Project Deliverable 2 - Technical Improvements Report

Group 4 - ECSE 437

Imad Dodin - 260713381
Fouad El Bitar - 260719196
Jules Boulay de Touchet - 260710129
Ege Odaci - 260722818
Irmak Pakis - 260733837
Baris Utku Cincik - 260730586

Design Report **2**
- Caching 2
- Maven → Gradle 2
- Automatically Deploying to Maven Central 2
- Travis CI → Github Actions 2

Experimental Design Procedure **4**

Experimental Results **6**

Replicability **6**

Technical Improvements **6**

I. Design Report

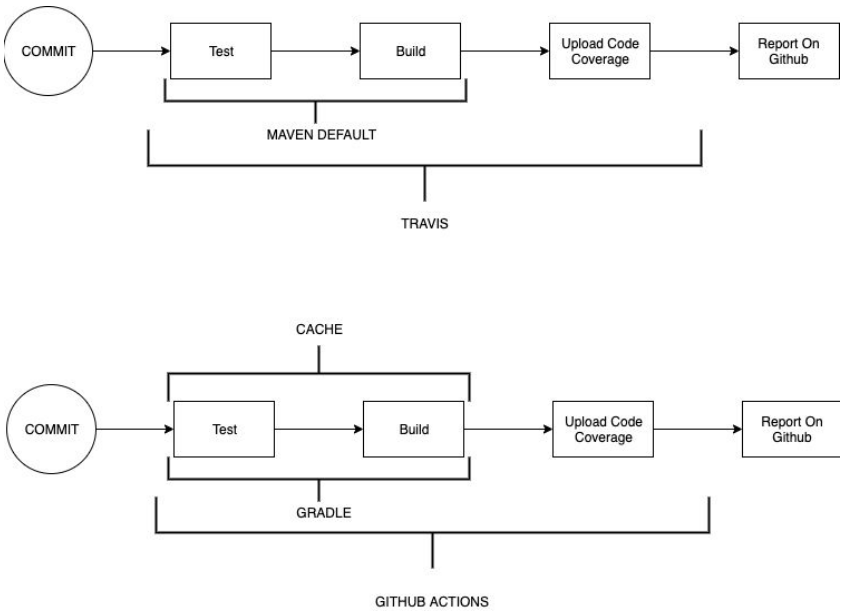


Figure 1 - IDiagram outlining the improvements to the pipeline implemented as part of our project.

A. Caching

The main objective for our improvements to Skrape.It, our chosen open source repository, is to speed up Continuous Integration time. A common technique to achieve this, is to cache reused files across builds. In Maven, for example, the .m2 directory holds the local repository for fetched dependencies. Rather than fetching these dependencies that seldom change across builds, it is advisable to keep this file cached. We demonstrate the positive effect of caching the .m2 directory, with regards to the build time elapsed, and provide our results in Section II.

B. Maven → Gradle

We similarly attempted to improve build times by migrating from Maven to Gradle. Figure X provides a chart from the Gradle site demonstrating the decrease in build time for the Apache Common Langs project after migrating from Maven to Gradle.

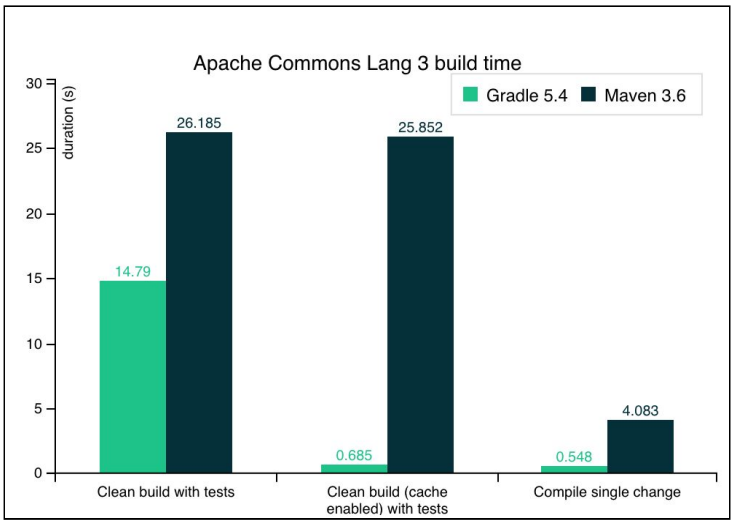


Figure 2 - Gradle and Maven Build Times for Apache Commons Lang 3 Project

We note, however, that such improvements were not experienced in our project and that, on the contrary, build times increased when we migrated to Gradle.

We chose to add the adarshr test-logger to the dependencies of the project to provide a more elegant logging of tests when executing the Gradle build.

The first design choice we made when starting the integration process was choosing which language to use for “gradle.build” file. Gradle supports both Groovy and Kotlin DSL. To be more consistent with the code base we chose to use the Kotlin DSL.

The next design choices we had to make were plugins. First, we wanted to ensure that we add the Kotlin plugin and it is targeting Java Virtual Machine. We also wanted to make sure that “JaCoCo” is added as a plugin to properly generate code coverage reports. “Dokka” is used as a documentation engine for Kotlin. We also decided to add test-logger plugin. Detekt is used to analyze code for Kotlin. These plugins were already implemented in the “pom.xml” file. We decided on keeping them.

We also decided to not use some of the plugins that were used in their implementation of “gradle build” file such as “gradle-use-latest-versions-plugin”.
Gradle-use-latest-versions-plugin updates module and gradle versions to the latest

versions. . We decided to not use “Gradle Versions Plugin” as they did. Gradle Versions Plugin is a plugin that provides a task to determine which dependencies have updates.

Instead we chose to use something like “dependabot” which maintains a more consistent and visible git history.

In the repositories section, in addition to jcenter() we added mavenCentral(). It is because “assertK” dependency is found on mavenCentral. For dependencies, we transferred all of them.

Some other design decisions:

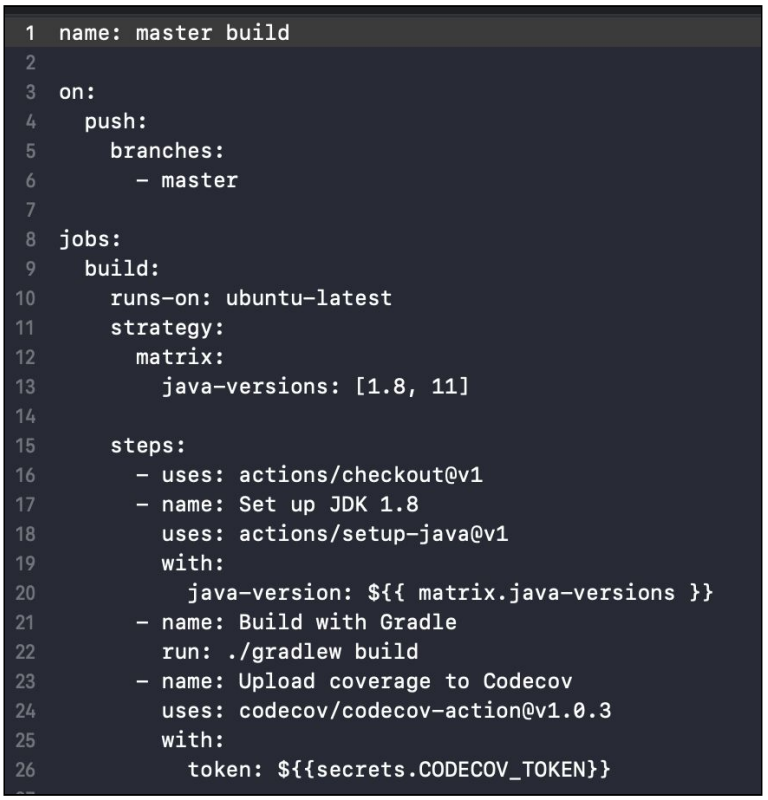
- We specified Java version compatibility to use when compiling Java source.
- We configured labels for tests such as “passes”, “skipped”, and “failed”. Also we set “slowThreshold” to identify the tests that exceeds the threshold time limit.

C. Travis CI → Github Actions

One of the advantages of using Github actions over Travis CI is being able to integrate workflows into the project. Workflows are custom automated processes that we can set up in our repository to build, test, package, release, or deploy any project on GitHub. With workflows we could automate our software development life cycle with a wide range of tools and services that are maintained on open-source. This provides a lot of adaptability and customization for relatively low effort.

Another potential advantage was not one that was substantiated in fact, but rather based on many rumours in the open source community that GitHub Actions provided faster pipeline times than its counterparts. This is not particularly difficult to believe, as one would expected lower overhead when reacting to commits and publishing feedback from within the GitHub ecosystem. We demonstrate that this is indeed the case, and that GitHub Actions provided us with a significant improvement in pipeline time, not to mention an extremely straightforward adoption.

We must store workflows in the .github/workflows directory in the root of the project. Here we have created a .yml file to store our configuration. **Figure X** below shows a screenshot of the file.



```yaml
name: master build

on:
  push:
    branches:
      - master

jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        java-versions: [1.8, 11]

    steps:
      - uses: actions/checkout@v1
      - name: Set up JDK 1.8
        uses: actions/setup-java@v1
        with:
          java-version: ${{ matrix.java-versions }}
      - name: Build with Gradle
        run: ./gradlew build
      - name: Upload coverage to Codecov
        uses: codecov/codecov-action@v1.0.3
        with:
          token: ${{secrets.CODECOV_TOKEN}}
```

Figure 3 - Github Actions Configuration File

Workflows must have at least one job, and jobs contain a set of steps that perform individual tasks. Steps can run commands or use an action. We can create our own actions or use actions that are developed by others. You can observe that on line 8 we have introduced a new job, which is the build. This job has several steps that are listed starting line 15. The first step is to set up the JDK. This step uses the action “setup-java”

The next step in the job is to execute the build process with Gradle. It should be noted that in our previous configuration file for Travis, the build process was taken care of by maven so we had a line that looked like “./mvnw clean verify”. Since we have migrated the build from maven to gradle we run the “./gradlew build” command for this step. The rationale for switching from maven to gradle is discussed separately in our report.

The next step in the job is to upload the code to Codecov to helps us calculate the code coverage which is a measurement used to express which lines of code were executed by a test suite. To accomplish this task we have used a github action called “codecov-action” which can be found here:
https://github.com/codecov/codecov-action

This way we could have all dependencies in one place, keep track of them easily and edit them when needed without trouble.

II. Evaluation Report

A. Experimental Design

For each experiment, the dependent variable is the time taken for the whole pipeline to execute. This time period consists of three main sections:

1. Time to spin up the VM.
2. Executing and running the build.
3. Uploading the code to code coverage.

In our experiment we sum up the time taken to complete these three tasks and compare the time in three different experiments:

1. **Maven build time with cache vs without**
2. **Gradle build time with cache vs without**
3. **Travis vs Github Actions time**

We have chosen the pipeline execution speed as our measurement because we believe that it is 's a good representation of the impact of the technical improvements. We contrast this value to simply measuring the build time, as we believe that the total time "from committing to feedback" is more representative of the concerns of a development team than the simple build time (additional overheads also significantly impacts the efficiency of the development team).

One other reason we have chosen pipeline execution speed is because we are in the lack of a better measurement. If the system was hosted on an infrastructure such as a server, we could measure alternate variables such as latency for requests or puppet configuration metrics etc. Since the system is not deployed on any infrastructure, however, and is a package that is installed by users rather than one users make requests to, there is a lack of any analogous metrics that are readily available to us.

In each experiment, we execute 10 Travis builds on the same codebase (within and across experiments, with the exception of changes to the CI configuration) and note the time taken to complete a matrix of two jobs: building the project on JDK 1.8 and 11. We note that we do not compare each of these jobs separately, as they are executed in parallel and do not differ significantly in execution times. We provide and discuss these results in Section III-B.

B. Experimental Results

1. Maven build time with cache vs without

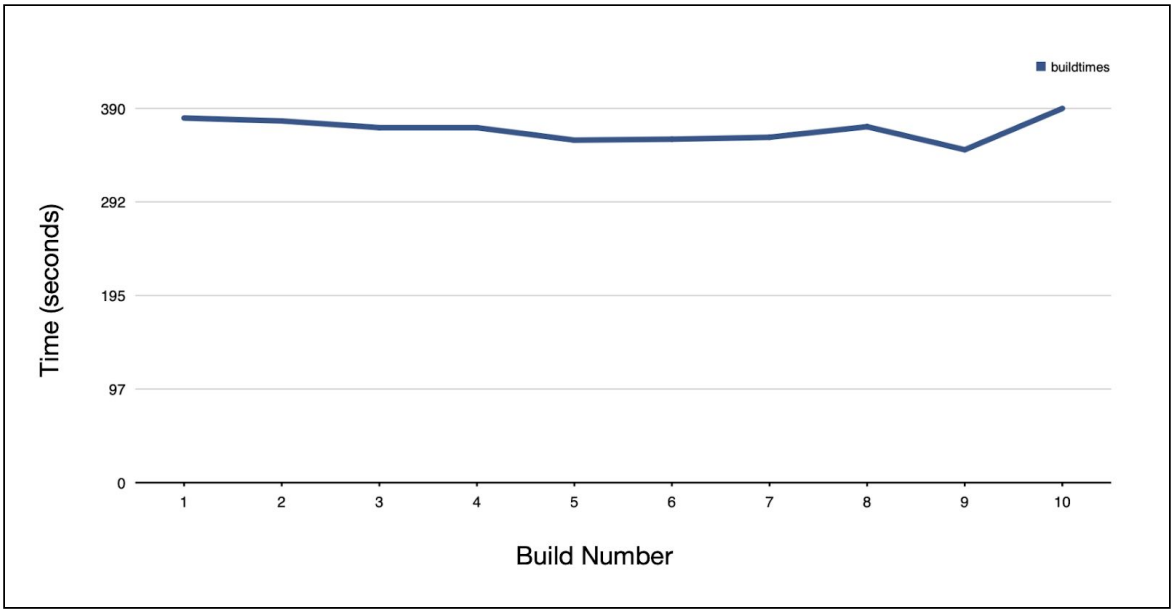


Figure 4 - Maven Build Without Cache

As would be expected, Maven Build times are fairly consistent without cache. Across our 10 build trial, we find that the average job time on Maven is 368 seconds with a standard deviation of 12.04.

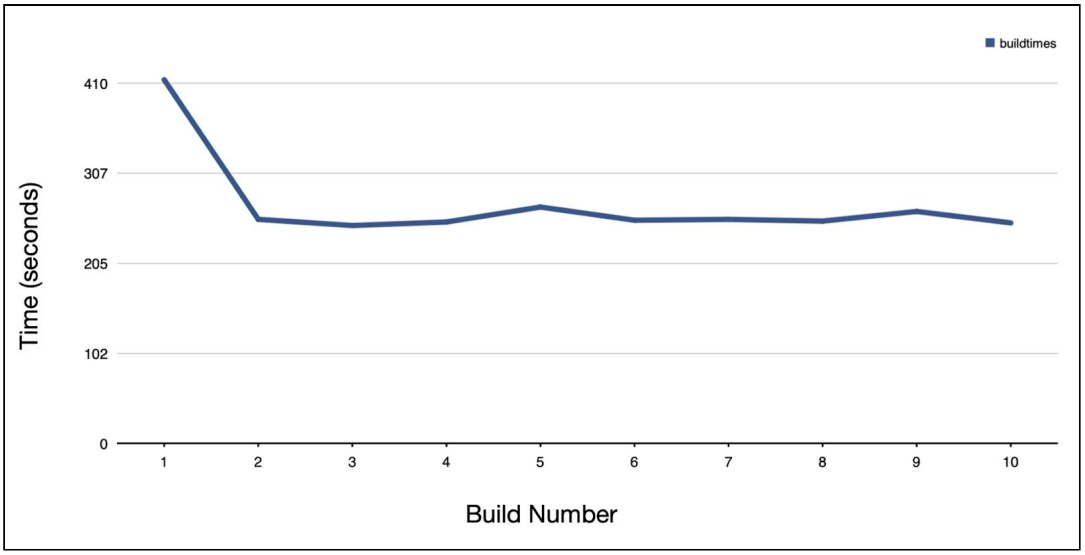


Figure 5 - Maven Build With Cache

We find, by contrast, but again as one would expect, that when caching, there is a sharp drop in build time after the first build. The noted mean job time in this case dropped down to 267 seconds - a 28% drop from without caching. By caching the .m2 directory,

we save on having to fetch all dependencies for each build, but note that this performance increase is likely to be greatly lessened with commits that introduce changes to dependencies, as the cache cannot be reused in this case.

2. **Gradle build time with cache vs without**

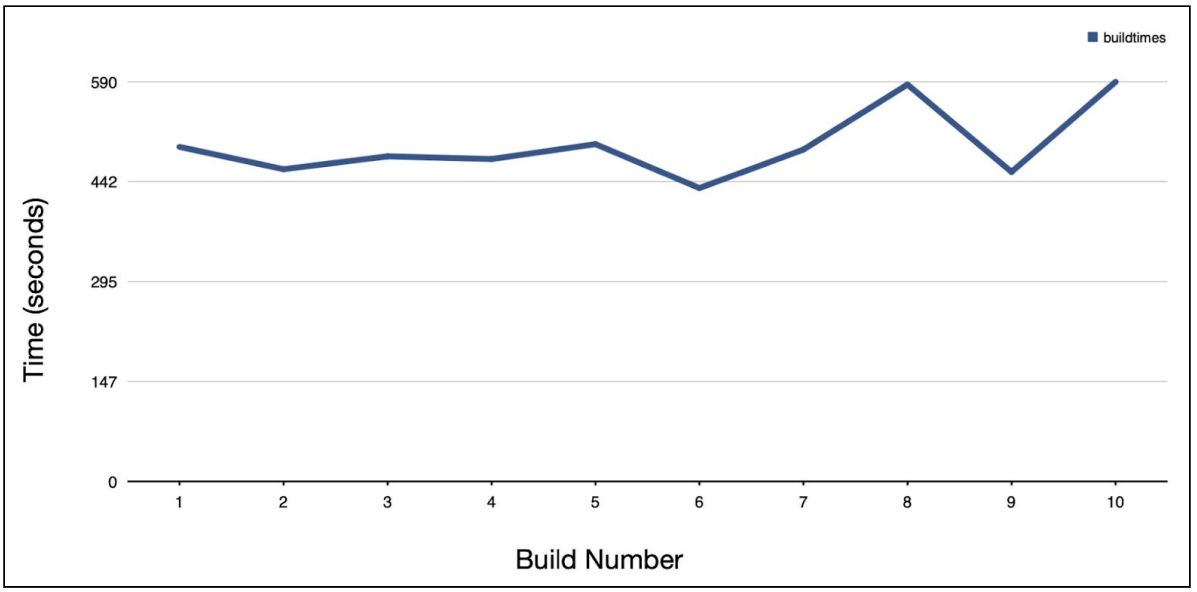


Figure 6 - Gradle Build Without Cache

We find that Gradle job times, contrary to what we expected, were significantly longer than their Maven counterparts. With a mean job time of 496.5, we note a demonstrated 35% increase in Job time when using Gradle without cache. We also note that Gradle jobs have a significantly higher standard deviation of 49.3 - suggesting that they may perform more inconsistently on Travis. This is, however, not a claim that we explore at any greater depth, as we hope to migrate to GitHub Actions in any case.

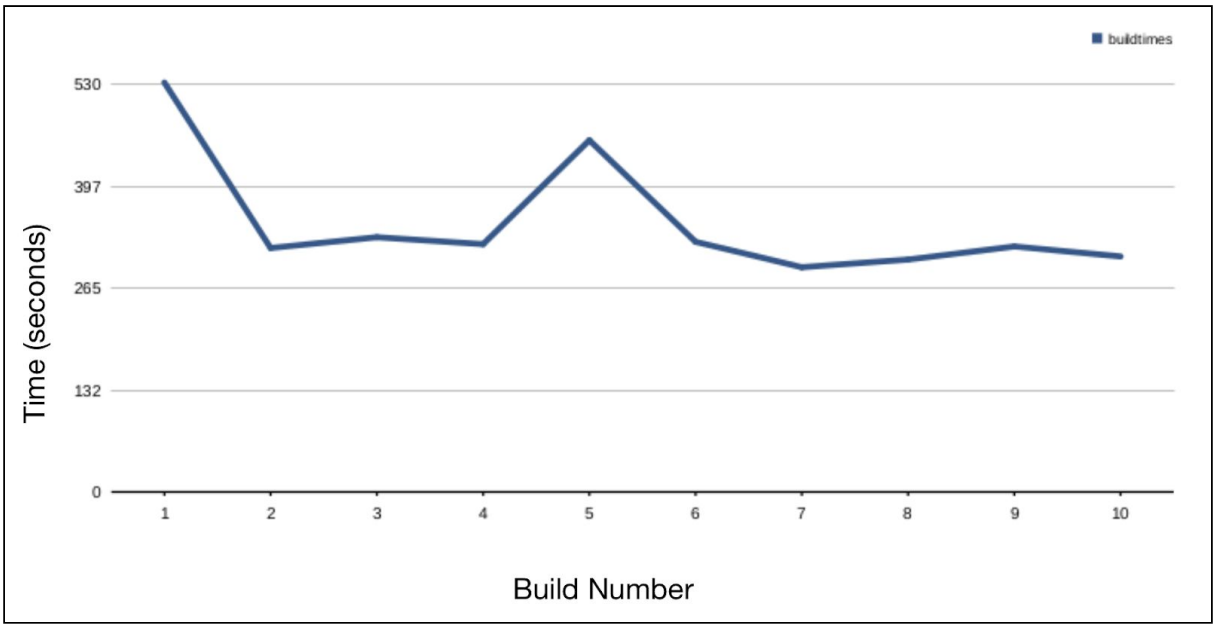


Figure 7 - Gradle Build With Caching

Similarly to Maven Jobs, we note that Gradle Jobs demonstrated a sharp decrease in Job Time after the first build, where the cache begins to be hit. For these experiments, we cache the .gradle directory which caches fetched dependency .jar files. The mean Job Time for the latter 8 jobs (9 excluding the outlier), is 287 seconds, which is still higher than the Maven mean Job Time without cache. We propose an observation that such outliers occur when Travis takes longer when fetching the cache and that this may be due to any number of reasons, such as fluctuating network latencies.

3. **Travis vs Github Actions time**

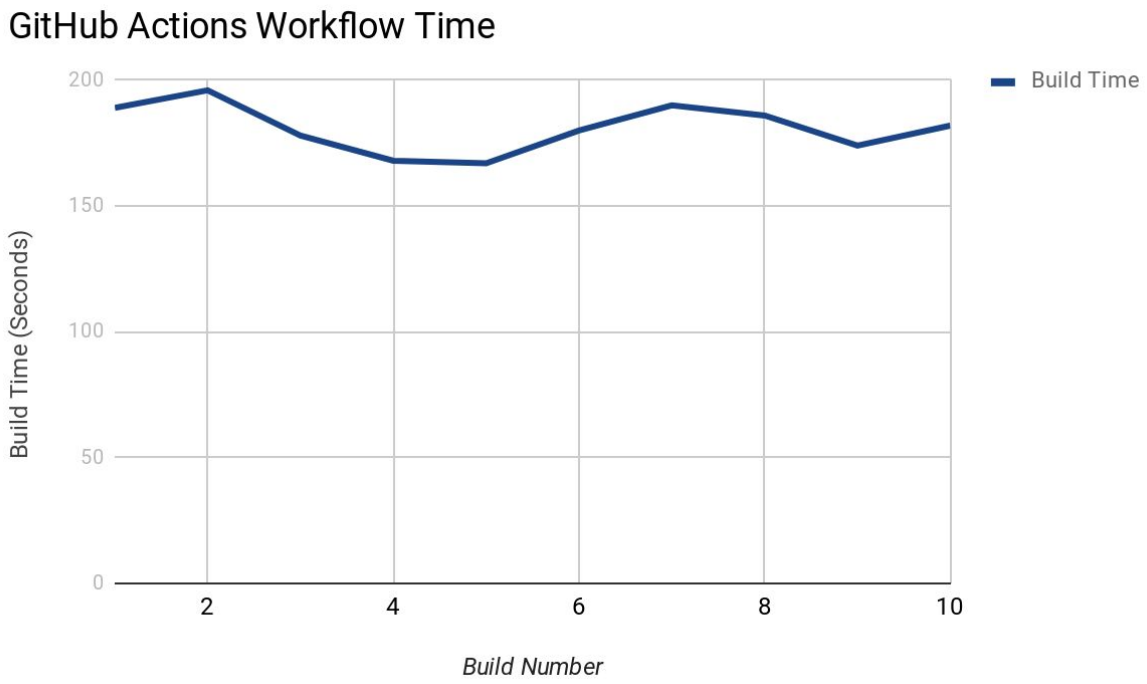


Figure 8 - GitHub Actions Workflow (CI) Time

We note an impressive decrease in pipeline time when migrating to GitHub Actions, which demonstrates an average 187 second pipeline time when building jobs with the same matrix (i.e. JDK 1.8 and 11) on Gradle. We note that these build times are experienced without caching and highlight a current drawback of GitHub Actions related to its infancy - the lack of documentation and community support. Although we were initially unable to find an action that would help us cache the dependencies, we note that we were finally made aware of such an action moments before submission.

C. Replicability

We execute our Travis experiment with use of the Travis Ruby Client and graph our results using Squid, a graphing library on Ruby. We provide the ruby script and Dockerfile used for our experiments in the experiments subdirectory on our fork for the Skrape.It system on Imad's GitHub, linked in the technical improvements section below.

Technical Improvements

The changes made for the system may, again, be found in our fork for the Skrape.It repository here. We note however, and acknowledge the contributors to the original repository who have also made some of the changes discussed in this report and whose changes, at time, were referenced when developing our own. We particularly thank Christian Draeger whose guidance was crucial in accomplishing these changes.