful, but require CI instrumentation. TSDetect [12] and PyNose [13] surface insights about test quality, but their use is confined to detecting "smells" in unit tests. Unlike these tools, TestMiner is a web application focused on the analysis of software testing, allowing developers to navigate and explore GitHub repositories through a testing-oriented perspective.

VI. Conclusion

We presented TestMiner, a tool for exploring software testing practice on GitHub projects. As future work, we plan to incorporate source code analysis into TestMiner, potentially leveraging `tree-sitter` parsers [20] to support multiple programming languages. We would also like to explore the use of Large Language Models (LLMs) to make the output of TestMiner even more accessible to users, e.g., by generating a concise natural language report highlighting the key testing practices observed in the user's provided repository.

Acknowledgments

Research supported by CNPq, CAPES, and FAPEMIG.

References

[1] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.
[2] A. Hora, "Test polarity: detecting positive and negative tests," in *FSE*, 2024.
[3] Exploring software testing with TestMiner, github.com/andrehora/exploring-testing-practices, May, 2026.
[4] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *ESEC/FSE*, 2018.
[5] A. Hora, "GitEvo: Code Evolution Analysis for Git Repositories," in *MSR*, 2026.
[6] O. Dabic, E. Aghajani, and G. Bavota, "Sampling Projects in GitHub for MSR Studies," in *MSR*, 2021.
[7] GitPython, github.com/gitpython-developers/GitPython, May, 2026.
[8] isomorphic-git, github.com/isomorphic-git/isomorphic-git, May, 2026.
[9] JGit, github.com/eclipse-jgit/jgit, May, 2026.
[10] CodeCov, codecov.io, May, 2026.
[11] SonarQube, sonarsource.com/products/sonarqube, May, 2026.
[12] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "tsDetect: an open source test smells detection tool," in *ESEC/FSE*, 2020.
[13] T. Wang, Y. Golubev, O. Smirnov, J. Li, T. Bryksin, and I. Ahmed, "PyNose: A Test Smell Detector For Python," in *ASE*, 2021.
[14] S. Ducasse, T. Gîrba, and A. Kuhn, "Distribution Map," in *ICSM*, 2006.
[15] L. L. Silva, M. T. Valente, M. d. A. Maia, and N. Anquetil, "Developers' perception of co-change patterns: An empirical study," in *ICSME*, 2015.
[16] A. Hora, N. Anquetil, S. Ducasse, M. Bhatti, C. Couto, M. T. Valente, and J. Martins, "Bug maps: A tool for the visual exploration and analysis of bugs," in *CSMR*, 2012.
[17] S. Nocera, S. Romano, M. Di Penta, R. Francese, and G. Scanniello, "On the adoption of software bill of materials in open-source software projects," *JSS*, vol. 230, 2025.
[18] ——, "Software bill of materials adoption: A mining study from GitHub," in *ICSME*, 2023.
[19] A. Hora and G. Fraser, "Understanding Bug-Reproducing Tests: A First Empirical Study," in *AST*, 2026.
[20] tree-sitter, tree-sitter.github.io, May, 2026.