Shaker Action: Finding flaky tests on the cloud in Continuous Integration

Marcello Santiago Cordeiro

Recife
April, 2021
Federal University of Pernambuco
Center of Informatics

Marcello Santiago Cordeiro

Shaker Action: Finding flaky tests on the cloud in Continuous Integration

A B.Sc. Dissertation presented to the Bachelor’s Program in Computer Engineering of the Center of Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Engineering.

Advisor: Marcelo d’Amorim

Recife
April, 2021
Abstract

Software testing is an important practice to guarantee the quality of a product in development. Tests can be performed manually or automatically and should always end with the same result, be it a success or a failure. However, there are tests said to be flaky that won’t always end with the same output, instead passing or failing depending on the testing environment. This problem causes failures to be mistaken for regressions, making tests unreliable. This document proposes Shaker Action, a tool capable of detecting flaky tests on the cloud in CI. The results show Shaker is 2.73 times more likely to find flakes than normal testing, can be seamlessly integrated with any GitHub repository and saves development time.

**Keywords:** testing, flaky, continuous integration.
Resumo

Teste de software é uma prática importante para garantir a qualidade de um produto em desenvolvimento. Testes podem ser realizados manualmente ou automaticamente, e devem sempre terminar com o mesmo resultado, sucesso ou com falhas. Porém, há testes chamados de flaky que nem sempre irão terminar com o mesmo resultado, podendo terminar com sucesso ou com falhas dependendo do ambiente de teste. Este problema pode ser confundido com uma regressão e diminuir a confiabilidade dos testes. Este documento propõe o Shaker Action, uma ferramenta capaz de detectar testes flaky na cloud em integração contínua. Os resultados mostram que Shaker é 2,73 vezes mais capaz de encontrar flakes que testes normais, pode ser facilmente integrado com qualquer repositório no GitHub e poupa tempo de desenvolvimento.

Palavras-chave: testes, flaky, integração contínua.
## Contents

1 Introduction 1  
1.1 Goals 1  
1.2 Structure 1  

2 Background 3  
2.1 Software testing 3  
2.2 Flaky tests 3  
2.2.1 Causes of flakiness 4  
2.3 GitHub Actions 4  
2.3.1 Usage 5  
2.4 stress-ng 5  
2.4.1 Usage 5  

3 Shaker Action 7  
3.1 Overview 7  
3.2 Implementation 7  
3.2.1 Core application 7  
3.2.2 Action 8  
3.3 Usage 8  

4 Evaluation 10  
4.1 Methodology 10  
4.2 Results 11  

5 Conclusion 12  
5.1 Related work 12  
5.2 Future work 12  

Bibliography 13
List of Figures

2.1 GitHub Actions results page 5
3.1 Shaker Action output 9
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>stress-ng configurations</td>
<td>8</td>
</tr>
<tr>
<td>4.1</td>
<td>Objects of analysis</td>
<td>10</td>
</tr>
<tr>
<td>4.2</td>
<td>Failures</td>
<td>11</td>
</tr>
<tr>
<td>4.3</td>
<td>Failure ratio</td>
<td>11</td>
</tr>
</tbody>
</table>
List of Source Codes

2.1 Sum test 3
2.2 API request test 4
2.3 GitHub Actions usage 5
2.4 stress-ng usage 6
3.5 Shaker integration 8
4.6 Using Shaker in other repositories 11
List of Acronyms

**API**  Application Programming Interface.

**CI**  Continuous Integration.

**CI/CD**  Continuous Integration/Continuous Delivery.

**OS**  Operating System.
Software testing is an important practice to guarantee the quality of a product in development. Tests can be performed manually or automatically and should always end with the same result, be it a success or a failure. A good test suite will help detect problems during development, maintenance and updates.

One way of automating software testing is in Continuous Integration/Continuous Delivery (CI/CD), a method to frequently deliver apps to customers by introducing automation into the stages of app development, such as building, testing and merging. Shahin et al. found that CI/CD reduces build and test times, increases visibility and awareness on the build and test results, and detects violation, flaws and faults.

However, there are tests said to be flaky that won’t always end with the same output, instead passing or failing depending on the testing environment, third-party libraries or Application Programming Interfaces (APIs). This problem causes failures to be mistaken for regressions, making tests unreliable and, if ignored, make the product more likely to crash. Most of the flakiness problems are related to asynchronous waiting, concurrency and test order dependency.

Micco reported that 16% of all tests at Google have some level of flakiness associated with them and that 1.5% of all test runs reports a flaky result. This causes developers to lose trust in their test suite, sometimes ignoring failures. Palmer reported that flakiness at Spotify is a major problem and presented their efforts to mitigate them, which reduced test flakiness from 6% to 4%.

1.1 Goals

The main goal of this project is to propose a tool capable of detecting flaky tests on the cloud in CI. With such tool, the following goals can also be achieved:

- Help developers regain trust in their test suite;
- Seamlessly integrate this tool with existing CI workflows;
- Prevent delays and development downtime due to repetitive testing.

1.2 Structure

This document is structured as follows:

- Chapter Concepts and tools used in this project, necessary to understand the remaining chapters;
- Chapter Description, implementation and usage of the tool proposed in this project;
• Chapter 4 Methodology, experiments and results obtained from the tool proposed in this project;

• Chapter 5 Conclusions, related work and future work.
In this chapter, we describe the main concepts and tools used in this project.

## 2.1 Software testing

Software testing is the process of analysing a software item to detect the differences between existing and required conditions, according to the ANSI/IEEE 1059 standard [8]. It’s an important practice to guarantee the quality of a product in development and helps finding bugs before the software or new features are shipped.

Tests should be deterministic: a test case should always pass or always fail. The example in the source code 2.1 is an example of a test that will always pass, confirming the intended behaviour of the sum operation.

```java
@Test
public void sumTest() {
    assertTrue( (1 + 2) == 3 );
}
```

Source code 2.1: Sum test

The scope of the object being tested can vary: testing individual components helps isolating problems, testing multiple integrated components assures the combined functionality works as intended, and testing the entire workflow, replicating real world usage, assures all functionality is working as intended. These types of tests are called **unit**, **integration** and **end-to-end**, respectively. There is also **regression testing**, which means re-running tests to confirm a code change has not caused any tests to fail. A regression is caused when any of these tests have failed after a code change.

## 2.2 Flaky tests

Flaky tests are tests that both passes and fails non-deterministically without any changes in the code base or test suite. This is a problem that causes developers to lose trust in their test suite and sometimes ignoring their results [9].

Due to their non-deterministic nature, companies like Google [5] and Spotify [7] have to develop tools focused on tracking flakes, but once tests are marked as flaky it encourages developers to ignore flakiness, which can mask a real bug in the code base [5]. Rahman et al. [3] observed that ignoring flakes causes the product to be more crash prone. Flakes also hinders regression testing: failures can be hard to reproduce and wastes development time because de-
Developers will have to debug an apparent regression only to find out the failure is due to a flaky test [4].

Consider the example in source code 2.2. An API request is asynchronously sent to a server, but the client does not wait for a response. If the server does not answer in time, the response will be a null pointer, causing the test to fail.

```java
@Test
public void apiTest()
{
    Data data = apiRequest();
    assertNotNULL(data);
}
```

Source code 2.2: API request test

### 2.2.1 Causes of flakiness

The most common causes of flakiness are due to concurrency, asynchronous waiting, test order and dependency, and end-to-end tests [4], [5], [7], [10].

- **Concurrency**: mostly caused by race conditions [4]. A race condition is when multiple threads try to access and modify shared data at the same time. This can be mitigated by using locks;

- **Asynchronous waiting**: when a test execution makes an asynchronous call and does not properly wait for a response [4]. For example, when a server takes too long to respond and the client doesn’t wait enough time. This can be mitigated by waiting more time or retrying to contact the server;

- **Test order and dependency**: executing tests in a certain order can cause the state of the program to be invalid. Zhang et al. [11] found that test independence is widely assumed, but test dependency does arise in practice. This can be mitigated by clearing the state of the program before or after every test;

- **End-to-end tests**: end-to-end tests are flaky by nature [7]. They simulate real world usage but can’t isolate failures and depends on the entire product workflow functioning correctly. Instead, having more unit and integration tests is more desirable because they are faster, reliable and can isolate failures [10].

### 2.3 GitHub Actions

GitHub Actions is a GitHub service to automate tasks within the software development life cycle [12]. It offers free CI/CD services and can be seamlessly integrated into any repository in GitHub.

A workflow in GitHub Actions can not only build, test and deploy, but also use community-powered workflows, also known as actions, to perform a variety of tasks, such as code coverage and third-party integration. These community workflows can hosted in GitHub or be self-hosted.
2.3.1 Usage

Workflows are configured with YAML files in the ".github/workflows" directory of any repository. The source code 2.3 is an example of a workflow to perform tests using an action in the repository "username/example-action". It hasbeen configured to be hosted in an Ubuntu Operating System (OS) and will be triggered after a push to the main branch. After a workflow is triggered, the developer can follow the progress of the CI in the Actions tab from the repository page, as shown in Figure 2.1.

```yaml
name: CI
on:
  push:
    branches: [main]
jobs:
  example:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Tests
      uses: username/example-action@v3
```

Source code 2.3: GitHub Actions usage

![GitHub Actions results page](image)

Figure 2.1 GitHub Actions results page

2.4 stress-ng

stress-ng is an open-source tool designed to stress test a computer in various selectable ways [13]. It inserts noise and load into the environment and can interfere with other running applications. For example, a memory intensive application will take slower to run if the memory is being stressed and causing the system to move parts of it to the disk (swapping).

2.4.1 Usage

The command 2.4 uses the following arguments [13]:

• **–cpu $CPU_WORKERS**: starts $CPU_WORKERS workers exercising the CPU;

• **–cpu-load $CPU_LOAD**: loads CPU with $CPU_LOAD percent loading for the CPU stress workers;

• **–vm $VM_WORKERS**: starts $VM_WORKERS workers continuously allocating memory in the Virtual Memory and writing to the allocated memory;

• **–vm-bytes $VM_BYTES%**: allocates $VM_BYTES percent of memory per VM worker;

```bash
$stress-ng --cpu $CPU_WORKERS --cpu-load $CPU_LOAD
    --vm $VM_WORKERS --vm-bytes $VM_BYTES%
```

Source code 2.4: stress-ng usage
In this chapter, we present our tool capable of detecting flaky tests on the cloud in CI: Shaker Action, or Shaker for short.

### 3.1 Overview

Shaker is based on the observations in Chapter 2 that concurrency and asynchronous waiting are the most common causes of flakiness, and that stressing the environment can interfere with other running applications. By inserting noise and load into the testing environment with stress-ng, thus competing for resources, Shaker attempts to cause flaky tests to fail, re-running tests with different stress-ng configurations.

Another motivation, also described in Chapter 2, is that flakiness causes delays, can be mistaken for regressions and aren’t easily reproducible, requiring multiple test runs. These problems can be mitigated by actively searching for flakes in CI instead of having tests suddenly fail in late stages of development. For this, GitHub Actions was chosen as supported platform: it’s free, GitHub is the largest code host in the world [14], and how seamless actions can be integrated into any repository.

Shaker is configurable in three ways:

- **Testing tool**: tool used to run the tests. Currently supports Maven (for Java projects) and pytest (for Python projects);
- **Number of no-stress runs**: number of runs without any stressing;
- **Number of sets of stress runs**: for each set of stress run, Shaker will run the tests for each stress-ng configuration. For example, if the user inputs 3 sets of stress runs and there are 4 different stress-ng configurations, the tests will run 12 times in total.

### 3.2 Implementation

This section is divided in two parts: the implementation of the core application and the action to be used in repositories.

#### 3.2.1 Core application

The core application is a collection of scripts written in the Python. Shaker will perform the initial setup: download dependencies, compile the user project and execute tool specific commands. Then, for each test run, Shaker will pass arguments to the tool to output the test results in files with the JUnit XML format. After the tests are finished, Shaker searches for all test result files, parses them and groups all failures by module and test name, accumulating the number
of failures in no-stress runs, the number of failures in stress runs, and the description of each failure (if unique).

The stress runs were implemented by spawning a stress-ng process, starting the test run, and then terminating the stress-ng process. Each stress run of the set will run with different configurations. Silva et al. [15] found 4 stress-ng configurations that are the most likely to find flakes, and these are being used in every set of stress run. The configurations are as follows:

<table>
<thead>
<tr>
<th>Configuration</th>
<th>CPU workers</th>
<th>CPU load</th>
<th>VM workers</th>
<th>VM bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>58</td>
<td>1</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>61</td>
<td>1</td>
<td>73</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>34</td>
<td>1</td>
<td>37</td>
</tr>
</tbody>
</table>

3.2.2 Action

The action is a Docker image containing the core application and configured to support repository integration in GitHub. The Docker image also contains all the supported tools, necessary dependencies and is based on Ubuntu 20.04. This helps controlling the testing environment and making sure all the necessary packages are present. As required for all actions, a YAML file specifies the action inputs and how to execute it, in this case, with Docker.

3.3 Usage

To use Shaker in a repository, the source code has to be included in a repository’s workflow file as a new job. It creates a new job called shaker and will be executed when the workflow is triggered. The three arguments, testing_tool, no_stress_runs and stress_runs are the inputs the user can configure and were described in Section 3.1.

After the workflow run is finished, the results will be displayed in the Actions tab. If no test failures have been detected, the job will be marked as successful. If test failures have been detected, the job will be marked as unsuccessful. Figure 3.1 shows how a failure is displayed.

```
shaker:
  runs-on: ubuntu-latest
  steps:
  - uses: actions/checkout@v2
  - name: Shaker
    uses: STAR-RG/shaker-action@main
    with:
      testing_tool: maven
      no_stress_runs: 3
      stress_runs: 3
```

Source code 3.5: Shaker integration
Figure 3.1  Shaker Action output

```
--- Failure in module com.netflix.exhibitor.core.config.zookeeper.TestZookeeperConfigProvider ---
  > at testConcurrentModification
  No stress failures: 0 (0.00%)
  Stress failures: 4 (23.53%)
  Total failures: 4 (23.53%)

  > Descriptions:
    java.lang.AssertionError: null
```
In this chapter, we describe how Shaker was evaluated and present the results obtained.

## 4.1 Methodology

To conduct this evaluation, 11 projects were chosen as objects of analysis based on the following criteria:

- The project is written in Java and uses Maven as its project management tool;
- The project has more than 1000 test cases or the repository has more than 1000 stars.

The list of repositories, followed by the ref used for analysis (release tag or commit SHA-1 hash), number of tests and stars is contained in Table 4.1

<table>
<thead>
<tr>
<th>Repository</th>
<th>Ref</th>
<th>Tests</th>
<th>Stars</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azure/azure-iot-sdk-java</td>
<td>a9226a5</td>
<td>4563</td>
<td>153</td>
</tr>
<tr>
<td>CorfuDB/CorfuDB</td>
<td>b99ecff</td>
<td>954</td>
<td>541</td>
</tr>
<tr>
<td>OpenHFT/Chronicle-Queue</td>
<td>bec195b</td>
<td>408</td>
<td>2.3k</td>
</tr>
<tr>
<td>soabase/exhibitor</td>
<td>d345d2d</td>
<td>52</td>
<td>1.7k</td>
</tr>
<tr>
<td>vaadin/flow</td>
<td>6.0.6</td>
<td>4679</td>
<td>304</td>
</tr>
<tr>
<td>apache/hbase</td>
<td>d50816f</td>
<td>6024</td>
<td>4k</td>
</tr>
<tr>
<td>intuit/karate</td>
<td>09bc49e</td>
<td>529</td>
<td>4.7k</td>
</tr>
<tr>
<td>killbill/killbill</td>
<td>killbill-0.22.21</td>
<td>1828</td>
<td>2.3k</td>
</tr>
<tr>
<td>mock-server/mockserver</td>
<td>b1093ef</td>
<td>3532</td>
<td>3.2k</td>
</tr>
<tr>
<td>apache/ozone</td>
<td>dfd2aaf</td>
<td>1900</td>
<td>359</td>
</tr>
<tr>
<td>RipMeApp/ripme</td>
<td>19ea20d</td>
<td>247</td>
<td>2.4k</td>
</tr>
</tbody>
</table>

To perform the experiment with all projects in parallel, a workflow file containing 11 jobs, one for each repository, was created. The `actions/checkout` action, used in all actions to allow the subsequent actions to use the repository, was configured to clone each repository with the given ref. The source code shows how a job can be created to perform such task.

After a workflow run was finished, results for each run were displayed in the workflow logs for each job. In case of always failing tests, the module containing these tests were discarded because they show defective tests, broken modules or broken dependencies, and aren’t classified as flakes.
### 4.2 Results

Considering that one set of stress runs consists of 4 stress runs, one for each configuration, Shaker executed 27 no-stress runs and 56 sets of stress runs (224 runs). The exception was hbase, which was subject to 27 no-stress runs and 19 sets of stress runs (76 runs) because more than one no-stress run and one set of stress runs per job causes the job to take longer than 6 hours, triggering a timeout and cancelling the job.

All the failures discovered are contained in Table 4.2. On average, a workflow configured to run 1 no-stress run and 4 sets of stress runs took 4 hours. Table 4.3 shows the ratio of failures discovered per run. It shows that Shaker’s stress runs are from 1.13 to 6.25 times more likely to find flakes.

#### Table 4.2 Failures

<table>
<thead>
<tr>
<th>Repository</th>
<th>Flakes</th>
<th>No-stress failures</th>
<th>Stress failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenHFT/Chronicle-Queue</td>
<td>6</td>
<td>4</td>
<td>38</td>
</tr>
<tr>
<td>soabase/exhibitor</td>
<td>3</td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>apache/hbase</td>
<td>2</td>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>RipMeApp/ripme</td>
<td>9</td>
<td>2</td>
<td>55</td>
</tr>
</tbody>
</table>

#### Table 4.3 Failure ratio

<table>
<thead>
<tr>
<th>Repository</th>
<th>Flakes</th>
<th>Failures per no-stress run</th>
<th>Failures per stress run</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenHFT/Chronicle-Queue</td>
<td>6</td>
<td>0.15</td>
<td>0.17</td>
</tr>
<tr>
<td>soabase/exhibitor</td>
<td>3</td>
<td>0.04</td>
<td>0.15</td>
</tr>
<tr>
<td>apache/hbase</td>
<td>2</td>
<td>0.04</td>
<td>0.25</td>
</tr>
<tr>
<td>RipMeApp/ripme</td>
<td>9</td>
<td>0.07</td>
<td>0.25</td>
</tr>
</tbody>
</table>
In this document, we presented a tool for detecting flaky tests that is capable of being integrated to any GitHub repository to be executed in CI. Results show that Shaker was successful: a testing environment under stress is on average 2.73 times more likely to find flakes. In projects like hbase, with more than 6000 tests, finding flakes can prove to be difficult due to the time it takes to execute the test suite. In our experiments, one no-stress run and two sets of stress runs were enough to trigger the 6 hours timeout and cancel the job. This shows that having an workflow on the cloud to perform these tests can save development time, also allowing this job to be executed in parallel with other tasks.

5.1 Related work

The most common technique to detect flaky tests is ReRun, which consists of re-running failing tests to determine if they are flaky or not. This approach does not actively searches for flakes and, as shown in Chapter 4, normal testing conditions are less likely to find flakes. Another problem is that an always failing test is likely to be mistaken for a regression. It may also break test order dependency.

Bell et al. [16] proposed DeFlaker, based on code coverage and marks a failed test as flaky if its coverage on the next build with changed statements included any changed code. However, this depends on the code being changed and does not work if the code is unchanged.

5.2 Future work

Because finding flakiness in code bases can’t be done in a single test run, as we need to determine if the test case both passed and failed, Shaker can be improved by using data from previous runs in future runs. This would allow Shaker to improve its detection and also be change aware, focusing on new or modified test cases. Other ways to improve Shaker are: opt-in analytics feature to help determine features that can be improved, automatically create an issue in the repository and support for more platforms.
Bibliography


[5] J. Micco, *Flaky tests at Google and how we mitigate them,* [https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html](https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html), 2016 (cit. on pp. [1, 3, 4]).

[6] *Force a validation test to be ignored in the open-source maven build... (for now).* [https://github.com/square/dagger/commit/d7fa773181a1ff1e71a591a858](https://github.com/square/dagger/commit/d7fa773181a1ff1e71a591a858), 2015 (cit. on p. [1]).


