# GivenWhenThen: A Dataset of BDD Test Scenarios Mined from Open Source Projects

Luciano Belo de Alcântara Júnior
luciano.alcantara@dcc.ufmg.br
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brasil

João Eduardo Montandon
joao@dcc.ufmg.br
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brasil

## Abstract

System tests play a crucial role in ensuring the overall quality and reliability of software systems as a whole. Still, we lack large-scale studies and datasets focusing on investigating this type of testing, particularly in the context of Behavior-Driven Development (BDD). In this work we present the GivenWhenThen (GWT) dataset, a collection of 2,289 BDD test scenarios mined from 1,720 real-world open source projects. Each test scenario contains three artifacts: (a) a feature file describing the BDD scenario in plain text, (b) a step definition file responsible for implementing the steps described in the feature file, and (c) a list of system code files used by the step definitions. This way, we ensure to provide a dataset suitable for training and evaluating AI models, conducting empirical studies on BDD practices, and developing tools to automate or assist the creation, maintenance, and execution of BDD test scenarios.

*Dataset Package.* The GWT dataset is publicly available at https://doi.org/10.5281/zenodo.17517696.

**ACM Reference Format:**
Luciano Belo de Alcântara Júnior and João Eduardo Montandon. 2026. GivenWhenThen: A Dataset of BDD Test Scenarios Mined from Open Source Projects. In *23rd International Conference on Mining Software Repositories (MSR '26), April 13–14, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3793302.3793308

## 1 Introduction

Modern software development practices have introduced a lot of automation to the software development lifecycle [18]. Among them, automated testing stands out as one of the most impactful ones. Implementing these tests have become a standard practice in several companies, like Google [19] and Meta [2]. The benefits are many, including the identification of early bugs, documentation of the systems' expected behavior, and help leveraging a more modular design [4, 5, 13]. Many academic works have studied the impact of unit tests—automated tests that verifies the behavior of single functions—in software quality [8, 9, 21]; some even leveraged well-known datasets to perform their large-scale analysis [1, 6, 14, 17].

While unit tests have been extensively studied, investigating system tests remains underexplored [12]. We are interested in a particular approach to system test, known as Behavior-Driven Development (BDD) [20]. BDD focuses on describing the expected behavior of the system in a sequence of test scenarios, written in a semi-structured natural language format. On top of that, developers can implement these scenarios using automation testing frameworks, such as Cucumber[1] and Behave,[2] enabling their automatic execution. This approach promotes collaboration among developers, testers, and product owners, besides providing actionable specifications for automated tests execution [20].

```
Feature: Sign up

Scenario: Successful sign up
  Given I have chosen to sign up
  When I sign up with valid details
  Then I should receive a confirmation email
  And I should see a personalized message
```

**Figure 1: Test scenario for sign-up written in Gherkin, adapted from Wynne et al. [20].**

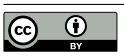
In this work, we present GivenWhenThen (GWT), a dataset of BDD test scenarios mined from real-world open source projects. At its current version, GWT contains 2,289 test scenarios extracted from 1,720 GitHub repositories. We developed a heuristic that maps, for each scenario, its corresponding step definitions—the code that "runs" the steps described in the scenario—as well as associated source code dependencies. As a result, the collected scenarios form a comprehensive BDD test suite.

To the best of our knowledge, **GWT is the first dataset of complete BDD test scenarios**. We envision several potential applications for GWT, including the development of tools to automatically generate BDD artifacts, the analysis of BDD bad practices in open-source projects, and the creation of benchmarks for evaluating AI models in software testing.

***Dataset Availability.*** The GWT dataset is publicly available at https://doi.org/10.5281/zenodo.17517696.

## 2 Background

***Behavior-Driven Development.*** Behavior-Driven Development (BDD) is a software development approach that focuses on describing acceptance tests in a clear and understable way [20]. While Test-Driven Development (TDD) emphasizes writing these tests in traditional programming languages, BDD practitioners specify them using a high-level, ubiquitous language called Gherkin. Figure 1 shows an example of a test scenario to verify the signup feature. As we can see, tests are specified in usage scenarios; each scenario




*MSR '26, Rio de Janeiro, Brazil*

ACM ISBN 979-8-4007-2474-9/2026/04
https://doi.org/10.1145/3793302.3793308

[1] https://cucumber.io/
[2] https://github.com/behave/behave

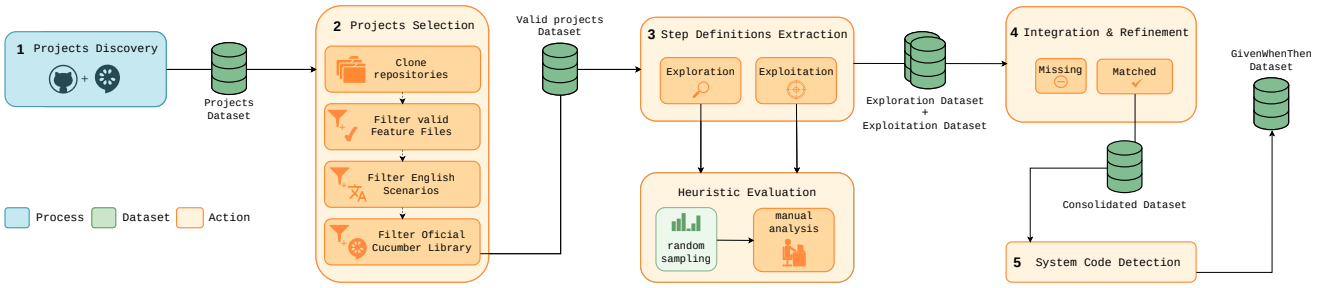



**Figure 2: Data Collection pipeline.**

is composed of a sequence of steps describing its context, actions, and expected outcome.

By adopting this approach, BDD handles two main challenges in software development. First, it fulfills the communication gap between technical and non-technical stakeholders, by using a language that is easily understood by all team members [7]. Second, its semi-structured language enables the use of automated tools to execute the described behavior as automated tests [20]. In practice, the scenarios described in BDD become a living documentation of the system, which is always up-to-date and reflects the current behavior of the software.

***The BDD Testing Stack.*** To make the execution of BDD scenarios possible, developers must implement a set of *step definitions*, pieces of code that map each step in the scenario to its corresponding implementation. Each step definition is implemented in the same programming language as the system under test, and it calls the methods provided by the *system code* to effectively execute the actions expected for the step.

In other words, creating a dataset of comprehensive BDD test scenarios requires at least three main components: (a) the *feature files* containing the test scenarios, (b) the *steps definition files* responsible for implementing the steps described in the scenarios, and (c) the *system code files* that contain the actual implementation of the system under test.

## 3 Data Collection

We organized the data collection process into five steps, as illustrated in Figure 2. We detail each of these steps as follows.

***1. Projects Dicovery.*** On October 26, 2025, we queried the GitHub API for projects that declared Cucumber as an explicit dependency in their configuration files. Since BDD is a practice that can be adopted in any programming language—and Cucumber is one of the most popular tools to support this practice—we leveraged the programming languages which Cucumber has official support, such as JavaScript, Java, and Ruby, and implemented search strategies that looked for the `cucumber` library in the configuration files of each language, such as `package.json`, `pom.xml`, and `Gemfile`. We also filtered the repositories that contained at least one file with the `.feature` extension, the standard format for feature files.

We obtained a total of 5,170 unique repositories in this initial phase. From these projects, 2,040 were written in JavaScript, 1,258 in Ruby, and 1,029 in Java; these three languages together represent more than 83% of the mined repositories. Thus, we decided to proceed with only projects written in one of these three languages, totaling 4,327 repositories. Projects implemented in other languages (e.g., Kotlin, OCaml, Scala, Go, C++, and Lua) were excluded due to their limited representation.

***2. Projects Selection.*** In this step, we cloned each repository locally and filtered them according to the following criteria:

- **Valid Feature Files:** We removed projects with invalid feature files, i.e., empty or commented-only files. A valid feature file should contain at least one scenario with at least one step (`Given`, `When`, `Then`, `And`, `But`).
- **Features written in English:** We restricted feature files to English to ensure consistent natural-language processing and similarity-based step matching, which would otherwise require language-specific pipelines. Language detection was performed using the *Langdetect* library.[3]
- **Uses Official Library:** We selected projects that not only declared Cucumber as a dependency, but also used it in their codebase. To perform this verification, we implemented a set of regular expressions to identify import statements and annotations related to Cucumber for Java, JavaScript, and Ruby. For example, we searched for import statements like `import io.cucumber.*` and annotations like `@Given`, `@When`, and `@Then` in Java projects. In JavaScript projects we looked for `require('cucumber')` or `import { Given, When, Then } from 'cucumber'` statements.

After filtering, 1,720 of 4,327 repositories remained, excluding 655 invalid-feature, 400 non-English, and 1,552 non-official Cucumber projects, totaling 7,872 valid feature files.

***3. Step Definitions Extraction.*** Step definitions are mapped to feature steps using literal strings or regular expressions [20]. In other words, both are not structurally linked, which makes the procedure to map them more challenging. We implemented a matching procedure to link, for each feature file, the step definition file used to implement its steps. This procedure adopts two different strategies to perform this match: *Exploitation* and *Exploration* [10, 16].

*Exploitation Match.* This heuristic matches feature steps to step definitions based only on properties we consider are highly indicative of a correct match, such as exact text match and explicit feature reference. The goal is to maximize the detection of correct pairs,

[3] https://pypi.org/project/langdetect/

even if that means missing some of them. This strategy detected 1,577 tuples of feature steps and their corresponding step definitions. We randomly selected a sample of 309 tuples (95% confidence level and 5% margin of error) for manual validation, and verified that 299 were correct, resulting in a precision of 97%.

*Exploration Match.* This strategy adopts a wider set of properties to map feature steps to step definitions, including filename match, step annotation match, etc. The goal is to favor the detection of weakly connected pairs, i.e., pairs that are defined with different naming conventions, more common in JavaScript and Ruby projects. This heuristic detected 3,221 tuples of feature steps and definitions. We randomly selected a sample of 343 tuples (95% confidence level and 5% margin of error) for manual validation, and verified that 323 were correct, resulting in a precision of 94%.

***4. Integration and Refinement.*** After obtaining the results from both heuristics, we merged them into a single dataset. As 1,051 pairs were identified by both heuristics, the integrated dataset contained 3,747 unique pairs of feature files and their corresponding steps definition files. We also detected that 1,458 feature files had no step definitions mapped to them; we removed these files from the dataset. Finally, 348 feature files had their features steps partially mapped, i.e., some steps were not matched to any step definition. In this case, we removed the test scenarios that contained at least one unmapped step, keeping only those scenarios that had all their steps mapped to step definitions.

***5. System Code Detection.*** In this last step, we obtained the system code files used by each step definition file extracted previously. For this, we implemented a static analyzer that inspects the source code of each step definition file and identifies the project files it depends on. Specifically, we leveraged all import statements that refer to source code files in the project—external dependencies were ignored—and include them as part of the system code used by the step definition file. At the end of this process, we obtained a total of 2,289 BDD test scenarios, each paired with its corresponding step definition file and related system code files.

## 4 The GivenWhenThen Dataset

The GWT dataset contains 2,289 complete BDD test scenarios. Most come from Java projects, since it is the language used in 1,755 scenarios (76%); JavaScript and Ruby are used in 380 (17%) and 154 (7%) scenarios, respectively.

### 4.1 Dataset Structure

The GWT dataset stores its BDD scenarios as JSON objects. Figure 3 shows one as example. Each instance contains basic metadata about the test scenario, e.g., its language, repository name, and the path to important source files. The object also includes the content of its *feature file*, *steps definition file*, and the *System Code files* the scenario depends upon. These properties are detailed below.

- `repository`: String that identifies the project from where the test scenario was extracted. It allows dataset users to traceback the project responsible for creating the test.
- `language`: String containing the programming language used to implement the step definitions and system code files used in the test scenario.

```
{
  "repository": "Test-Architect/playwright-java-bdd-sample",
  "language": "java",
  "normalized_repo_path": "<path>/test-architect_playwright-java-
       bdd-sample",
  "feature_file": "src/test/resources/features/google_search.
       feature",
  "feature_content": "Scenario: Search for Playwright on Google
       Given I am...",
  "step_definitions_file": "<path>/steps/GoogleSearchSteps.java",
  "step_definitions_content": "... public class GoogleSearchSteps
       { private ...",
  "system_code_files": [
    {
      "name": "GoogleHomePage.java",
      "path": "<path>/framework/pages/GoogleHomePage.java",
      "content": "... public class GoogleHomePage {  private final
            Page page ..."
    }
  ]
}
```

**Figure 3: A JSON example containing a BDD test scenario stored in GWT.**

- `normalized_repo_path`: Path leading to the project's root when cloned locally. It helps dataset users to direct access the source code files.
- `feature_content` and `feature_file`: The path and the full content of the *feature file* used in the test scenario.
- `step_definitions_content` and `step_definitions_file`: The path and the full content of the *step definitions file* used in the test scenario.
- `system_code_files`: A list of *system code files* the scenario depends upon. Each element maps the name of the system code file, its file's path, and the full content of the file.

### 4.2 A Closer Look at the Scenarios

Figure 4 shows the snippets of the *feature file*, *step definitions file*, and the *system code file* extracted from the scenario presented at Figure 3. Due to space constraints, we ommitted the full content of these files.

Figure 4a lists the *feature file*, written in plain text using the *Gherkin* syntax. Lines 1–2 declares the feature and test scenario of this example, which is to perform a Google search with the "Playwright" word. Lines 3–5 describe the three steps needed for a user to execute this scenario. It first opens Google's home page, then searches for the "Playwright" word. The last step checks if the returned page has the "results" word. If read in order, these steps outline the actions a user should take to execute this test scenario.

Figure 4b provides a snippet of the *step definitions file*. The `GoogleSearchSteps` class contains a `home` attribute, of the type `GoogleHomePage`. Originally this class defines all steps, but we focus on the second one: `When I search for "Playwright"`. This step is linked to the `search(String)` method (lines 5–7) through the `@When("I search for {string}")` annotation (line 4). The `"{string}"` placeholder maps the `"Playwright"` text to the `term` parameter. Inside the method (line 6), `home.search(term)` command calls the `search()` method in the `GoogleHomePage` class. Thus, executing the step `When I search for "Playwright"` triggers the call `home.search("Playwright")`.

Finally, Figure 4c presents the `GoogleHomePage` class. This class encapsulates the interactions with Google's home page inside its

```
Feature: Google Search
  Scenario: Search for Playwright on Google
    Given I am on Google home page
    When I search for "Playwright"
    Then I see results stats containing "results"
```

**(a) Feature File.**

```
public class GoogleSearchSteps {
    private GoogleHomePage home;
    // ...
    @When("I search for {string}")
    public void search(String term) {
      home.search(term);
    }
}
```

**(b) Step Definitions File.**

```
public class GoogleHomePage {
  private final Locator searchBox;
  public GoogleHomePage(Page page) {
    this.searchBox = page.locator("textarea[name='q']");
  }
  public void search(String term) {
      searchBox.fill(term);
      searchBox.press("Enter");
  }
}
```

**(c) System Code File.**

**Figure 4: The source code files of one BDD test scenario stored in GWT.**

atributtes and methods. When initialized, the class first maps the `searchBox` attribute to its correponding interface element (line 4). The instructions needed to perform the search action are implemented in the `search(String)` method, where, it fills *searchBox* with the text passed as parameter and then press the `Enter` key to execute the search query (lines 6–9).

When combined, these files provide a comprehensive view of the presented BDD test scenario. The *feature file* specifies the behavior to perform a search query in a human-readable format. The *step definitions file* bridges the gap between the natural language steps and the executable code. Finally, the *system code file* effectively interacts with the web page and execute the search query. This complete structure ensures that the BDD test scenario can be fully automated and executed as intended.

## 5 Use Cases and Limitations

***Use Cases***. GWT opens up several opportunities for research and practical applications, as follows.

*Training and Evaluating AI Models.* GWT can support large language models trainining, finetuning, and evaluation, for tasks such as generating step definitions from feature files, or leveraging BDD scenarios from source code or natural language descriptions.

*Empirical Studies.* Researchers can use GWT to study the adoption of BDD in open-source projects, such as the structure, complexity, and quality of the BDD artifacts.

*Tool Development.* The dataset can serve as a benchmark for developing and evaluating tools that aim to automate or assist in the creation, maintenance, and execution of BDD test scenarios.

***Limitations***. As it is in an early stage, GWT has some limitations. The dataset includes only open source projects collected from GitHub, which may not reflect practices in private or enterprise-level projects. Most test scenarios were collected from Java-based systems, which may limit the representation of BDD practices in other languages. Moreover, the projects were not compiled nor executed, and the static analysis used to map step definitions and feature files may have overlooked dynamic dependencies. Future versions of GWT can address these limitations by including additional programming languages and improving mapping techniques.

## 6 Related Work

Many research works have used or proposed test-based datasets to support the investigation of automated tests [1, 3, 6, 11, 12, 14, 15, 17]. For example, *Methods2Test* [17] leveraged a dataset with several unit tests implemented in Java, and their corresponding methods under test. Later, Abdelmadjid and Dyer [1] generated an equivalent dataset but for the Python ecosystem. Some datasets focused on tests' quality issues. Just et al. [11] introduced *Defects4J*, a dataset with reproducible bugs in Java systems, where each bug is accompanied by the test that exposes it. Other datasets tackle domain-specific problems, such as GUI testing in mobile applications [15] and test library migration [3].

Recently, Meglio et al. [12] proposed a dataset containing end-to-end tests from web applicaitons. By contrast, the GivenWhenThen dataset focused on providing a *collection of full BDD test scenarios*, including the *feature file*, *step definition file*, and *system code files* for each BDD scenario we extracted.

## 7 Next Steps

This work presents GivenWhenThen (GWT), the first comprehensive dataset of BDD test scenarios. At its current version, GWT contains 2,289 BDD complete scenarios mined from 1,720 open source projects. We intend to extend this work in two major directions. We plan to use the dataset as a benchmark for evaluating the effectiveness of LLMs at generating BDD artifacts, i.e., automatically generate the step definitions for a given feature file. Furthermore, we intend to use GWT to support the development of agent-based approaches to act on the whole BDD cycle, from writing the feature file to executing the test scenario. Finally, we plan extend GWT with BDD scenarios from other languages and frameworks, such as Python (Behave), and C# (BDD-SpecFlow).

## Acknowledgments

This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) www.ines.org.br, CNPq grant 408817/2024-0. It was also supported by grants from CNPq (403304/2025-3) and by FAPEMIG (APQ-02419-23).

## References


[1] Idriss Abdelmadjid and Robert Dyer. 2025. pyMethods2Test: A Dataset of Python Tests Mapped to Focal Methods. In *22nd International Conference on Mining Software Repositories (MSR)*. 846–850.

[2] 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 *32nd ACM Symposium on the Foundations of Software Engineering (FSE)*. ACM, Porto de Galinhas, Brazil, 1–12.

[3] Altino Alves and Andre Hora. 2025. TestMigrationsInPy: A Dataset of Test Migrations from Unittest to Pytest. In *Mining Software Repositories (MSR): Data and Tools Showcase Track*. 1–5.

[4] Kent Beck. 2003. *Test Driven Development: By Example*. Addison-Wesley Professional, Boston.

[5] Pedro Calais and Lissa Franzini. 2023. Test-Driven Development Benefits Beyond Design Quality: Flow State and Developer Experience. In *45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, Melbourne, Australia, 106–111.

[6] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *ACM International Conference on the Foundations of Software Engineering (FSE)*. Association for Computing Machinery, Porto de Galinhas, Brazil, 1–5.

[7] Eric Evans. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, Boston, MA, USA.

[8] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2015. Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 24, 4 (Sept. 2015), 23:1–23:49.

[9] Vahid Garousi, Michael Felderer, Marco Kuhrmann, Kadir Herkiloğlu, and Sigrid Eldh. 2020. Exploring the Industry's Challenges in Software Testing: An Empirical Study. *Journal of Software: Evolution and Process* 32, 8 (2020), e2251.

[10] Katja Hofmann, Shimon Whiteson, and Maarten de Rijke. 2013. Balancing Exploration and Exploitation in Listwise and Pairwise Online Learning to Rank for Information Retrieval. *Information Retrieval* 16, 1 (2013), 63–90.

[11] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *2014 International Symposium on Software Testing and Analysis (ISSTA)*. 437–440.

[12] Sergio Di Meglio, Luigi Libero Lucio Starace, Valeria Pontillo, Ruben Opdebeeck, Coen De Roover, and Sergio Di Martino. 2025. E2EGit: A Dataset of End-to-End Web Tests in Open Source Projects. In *22nd International Conference on Mining Software Repositories (MSR)*. 836–840.

[13] John Ousterhout. 2018. *A Philosophy of Software Design*. Yaknyam Press, Palo Alto, CA.

[14] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (Jan. 2024), 85–105.

[15] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking Automated GUI Testing for Android against Real-World Bugs. In *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 119–130.

[16] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press, Cambridge, MA.

[17] Michele Tufano, Shao Kun Deng, Neel Sundaresan, and Alexey Svyatkovskiy. 2022. Methods2Test: A Dataset of Focal Methods Mapped to Test Cases. In *19th International Conference on Mining Software Repositories (MSR)*. 299–303.

[18] Marco Tulio Valente. 2024. *Software Engineering: A Modern Approach*. Leanpub, Online.

[19] Titus Winters, Tom Manshreck, and Hyrum Wright. 2020. *Software Engineering at Google: Lessons Learned from Programming Over Time* (1st edition ed.). O'Reilly Media, Sebastopol, CA, USA.

[20] Matt Wynne, Aslak Hellesoy, and Steve Tooke. 2017. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, Raleigh, NC, USA.

[21] Tao Xie, Nikolai Tillmann, and Pratap Lakshman. 2016. Advances in Unit Testing: Theory and Practice. In *38th International Conference on Software Engineering Companion (ICSE)*. ACM, Austin, TX, USA, 904–905.