

1 Usage and Structure of continuous integra-  
2 tion as configuration?

3 Joseph Ling  
jl653@kent.ac.uk



School of Computing  
University of Kent  
United Kingdom

Word Count: around 6296

4 October 13, 2020

6 Continuous integration (CI) is becoming more popular as software develop-  
7 ment moves to an Agile fast paced development life cycle. Most CI is done  
8 automatically using a service which runs based off configuration. Our major  
9 questions is how much is CI actually being used? As well as how are these files  
10 being structured? We got 31,494 open source projects from Github to answer  
11 these questions. In doing so compared our results against Michael Hilton,  
12 Marinov and Dig [23] work to see if there has been an increase in usage. We  
13 found a shift in CI services being used and were able to get similar results  
14 to their study. In terms of structure we found that configuration files are  
15 written with no comments normally. We suggest at the end further research  
16 is needed to get a better understanding of this growing field.

similar is a  
bad word to  
use to de-  
scribe the  
comparison

# 1 Introduction

Continuous integration (CI) is becoming more popular over the last few years. This can be seen by how major version control hosting services Github, Bitbucket and Gitlab have all released CI products or have been improving their CI products. In terms of research, Infrastructure as Code in Rahman, Mahdavi-Hezaveh and Williams [24] which does a systematic mapping of research in that area. For Continuous Integration with Shahin, Ali Babar and Zhu [25] which does another systematic review on how it is used. These two papers demonstrate some of breadth of research that has taken place. In addition you have papers like Google's Innovation Factory: Testing, Culture, and Infrastructure Copeland [11] which demonstrate some of the depth that the papers go into.

Continuous Integration is a process of automatically compiling, running tests and checking that the product works. This can be combined with Continuous Delivery where the product is deployed or released after it has gone through successfully CI.

This can get complicated quickly therefore Configuration as Code (or Infrastructure as Code) is used to configure it. The main kind of configuration format used for this is Yaml followed by Xml and Java based scripting formats.

In order to look at our first theme CI usage we looked at In Usage, Costs, and Benefits of Continuous Integration Open-Source Projects [23]. They looked closely at the usage of CI as well. As we are looking at CI usage as well we are going answer the first three questions from their theme "Usage of CI".

- **RQ1** What percentage of open-source projects use CI?
- **RQ2** What is the breakdown of different CI services?
- **RQ3** Do certain types of projects use CI more than others?

45 We will be using doing a comparison with our corpus against theirs in  
46 order to work out what has changed over the last 4 years.

47 It would have been really interesting to do a full in depth analysis of  
48 each CI configuration format like Gallaba and McIntosh [14] does for Travis.  
49 However we can look at the general structure of all the CI configuration files  
50 allowing for comparisons to be made between configuration files. As that will  
51 allow comparisons to be made more easily otherwise comparing very specific  
52 features would have been harder.

- 53 • **RQ4** What are the common errors when loading yaml configuration?
- 54 • **RQ5** How are comments used in the configuration?
- 55 • **RQ6** How are external scripts used within the configuration?

## 56 2 Related Works

### 57 2.1 Continuos Integration

58 Continuos Integration is frequently submitting work normally tied into a  
59 feedback loop. For example using version control and committing changes  
60 daily. For each changed committed a server builds and tests the changes  
61 informing you of status of those changes. As well as providing a build which  
62 is typically a binary executable of code that can then be saved if necessary.  
63 In doing you can reduce the chances of facing the situation off “It works  
64 on my machine...”. As the building and packaging of the code is done on a  
65 server to make sure everything integrates.

66 An early definition of CI was written up and then updated later by Martin  
67 Fowler [13]. A key part of the CI is that allows teams to work on the same  
68 code base which without CI could easily lead to integration bugs and broken  
69 builds.

70 To enable to this to happen automation needs to put in place for build,  
71 testing and other aspects of the integration process in order that a clear piece  
72 of feedback (yes or no) can be given about the status of the build. If done

73 with a version control system, if the same commit is built twice (so no changes  
74 have happened) it is vital that it produces the same result. Otherwise it is  
75 hard for a team to be able to depend on CI result if they are getting flakey  
76 test results or flakey build results.

## 77 2.2 Usage of Continuous Integration

78 The actual usage of CI was looked at by [23]. In this they use three source  
79 of information Github repositories, Travis builds and a survey. The impact  
80 of CI usage on Github was looked as well in Vasilescu et al. [28]. The key  
81 difference is that we will be looking at the configuration alongside the CI  
82 usage and leaving impact of CI on the project to further research or current  
83 existing research. In analysing that data they found that “The trends that  
84 we discovered point to an expected growth of CI. In the future, CI will have  
85 an even greater influence than it has today.” As we are looking at the same  
86 question we will use four of the research questions out of the fourteen (4.1,  
87 4.2, 4.3, 5.1). In order to see what difference four years has made to the  
88 growth of usage of CI.

## 89 2.3 configuration as code

90 Configuration as code or Infrastructure as Code has been an increasing area  
91 of research over the last few years. There seems to be slightly more re-  
92 search in infrastructure as code, for example see Rahman, Mahdavi-Hezaveh  
93 and Williams [24]. There has been a focus on Puppet and Chef, for example  
94 Sharma, Fragkoulis and Spinellis [26] looked at code quality by the measure  
95 of “code smell” of Puppet code. This tackles the problem by defining by best  
96 practices and analyzing the code against that. In the case of Cito et al. [10]  
97 it uses the docker linter in order to be able to analyse the files. For the CI  
98 systems we pick we will look into the tooling around that to aid the analysis.

## 99 3 Methodology

100 In order to answer the research questions we needed to find projects for CI  
101 configuration files. This is because we needed to get the contents of the  
102 configuration in order to analyse the structure of it. We chose to scrape  
103 Github via their API as it was easy to setup and test whether or not it  
104 was working as there is a one-to-one mapping between the API and user  
105 interface. Using Ghtorrent GhT [4] it may have been easier to gather more  
106 data because the rate limiting wasn't as strict. Yet harder to test whether or  
107 not it is working. Therefore we decide to use the Github API as the source  
108 for our corpus.

109 We chose to use a config file to specify which CI systems configuration  
110 files we would look for. If it was a directory then it would get all ".yaml" or  
111 ".yml" along with any Teamcity ".kts" and ".xml" files. However the script  
112 did not look into any of the sub directories which might be the cause for the  
113 low number of Teamcity configuration files found. In the case that it was a  
114 file that was on the top level directory we matched it with the lowercase file  
115 name we found against the query.

116 In terms of which configuration files to pick we based our list from Github  
117 Welcomes all CI Tools blog post in 2017 [16]. In addition we added Github  
118 Actions and Azure Pipelines to the list as they are new and potentially  
119 popular systems.

120 As can be seen in Figure 3 a query based on the number of stars a project  
121 has on Github. This is because we need a way of getting a large sample from  
122 Github without introducing too much bias into the sample. That is not too  
123 say that our method is perfect but it provides an easy way to get a large  
124 sample that includes projects with and without CI. Another potential solu-  
125 tion would have been to use the "filename:travis.yaml" search API. However  
126 this did not provide information about which projects did not use CI. Ad-  
127 ditionally for one unique search there can only be 1000 results returned by  
128 the Github API. To mitigate that limit we search based stars as we did do  
129 a search for a 1000 results per star count. The limitation of this though was

















Branch: master <span>New pull request</span>	
JosephLing Create test.yml	
 .github/workflows	Create test.yml
 output	added song beamer co
 .travis.yml	Create .travis.yml
 Jenkinsfile	Create Jenkinsfile
 README.md	added song beamer co
 example.log	powerpoint support ac
 hymns.txt	init TODO: unicode ern
 main.py	added song beamer co
 modernWorship.txt	init TODO: unicode ern
 notWellKnown.txt	init TODO: unicode ern
 powerpoint.py	added song beamer co
 requirements.txt	init TODO: unicode ern
 scaper.py	added song beamer co
 songbeamer.py	added song beamer co
 worshipNight_1.txt	fixed unicode errors an
 worshipNight_2.txt	fixed unicode errors an

Figure 1: Example Github repository that has multiple configuration types in it [21]. (This is an old repository that was reused in order to test out the scraper)

```
PATHS = {
    "travis": "travis",
    "gitlab": "gitlab-ci",
    "azure": "azure-pipelines",
    "appVeyor": "appveyor",
    "drone": "drone",

    "jenkinsPipeline": "jenkinsfile",

    "teamcity": ".teamcity/",

    "github": ".github/workflows/",
    "circleci": ".circleci/",
    "semaphore": ".semaphore/",
    "buildkite": ".buildkite/"
}
PATHS_MULTIPLE = ["github", "circleci", "semaphore",
                  "teamcity", "buildkite"]
NONE_YAML = ["jenkinsPipeline", "teamcity"]
```

Figure 2: Python configuration file used to specify what types of configuration to search for. The key specifies the name of the configuration and the value is the location in the repository the config should be found.

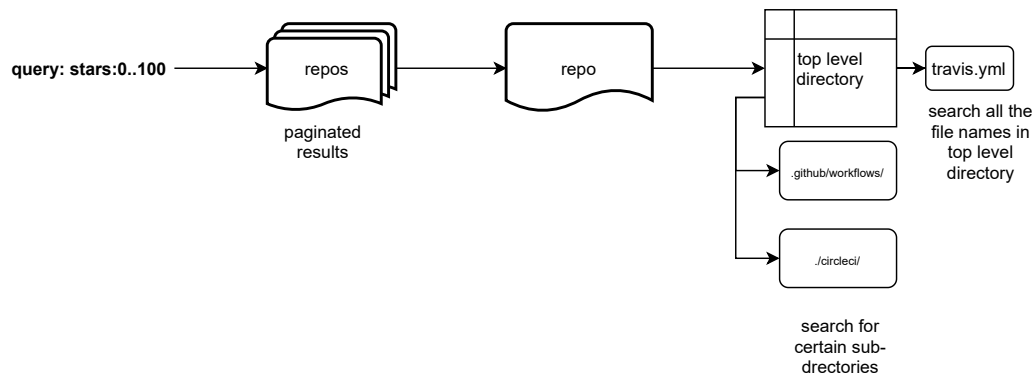


Figure 3: Diagram of the process used to search for projects with CI files in them

that there will be over a 1000 repositories that have 0 to 500 or even 500 to 501 stars. That means it is a sample that represents some of the population but not a sample of all CI files on Github.

As the config could have mistakes in it or we missed out a major CI system. We also saved the ReadMe.md when we scraped each project. A Readme.md is used to describe a project and will be displayed on Github at the bottom of the root directory. As can be seen in Figure 4 some ReadMe's have a label and/or links to the CI system used for that project. Therefore we also saved that data when we scraped a project.

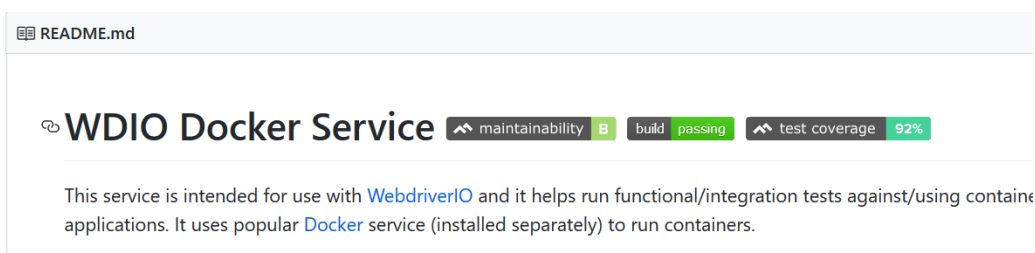


Figure 4: Example of CI tag for Github ReadMe [27]

We ended up with searching for the following CI systems: Travis, Gitlab, Azure, App Veyor, Drone, Jenkins, Github, Circleci, Semaphore, Teamcity and Buildkite.



142 We excluded Wrecker from the search because they represented a very  
 143 small number of projects in comparison to the other projects. As it seems  
 144 since the Github survey in 2017 they got bought by Oracle and from doing  
 145 a search on Github for what we think based on the docs [30] and [18] for  
 146 their configuration file naming convention. We were only able to find 20  
 147 results so did not include in the scraping script to speed up the process of  
 148 searching for the other configuration file formats. Despite it being used in  
 149 the Michael Hilton, Marinov and Dig [23] paper.

150 Along with information of what CI is being used for a project we also  
 151 gathered metadata about the project. The available metadata through the  
 152 API is largely what can be seen on a repository for example in Figure 5.  
 153 We have the star count which is an indication how popular a project is as  
 154 users can star projects that they like Borges, Hora and Valente [9]. Then  
 155 we have watchers which is users that have subscribed to the project to get  
 156 notifications about the project.

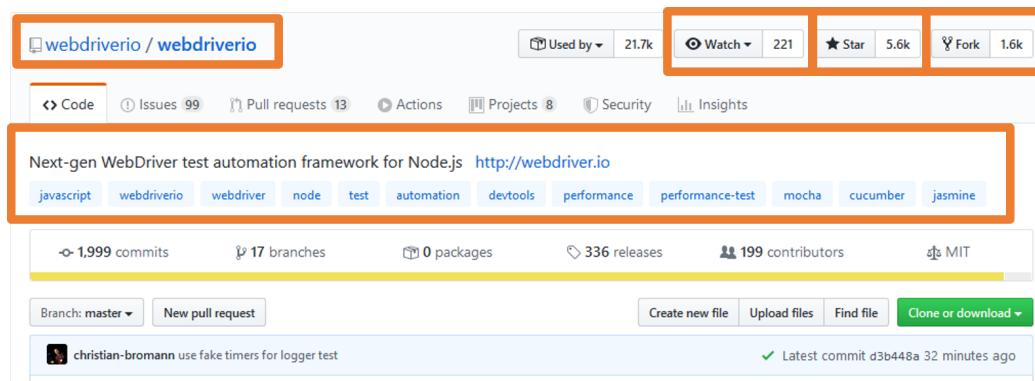


Figure 5: Example Github project description and metadata[29]. The orange sections sections highlighted are the metadata that we scraped.

### 157 3.1 Data corpus

158 This all produced a sample of 32,660 projects from open source projects on  
 159 Github. As can be seen in Figure 6 we weren't able to scrape the whole  
 160 star count range easily. This is because the script would crash when Github

161 gave a 500 error code at us randomly. Along with empty repositories initially  
162 causing a problem. In order to mitigate the damage of this the scraper would  
163 create a new Comma Separated Value (csv) file search e.g. one for stars:0..1  
164 and another for stars:1..2. As all the csv file contained the same header we  
165 ran a script to combine all together at the end. Making sure to remove any  
166 duplicates by filtering on the Github project id.

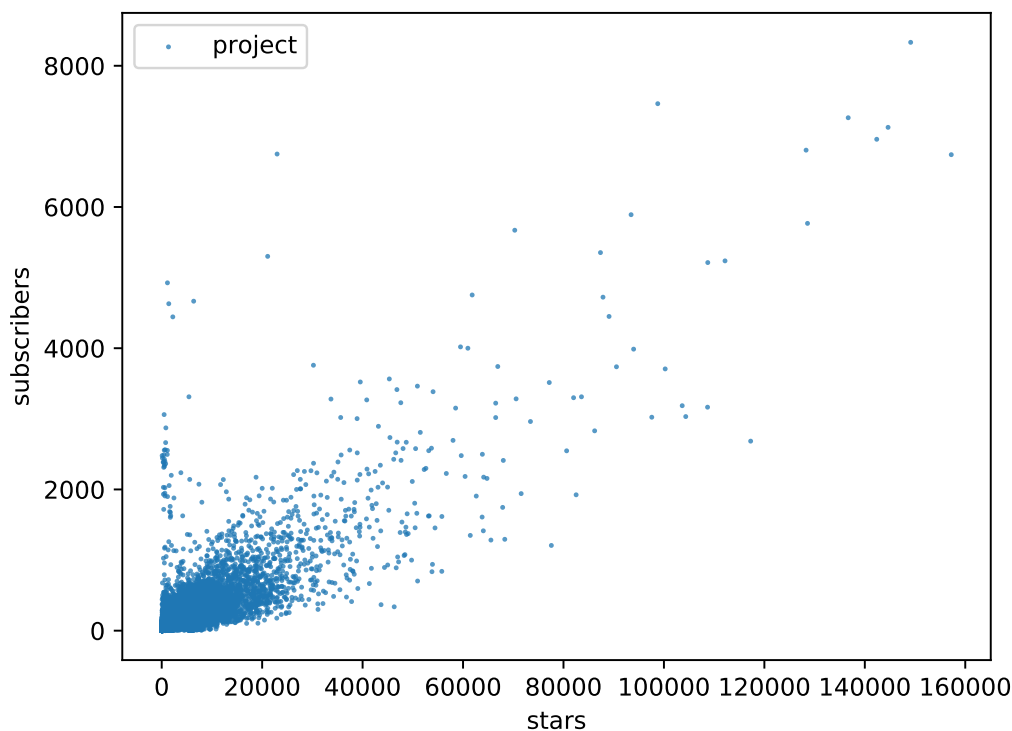


Figure 6: Github stars against subscribers

## 167 3.2 Comparison corpus info

168 In Michael Hilton, Marinov and Dig [23] paper they use a similar method  
169 of using the Github API in order to create their corpus. Additionally they  
170 contacted Cloud Bees Clo [2] to get a list of all open source projects that  
171 used their services. This helped them not to miss out on projects that they  
172 would otherwise missed out on. They kindly gave a copy of their final corpus.

173 However it does contain the data on the Cloud Bees projects which is  
 174 223 projects. As far as we can tell this only affects the comparison done in  
 175 RQ2 4.2. Additionally we found slight discrepancies between the paper and  
 176 corpus in RQ1, RQ2 and RQ3, these being mainly mainly just a few numbers  
 177 off in a few places. In order to do comparisons well and to keep it consistent  
 178 we will be basing all our comparisons from the corpus. As the discrepancies  
 179 are small we will still be using the conclusions from the paper where possible.

180 In order to get a better understanding of the results of the methodologies  
 181 chosen in both cases. We created two histograms to showing the density by  
 182 the stars of the spread of data using the Sturge's rule (Figure 9). As we  
 183 expected both corpses are skewed to the left. They use a logarithmic scale  
 184 and initially it can look like there more data in 2020 corpus. However it is  
 185 just more spread over the stars and potentially had a larger search range  
 186 used. In Figure 7 there is data after 40,000 stars potentially this could be  
 187 down to the search method used or that most projects weren't dispersed over  
 188 so many stars on Github.

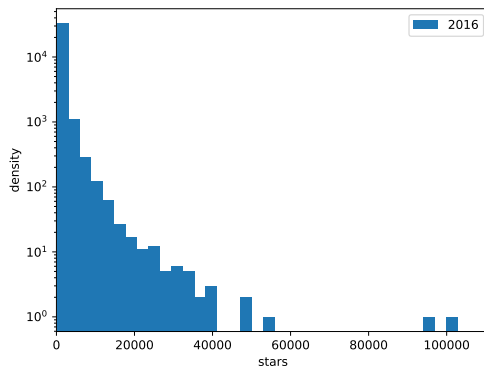


Figure 7: 2016 corpus

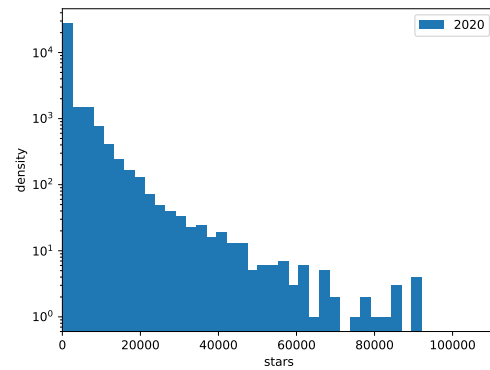


Figure 8: 2020 corpus

Figure 9: Histogram showing the density for both corpses via the stars of the projects. They are both skewed towards to the lower star count projects.

## 4 Usage of CI

### 4.1 RQ1: What percentage of open-source projects use CI?

Out of the 31,752 projects 38.39% of them had CI configuration files in them indicating that they used CI in our dataset.

	2016		2020	
	count	percentage	count	percentage
<b>Found CI</b>	13752	39.81%	12538	38.39%
<b>Number CI found</b>	20792	60.19%	19214	58.38%
<b>ReadMe has CI status</b>	n/a		908	2.78%
<b>Total</b>	34,544		31,494	
<b>Multiple CI</b>	1796	12.91%	1764	14.07%

Table 1: This table shows the comparison between the 2016 dataset which is Michael Hilton, Marinov and Dig [23] and our dataset labeled as 2020. For the CI usage in each dataset along with what percentage of projects contained multiple CI setups.

An interesting factor in Table 4.1 is the percentage of that 38% that has multiple CI in them. This is because configuration files can be used to CI or Continuous Deployment (CD) and some projects are run a monorepo which means that they have multiple projects inside them. Another simple explanation is that although the configuration is stored version control it just hasn't been deleted.

We scraped the "ReadMe.md" files from the projects to check if they had a CI status label in them as shown in Figure 4. To do this we checked for `alt="Build Status"`, `alt='Build Status'`, `Status` and `status` being in the file. Then if that same line of text contained a url specified by if contained `http://` or `https://` then we counted it as potentially being a project that used CI. In order to check the validity of this method we ran it on all projects that we had found configuration files for. We got 6782

207 (55.92%) projects with a ReadMe that had a CI status label that we could  
208 find. However this method is not perfect, for example “awesome-bootstrap-  
209 checkbox” by “flatlogic” [3] their ReadMe has the following line:

```
210     [![Dependency Status]  
211     (https://img.shields.io/david/dev/flatlogic/awesome-bootstrap-checkbox.svg?branch=master&style=flat)  
212     ]  
213     (https://www.npmjs.com/package/awesome-bootstrap-checkbox)
```

214 This contains **Status** and a url so we say it has got CI when the repository  
215 currently doesn't. Yet this is not the case for all of them as for example  
216 “SyncTrayzor” by “canton7” [22] uses AppVeyor but doesn't use a configura-  
217 tion file for it. Therefore we didn't find it as we searched for a configuration  
218 file only.

219 The percentage of CI projects in 2016 was 39.81%. If you look at Table 1  
220 it shows that we got 38.51% CI projects. This is interesting as we searched  
221 for more kinds of CI configuration so there was a potentially a higher chance  
222 of having CI.

223 One possible reason could be because in RQ3 4.3 it shows that the more  
224 popular a project the higher chance it has of using CI. Therefore as their  
225 sample contains a few more projects that are popular they could all be more  
226 likely to be using CI. However that is a weak tangent to make in order to  
227 full explain it.

228 Another possible reason could be if you combined the “Found CI” and  
229 “ReadMe has CI status” results together for 2020 you would get 41.28%  
230 which is shows that our sample is within the margins of the same results  
231 that of CI usage for 2016.

232 A further possible explanation Github's growth over the last 4 years (Git  
233 [5] to 2019 Github [17]) so that Github is now at 40 million active users. It  
234 means that there are more projects that are using CD setups for building  
235 their static sites and in general Github is being used for more things that  
236 wouldn't require CI.

237 We think that the last two factors are the most likely contributors to  
238 why there is less CI usage now. Another important interesting aspect is that

239 despite Github growing so much the CI usage rate has stayed relatively the  
240 same.

## 241 4.2 RQ2: What CI systems are projects using?

242 In Table 2 we, like all other research, found that Travis is the most popular  
243 CI system in use. However over the last 4 years the [16] CircleCi has lost out  
244 on it's rough quarter that it owned in the market. Notably the rise of Github  
245 Actions seems to have taken second place in the market share even though it  
246 is still very young in comparison as it was officially released November 13th  
247 2019 but had a closed beta since the summer of 2019. This might not be  
248 down to the CircleCi losing out on their existing share, but maybe due to  
249 the rise in CI usage on Github. As well as because we searched Github for  
250 projects Github's CI product would be popular on their platform.

Table 2: Configuration types spread

	config	percentage
Travis	10607	74%
Github	2301	16%
CircleCi	1109	8%
Jenkins pipeline	161	1%
Drone	84	1%
Buildkite	32	0%
Teamcity	4	0%
Semaphore	2	0%
Azure pipeline	1	0%

251  
252 Our sample size of repositories is 31,494 that as it is a representation  
253 of projects on Github so won't account for the whole of it. This means  
254 that although Wrecker had the smallest count of CI when researching of 20  
255 projects. In Table 2 we have configuration types that have lower counts.  
256 This is because that search for the 20 searched the whole of Github but the  
257 scraping was only able to do a small sample. Additionally there potentially

double check  
this para-  
graph!!

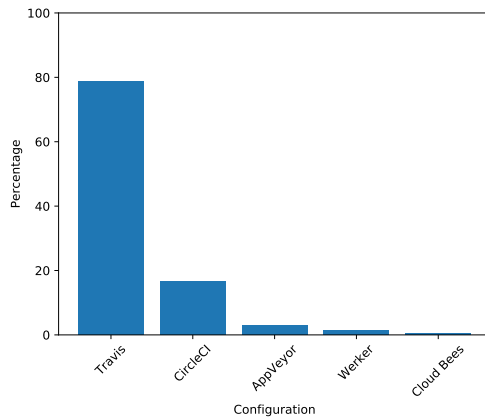


Figure 10: 2016 corpus

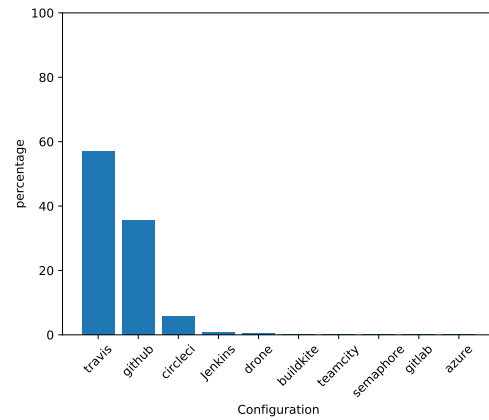


Figure 11: 2020 corpus

Figure 12: Percentage bar graph showing the usage of each CI service. The key difference is how CircleCI has got a lower rank. Due to rise of Github Actions which only open to closed beta in August 2019

could be faults in the scraping causing it show such low numbers for the last 3.

### 4.3 RQ3: Do certain types of projects use CI more than others?

In Figure 15 shows all the CI projects sorted then grouped together per 540 projects. The number used to group the projects was chosen based off the average group size used when Michael Hilton, Marinov and Dig [23] conducted this question. Then in this case we choose to categories via star count for each project.

In Figure 13 and 14 we are comparing whether or not in the last 4 years the number of stars increases the amount CI being used. It shows how the trend in the more popular the project by how you have more stars for a project increases the chances it uses CI has stayed the same. However the gradient of that trend which has changed to be slightly greater overall, yet not quite as sharp for the end of the graph, this is most likely because the 2016 dataset doesn't have as much data between 40000 and 90000 as seen in

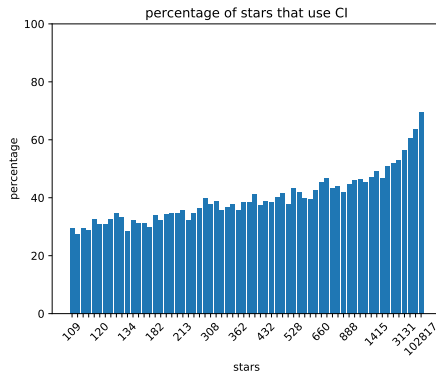


Figure 13: 2016 dataset

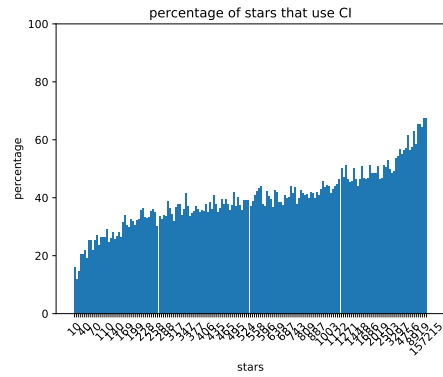


Figure 14: 2020 dataset

Figure 15: In Figure 13 is the results from this research and in Figure 14 is the results from [23]. The results show the percentage chance of CI usage depending on the number of stars a group of 540 projects has on average.

274 Figure 7.

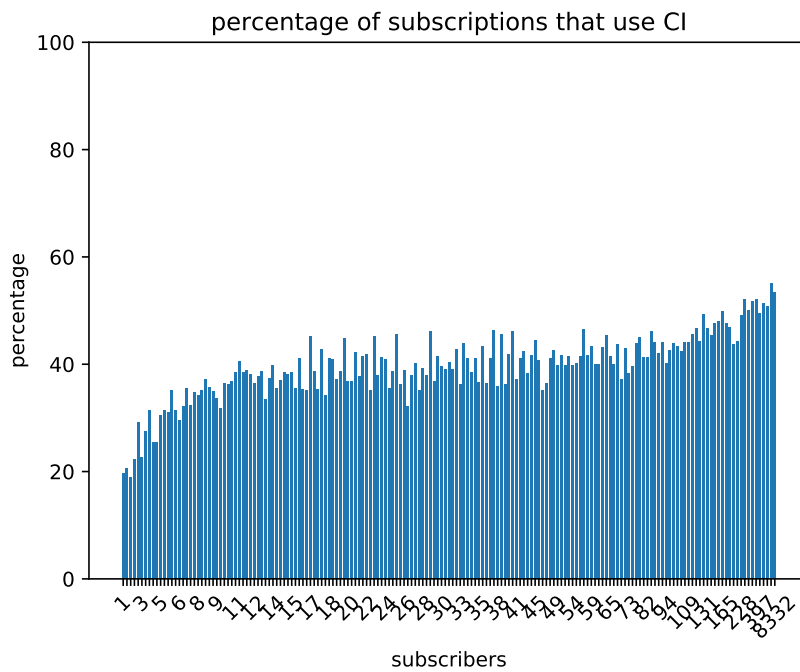


Figure 16: Subs graph



275 Figure 16 uses the same method as Figure 15 except it does it based  
276 the number of subscribers. Subscribers are users on Github who want to  
277 keep update on the changes to a project. This could range from core team  
278 members working on the project to people that want to be notified about a  
279 new release. In looking at this metric the hypothesis was that it would have  
280 a sharper rise in percentage of projects using CI per subscriber. However  
281 that was not the case overall as the gradient is not as strong. There is no  
282 comparison to [23] because their final corpus does not contain the subscriber  
283 count for each project.

284 That gives us a good look at how projects can be viewed through Github's  
285 metadata.

286 Next we turn to what kind of programming languages are being used for  
287 CI. Figure 17 shows the top 20 programming languages by count and we can  
288 see that Javascript is the most common kind of project by a considerable  
289 margins. This was to be expected as it ties in with Github's annual report  
290 [17] on the platform they reported that Javascript has been the most pop-  
291 ular programing language for the last 5 years. The interesting part is that  
292 our sample matches the rise in Python over Java. This is despite the fact  
293 that they are using "unique contributors to public and private repositories  
294 tagged with the appropriate primary language" and we are using the count  
295 of projects by primary programming language tag.

296 In order to get a better idea of the breakdown of the effect programming  
297 languages have on CI usages we created Figure 18. This shows three pieces  
298 of information the percentage of CI usage on the y axis, average star count  
299 on the x axis and then number of projects using the language by the size of  
300 the dot.

301 The most striking part of Figure 18 is the clear divide between differ-  
302 ent programing languages star count. The programming with the languages  
303 with the highest CI usage are Rust (94.6%), Typescript (86.56%) and Go  
304 (78.31%). This is interesting in that they are all fairly "new programming  
305 languages" in comparison to the others in the graph. They are all languages

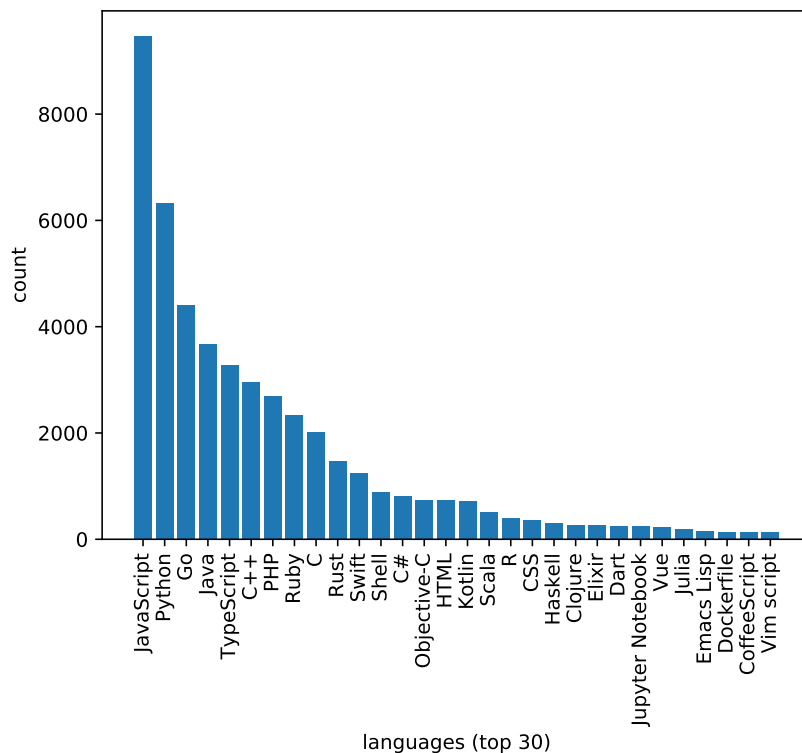


Figure 17: Count of top 20 programming languages used by projects using CI

306 which are developed and open source on Github. In terms of Rust and Go  
 307 it could be down to their tooling that comes built in to the language as  
 308 that would lead to implementing CI to be a lot easier. Typescript is more  
 309 a special case as it is a subset of Javascript so uses ‘npm’ to deal with de-  
 310 pendency management which was some of the inspiration for Rust’s tooling  
 311 Rus [1]. Older programming languages like Java and C# both have tooling  
 312 for dependency management but the chances that they use CI is much lower.  
 313 Therefore an area for further research would be whether or not the use of  
 314 “modern” dependency management systems increases the chance of CI.

315 In figure 19 it shows the sample from 2016 in comparison to 18. The first  
 316 major difference is in how the spread of languages by stars isn’t as divided.  
 317 This could be because of how the sample is not spread out as seen in Figure

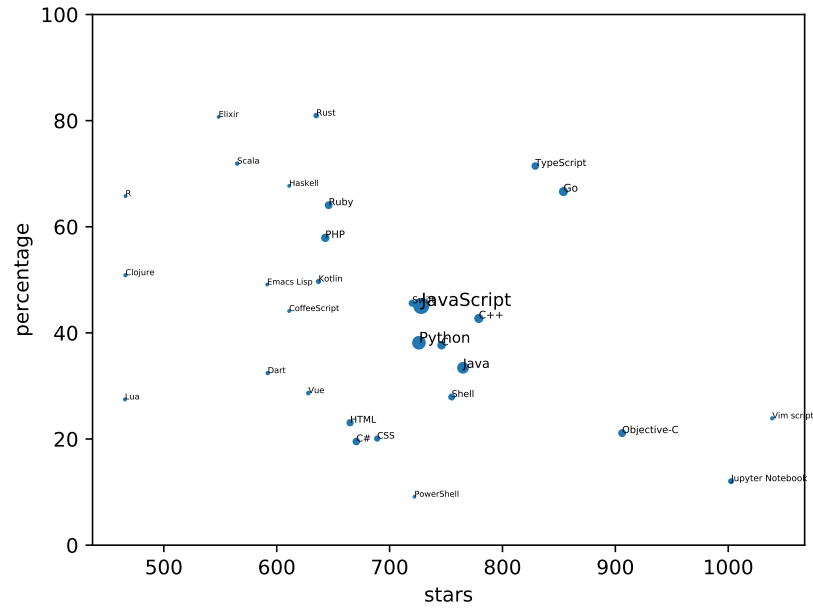


Figure 18: Scatter graph showing the top 30 most used programming languages against how much they use CI. The key points are Rust, Typescript and Go being the top three programming in CI usage.

9. The scatter graph is for the top 30 most common programming languages found and from this it can be seen that Rust, Typescript and Scalar have the highest chance of using CI. This is really interesting Rust and Typescript are still really within the top 3 after 4 years. Potentially this could be to do with the ecosystem around the languages that lead this. However area for further research is looking into why different languages have a higher chance of using CI.

Finally one observation that was made in the Michael Hilton, Marinov and Dig [23] paper was that there was a higher chance of CI usage for dynamically typed languages. We looked into analysing this as we found both Rust and Typescript having really high CI usage probability, yet at the same time Javascript and Python had the most projects overall that used CI. So we wanted to look at where the balance lied in the difference between the two. However categorising the programming languages by their usage is difficult.

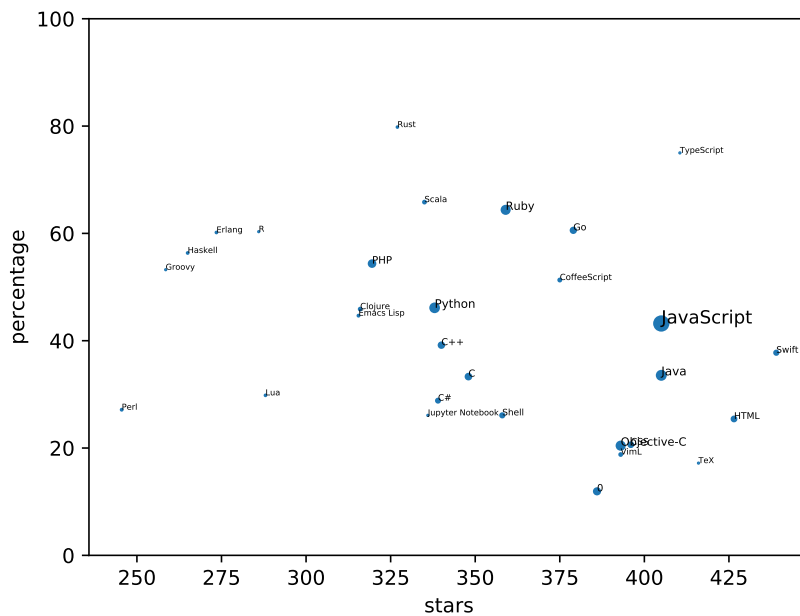


Figure 19: Scatter graph showing the top 30 most used programming languages against how much they use CI for Michael Hilton, Marinov and Dig [23] from 2016. In comparison to Figure 18 the data is not as grouped as clearly by the star count. Rust, Typescript and then Scala have the highest programming CI usage.

For example is a Javascript a project that is using Typescript's js checking dynamically typed or statically typed? And then how do you tell? Or if you have a similar situation where Python has static types added into through a library. Therefore this an area for further research as it is a question that would need to be carefully answered.

Overall the popularity of the project increases the chances of it using CI. The programming language has an effect on the chances of it using CI. However what properties of the language cause this effect is unclear so is an area for further research.

## 341 5 Structure of configuration files

342 The following three research questions will focus on the 14,302 CI projects  
343 (found CI plus multiple CI 4.1). In order to be able to ask the questions  
344 about the data we filtered the sample to only include CI projects. Then we  
345 created a csv table with a row per CI type in that project as some projects  
346 had multiple versions of CI as shown in 4.1. Then we processed each CI file  
347 to get the necessary data to be able to ask questions about it's structure as  
348 we wanted to be able to process files with or without errors in along with all  
349 types of CI. We created a parser to go through each line of the configuration  
350 file working out what that line is. For example is it a comment or blank line  
351 or does it have code.

### 352 5.1 RQ4: What are the common errors when loading 353 yaml configuration?

**Scanner error** The first step of loading the yaml is to scan it to create the tokens. However for example tabs are not allowed in yaml YAM [8] as seen in the example with “\t” representing a tab.

```
definitions:  
  \t- build
```

**Parse error** In this example it has scanned the file and created tokens for the syntax. Now it parses the syntax and works out if each token is valid given it's current context. In this case a closing ] without an opening [ is invalid.

```
definitions: ]
```

354 As can be seen in the Table 3 there are configuration files with yaml errors  
355 meaning that the CI for that project will not load correctly. Yet it seems

**Composer error** In the example it has two steps that are using a yaml anchor. This allows for the yaml to be referenced somewhere else. However if you define the anchor twice with the same name it causes a composer error, as you have two references using the same name so it won't know which one to use.

```
definitions:
  steps:
    - step: &build-test
      name: Build and test
      script:
        - mvn package
    - step: &build-test
      name: deploy
      script:
        - ./deploy.sh target/my-app.jar
```

**Constructor error** In Yaml if you have a ! it will treat the build\_matrix as a tag. Then as it is a tag it will require it to have a constructor in what loads in the configuration. In this it could allow for a default or complex build matrix to be made.

```
definitions:
  steps:
    - step: &build-test
      name: Build and test
      matrix: !build_matrix
```

Table 3: yaml configuration errors

config	composer error	constructor error	parse error	scanner error	no. config
buildkite	0	0	0	1	100
circleci	6	0	3	3	2790
drone	70	1	0	0	211
github	0	1	4	28	17631
semaphore	0	0	0	3	28
travis	22	0	27	56	28143
azure	0	0	0	0	1
gitlab	0	0	0	0	1

356 that a very small percentage of projects that have them. For example the  
 357 two highest configuration types with errors are Drone (36.90%) followed by  
 358 Travis (0.348%).

359 In the case of Drone all the errors are for the same type of error. Po-  
360 tentially this could be because of how anchors are use a lot more in Drone  
361 configurations.

362 Travis is the largest configuration type out of the sample by a significant  
363 amount it is more likely to contain more errors. Yet with such a small amount  
364 it seems like yaml errors aren't a major problem in CI. Although as they are  
365 required to be fixed in order for the CI to run the chances are the ones with  
366 errors ones that are being changed when the scraping was being done. This  
367 means that as the CI has been set up correctly for the other 99.632% as  
368 they are not needing to change because their our no yaml errors in it and  
369 presumably it is doing what they intend for it to do.

## 5.2 RQ5: How are comments used in configuration?

check this

The assumption was that continuous integration setups can be complicated and have edge cases, therefore comments in the configuration would be used to describe and handle that complexity.

An example of this in Figure 20 for Github Actions shows a number of the cases of comments. The first being including useful information about why a particular version of the programming language was chosen. The second is that the tests have been disabled by commenting them out.

Figure 21 gives the initial breakdown of the structure of the files. This is achieved by counting the amount of blank lines, code and comments used up in each file. One of the key findings from this was the lack on average (mean) of comment usage.

Figure 22 shows only the yaml based configurations, in order to get a better understanding of comment usage for them. As comments weren't used on average so much that in 21 the breakdown as visible. The other really interesting finding was that the average line count for Travis CI files was the smallest. This is because Travis is the most popular CI service so there would be a higher chance to have large configuration files.

Figure 22 shows how the comments are broken down for each CI service

comment type	count
comments	5
single line comment	1
multiple line unique comments	1
multiple line comments	4
code with comments	0
file lines	18
blank lines	0
code	13

Table 4: Line structure analysis of Figure 20



```

name: Python package
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Python

    uses: actions/setup-python@v1
    # note: only works with python 3
    with:
      python-version: 3.8
      - name: Install dependencies
    run: |
      python -m pip install --upgrade pip
      pip install -r requirements.txt
      # - name: Test with pytest
      #   run: |
      #     pip install pytest
      #     pytest ./src

```

Figure 20: In order to pick up on all these different types of comments. All the CI files were parsed and then regular expressions were used to pick on up key factors such as “note:” along with multiple single line comments which made up a block/multi-line comment. For example in to the above there is an example Github Action yaml file. If were it would be parsed we would get: Table 4.

389 on average. In the case of Azure our sample size is only one project so  
 390 that doesn’t give us any significant insight. The blue line which represents  
 391 code that has a comment after is the most commonly used kind of comment.  
 392 Apart from for Drone which has more multiple line comments. This is really  
 393 interesting as it highlights that comments tend to be tied to the code that  
 394 is written. We had expected multiple line or single comments to be the  
 395 most common kind of comment used. This is because code tend to follow a  
 396 style guide to limit the maximum characters for a single line, for example in  
 397 Python’s PEP [6] and Ruby’s RUB [7] which both use the same commenting

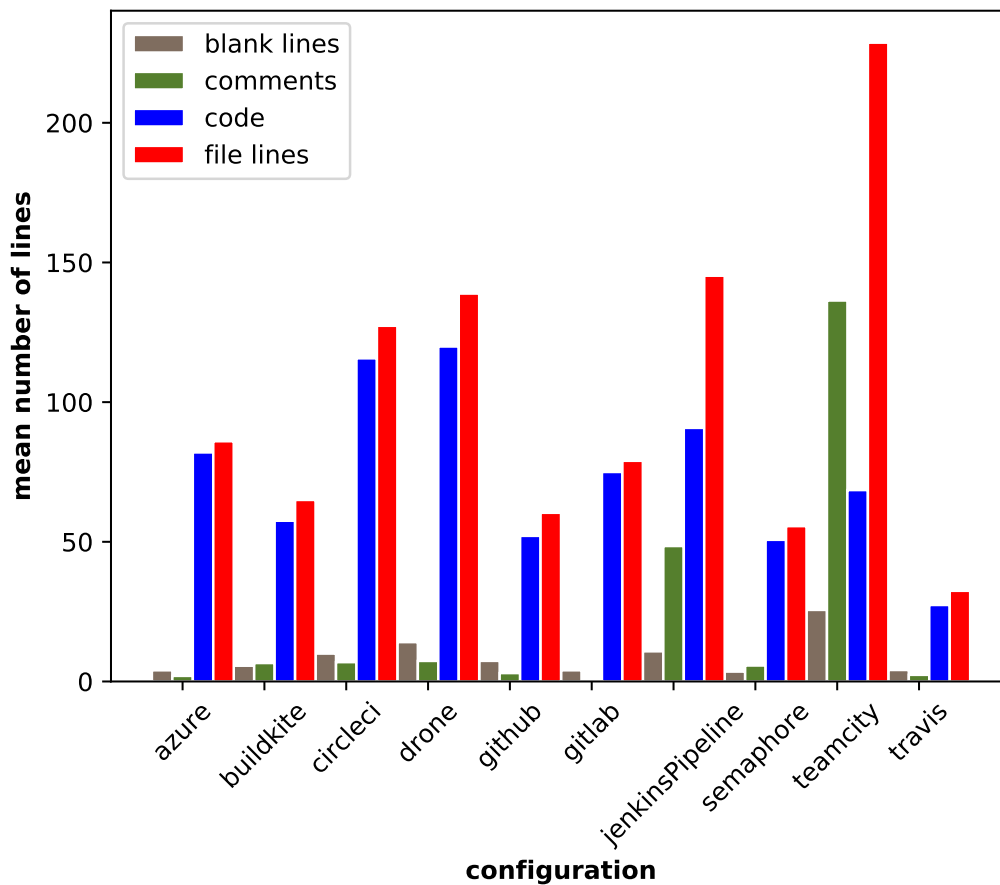


Figure 21: Mean of line counts for all CI configuration types. Showing the blank, coded and commented line average breakdown.

398 syntax style guides. In doing so this rule also applies for comments as well  
 399 therefore you don't normally have space for a code and a comment. However  
 400 the code with comments is the most common type of comment shown in 23.

401 In the case of Jenkins pipelines and Teamcity there is a much higher usage  
 402 of having code with comments. Therefore we have separated the analysis and  
 403 comparison from Figure 21 to Figure 24 for the non yaml based configuration.  
 404 Jenkins and Teamcity configurations is Kotlin based and for TeamCity is  
 405 also xml based. The key difference we find here is that multiple line comments  
 406 are more common than code with comments when compared to the yaml

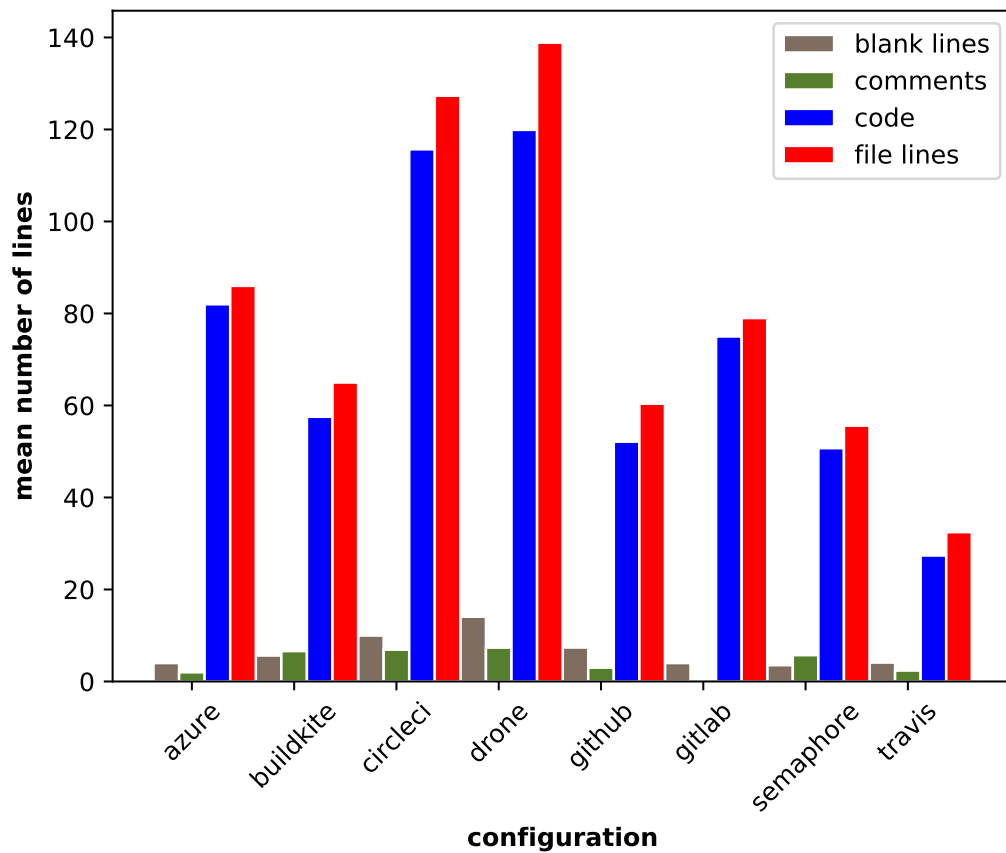


Figure 22: Mean of line counts but only for yaml based configuration. Giving a clearer view on the breakdown of how little blank lines and comments are used.

407 configuration. The second key difference is a much higher mean number of  
 408 lines for comments. These two difference are probably combined because both  
 409 Kotlins and xml allow block comments which allow for multi-line comments  
 410 to be done easily.

411 We have looked at the structure of the file and the what kind of comments  
 412 are used. Then we looked at the structure of the comments. In order to  
 413 do this we created a list of regular expressions used to categorise how the  
 414 comments were structured.

415 From labelling the comments as shown in Figure 25 we can see that

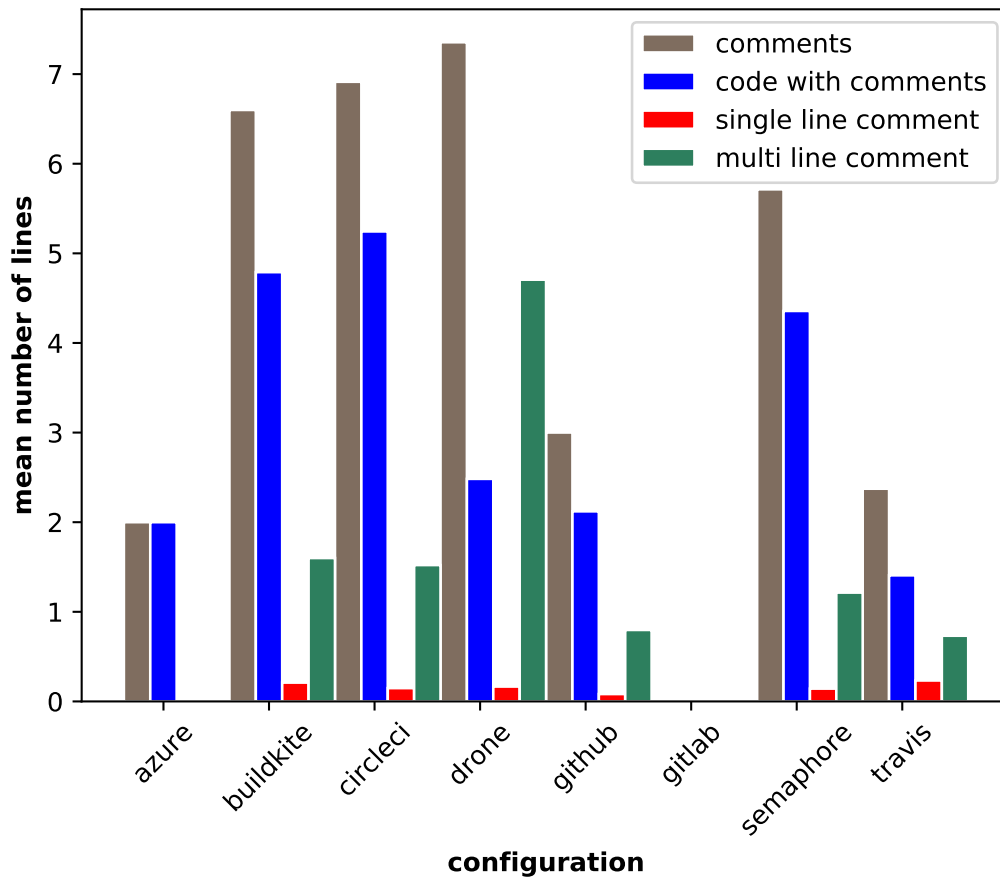


Figure 23: Mean of line counts for comments, code with comments, single line comments and multiple line comments for yaml configuration files

416 having comments with versions in and urls is most common. This could  
 417 indicate comments from templates or how they are commented. Although  
 418 yet again the amount of labels found on average is still very low.

419 Overall we have found that comments are not used a lot. However where  
 420 they are used they tend to be comments on the same line as code. In the  
 421 cases that they are used it's more likely to be from a configuration template  
 422 or commenting out configuration.

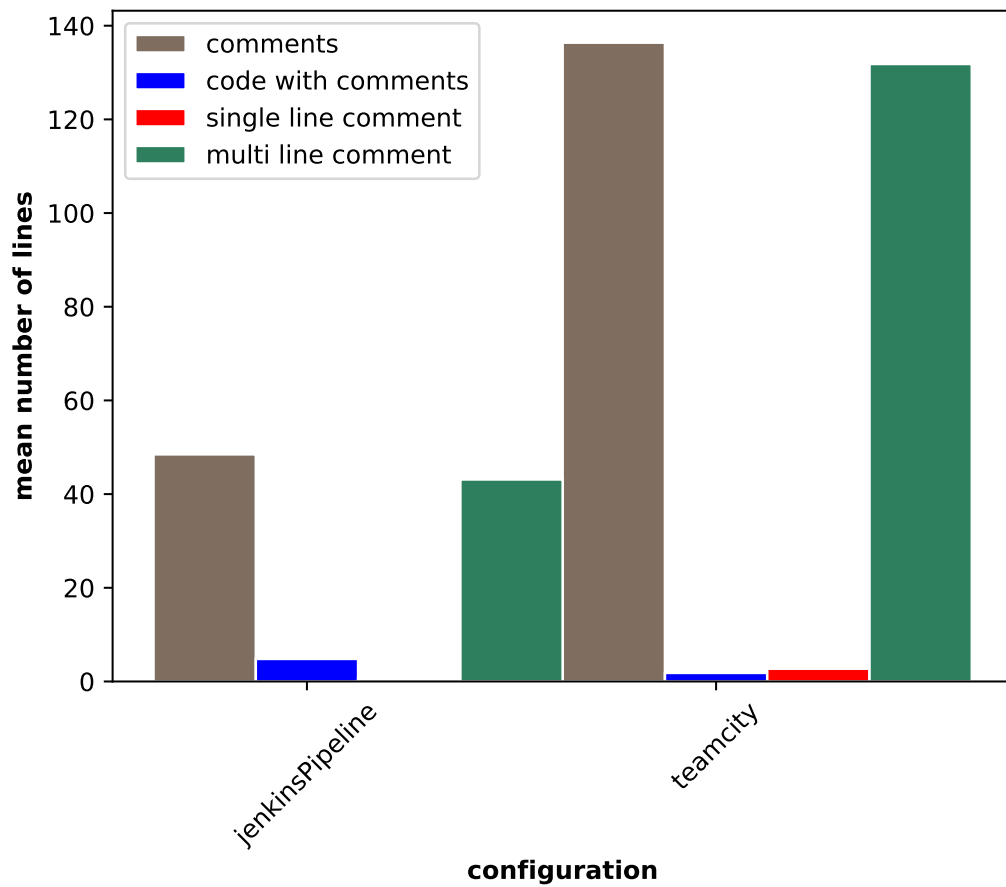


Figure 24: Mean of line counts for comments, code with comments, single line comments and multiple line comments for none yaml configuration files

### 423 5.3 RQ6: Are external scripts used within the config- 424 uration?

425 An external script is typically a Bash or Powershell script, depending on the  
426 operating system. It can be used to build, deploy or do any step that CI  
427 takes. The key difference between it and the CI configuration is that it is  
428 executed on a users machine. Therefore you can get some setups where you  
429 have scripts defined for building and deploying the code that the users and CI

In Figure 25 a regular expression was used to label the comments. There were key different types of comment that we wanted to find. The first being the commented out code which we did by searching for version numbers in comments. The second being useful information about the structure of the CI file such todo, note, important comments (e.g. `//todo`). In order to increase the search for this we included searching for urls and separation comments (e.g. `//===`).

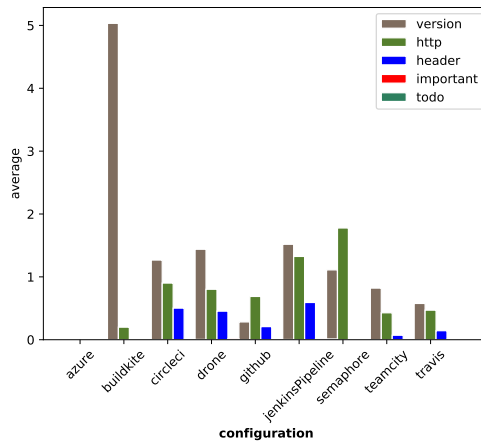


Figure 25: Comment types

both use. Most CI systems allow for “script” tags to be used which could be described as an internal script. Therefore external scripts are defined outside the CI configuration in the directory.

The methodology we used to handle this was to look at how many bash or powershell scripts where used in CI. Using the code that parsed the yaml files for comments we were able to do a check using a regular expression for either of those files.

Figure 26 shows the average script usage per CI service. We were surprised that on average multiple scripts were used for each across each CI service. This could be for a number of reasons as CI can be also be used for deployment either to production or for setting end to end testing environments. Part of CI is to be able to have the software build on any machine. In order to do this scripts can be used to simplify the process for the developers and also when the CI is running in the CI service. This is to avoid the “it works on my machine” situation. Another potential reason is that it is easier to write the logic needed for the CI process in the script than in the configuration for the CI. These are interesting results and in order to be able

447 know the reasoning for the high numbers further research needs to be done.

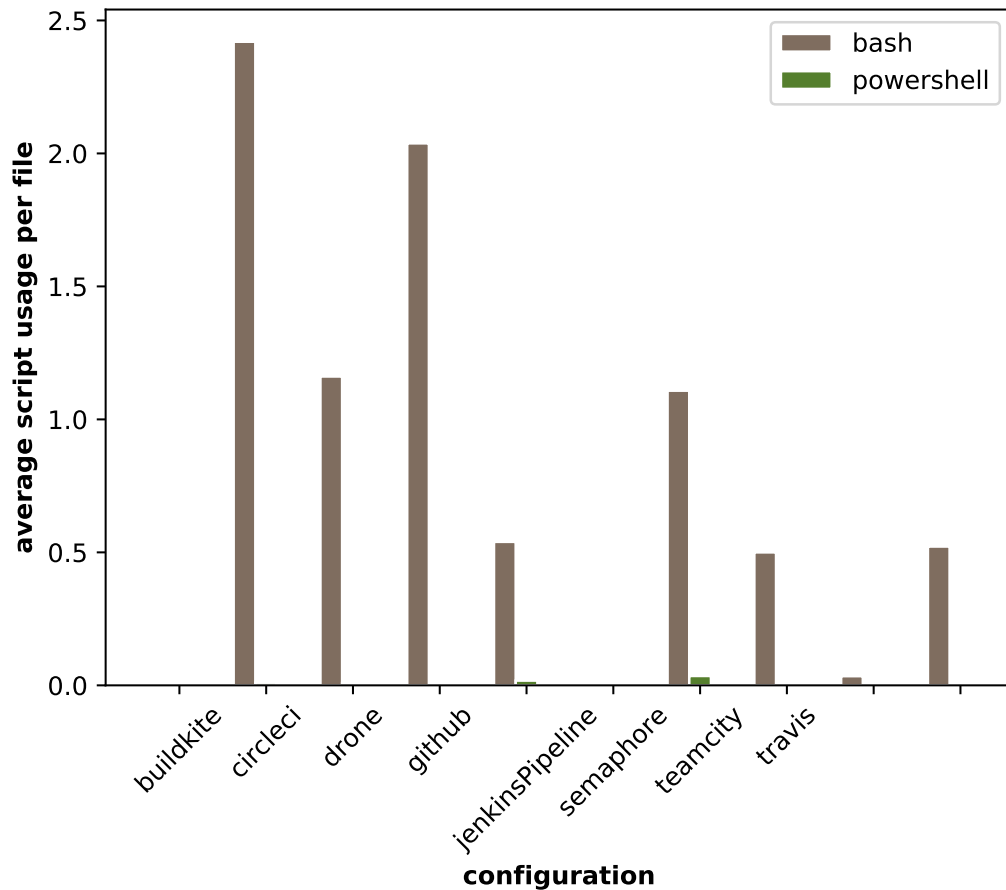


Figure 26: Mean script usage for each file

448 In Table 5 we can see the raw count found for each CI service for Bash  
449 and Powershell. We were surprised to see the percentages for external scripts  
450 to be so high as we expected that use of internal script and functionality that  
451 the CI configuration enabled would used relied on more.

452 As some of the logic is defined in the external scripts and not in the CI  
453 file. Potentially there could be less lines of code in the configuration for  
454 files that use scripts. However in Figure 27 we can see that there is little  
455 to correlation between script usage and number of lines. Then in Figure 28  
456 we can see the same affect when trying to see if the popularity of a project

	<b>bash</b>		<b>powershell</b>		<b>total config</b>
	count	percentage usage	count	percentage usage	
<b>Buildkite</b>	61	190.62%	2	6.25%	32
<b>CircleCi</b>	1497	134.99%	8	0.72%	1109
<b>Drone</b>	230	273.81%	0	0.0%	84
<b>Azure</b>	0	0.0%	0	0.0%	1
<b>Github</b>	1097	47.67%	65	2.82%	2301
<b>Jenkins Pipeline</b>	171	106.21%	0	0.0%	161
<b>Semaphore</b>	2	100.0%	0	0.0%	2
<b>Teamcity</b>	0	0.0%	0	0.0%	4
<b>Travis</b>	5937	55.97%	3	0.03%	10607

Table 5: Raw count and percentage chance of each CI service containing an external script

457 affects the chances of it using CI.

458 Figure 27 shows the number of lines against the number of scripts used.  
459 There is a slight increase in the number scripts used per number of lines.  
460 However it is a only very slight increase.

461 Figure 28 shows that there is no clear correlation between popularity and  
462 external script usage. The graph more closely follows the same shape as  
463 Figure 6. However much like that other graph there is no clear correlation  
464 to be made. Yet there is a clear increase in scripts used after 10,000 stars  
465 which is hard to view on the graph. But as the inclusion of a external script  
466 is a change because of the amount of potential extra logic added to the  
467 configuration file it is significant change.

468 Overall we can see that external scripts are used at least once or more in  
469 configuration files. The larger the configuration file their is a slightly higher  
470 chance in more external scripts being used. Further research would need to  
471 be done to understand at what stage in the CI process they are used.



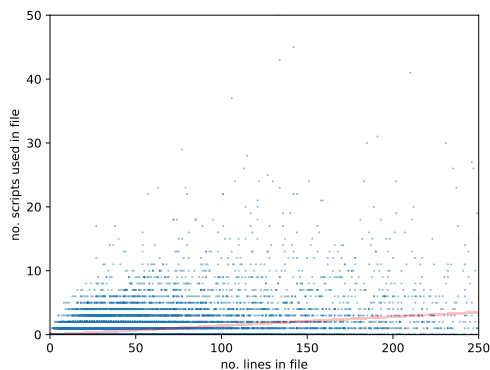


Figure 27: number scripts to number lines with the extreme values cropped out. This shows a slight trend in the more lines you have the more scripts you will use.

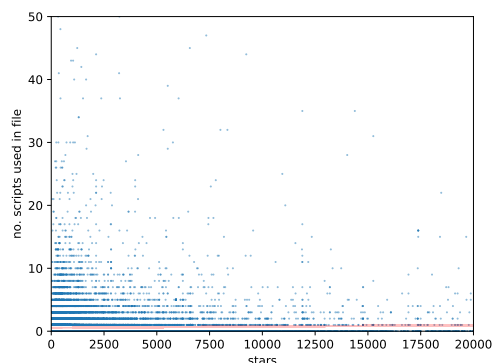


Figure 28: number scripts to stars with the extreme values cropped out. As we are comparing stars along the x axis it looks similar too Figure 6. This figure focuses on the first 20,000 stars as that is where the data is most dense. We found a slight increase in usage of scripts the more popular a project became. Yet this is only a correlation and further research would need to be done to work out if there is a causation

## 472 6 Threats to validating

473 The major and most obvious threat is the sample's reliability and validity as  
 474 it was gathered from scraping the data from Github. This has already been  
 475 touched on in the 3 section but now we are going to look at it in more detail  
 476 critically.

477 Firstly if we assume that the scraping works perfectly then it only has  
 478 a maximum of 1000 open source projects per star. That is excluding closed  
 479 source projects which would range from personal projects to companies. Ad-  
 480 ditionally we only used data from Github not from Gitlab, bitbucket or other  
 481 version control hosting services. This leads to bias in the data for example if

482 Gitlab was also scraped then we would get a lot more Gitlab CI files. How-  
483 ever in order to get the best spread of data Github has the best API and  
484 most of their services do not tie you down to use only their service. Also al-  
485 though we could get a 1000 projects per star we were still able to get around  
486 30,000 projects and a wide spread across Github. The key aspect being that  
487 because it was a sample we focused on getting a good spread of data.

488       Secondly, the scraping script is not perfect in how it finds configuration  
489 files. This is because it only looks in the top level directory for the file name  
490 pattern described in their docs or unique folder. Therefore if the systems  
491 allowed many different names or different names in the past it wouldn't have  
492 picked it as a CI system. Additionally we only decided to scrape for certain  
493 CI files, yet we chose a good scope based on previous research into the top CI  
494 files. The scraping script has been tested and worked on to try and minimise  
495 any bugs. In the case that we did not pick up a CI file we ran a regexp  
496 against the ReadMe file to get a better understanding of the error bounds.

497       Thirdly, identifying which projects are programming projects or would  
498 have a need for CI on Github and which are not. Based on the research [20]  
499 it is important to filter out repositories that aren't part of the question being  
500 asked. Therefore we could have looked to try and filter out Github static  
501 sites and other none software based projects. However if assume a certain  
502 type of project won't be using CI then we would be introducing bias when  
503 trying to answer how CI is used. For further research better categorising  
504 of projects would help a lot. The major difficulty would be if for example  
505 you had a static site that ran on Netlify that will be using CI/CD but very  
506 different from a Travis setup. Being able to get the correct data to be able  
507 to distinguish and analyse both situations.

## 508 7 Summary

509 We got a sample of 32,660 open source projects from Github and were able  
510 to compare that to a previous study 4 years ago. In doing so we found that  
511 usage of CI projects was similar and that the more popular a project the  
512 higher chance it would be using CI. This linked with the research from 4  
513 years ago. The major change was the increase in the popularity of Github  
514 Actions, which took over second place from CircleCi. Additionally we looked  
515 at whether or not the number of people watching the project had the same  
516 effect, it did but to a lesser extent.

517 In terms of structure of CI configuration we looked each line of was used  
518 in context of comments. We found that a very few projects use comments in  
519 their CI. In terms of how they used scripts, we found the majority of projects  
520 do not use external scripts.

521 From this a better understanding of this topic could be gathered by look-  
522 ing further into the data gathered, as we found we were faced with a lot more  
523 questions while doing this research as we go into below.

### 524 7.1 Discussion and further research

525 In the process of writing this paper we kept on considering more research  
526 questions, as there is a lot of meta data that you can get for a single project,  
527 in addition to what was used for this paper.

528 Further research into usage that we would like to do is to look into how  
529 the size of the project affects the chance that it uses CI. Then looking at  
530 the usage of scripts within CI configuration, for example using a script tag  
531 to run a shell script, as while doing the research we found some projects use  
532 scripts a lot while others just used the CI configuration. This would lead to  
533 questions around which CI system have a higher amount of scripts used, but  
534 also looking at how much they enable them to be used and what is the size  
535 of those scripts. The data for the programming language and version(s) is  
536 in the configuration. Therefore it would be possible to work out how much  
537 usage of a particular programming language each version is getting.

Further research into structure could look into the naming of each part of the build process that is used. This would be interesting as it would provided insight into what terms are commonly used, as well an idea into how people plan or don't plan out their configuration files. Additionally CI systems can be designed to run on every commit to version control or only commits to certain branches. Therefore by looking at the branching regexp that are being used a better understanding of how branches are actually used in software development where CI is also used could be found out. In particular looking into which branching method (e.g. [12], [15], [19]) is used more for projects with CI and those that don't.

## 8 Acknowledgement

We wish to thank Michael Hilton in particular for providing the corpus for their research Michael Hilton, Marinov and Dig [23].

## References

- [1] (2020). Cargo: Rust's community crate host | Rust Blog.
- [2] (2020). Cloudbees website.
- [3] (2020). flatlogic/awesome-bootstrap-checkbox.
- [4] (2020). Ghtorrent website.
- [5] (2020). GitHub State of the Octoverse: 2016.
- [6] (2020). PEP 8 – Style Guide for Python Code.
- [7] (2020). >Ruby Style Guide.
- [8] (2020). Yaml faq.

- 560 [9] Borges, H., Hora, A. and Valente, M. T. (2016). Understanding the  
561 Factors That Impact the Popularity of GitHub Repositories. In *2016*  
562 *IEEE International Conference on Software Maintenance and Evolution*  
563 *(ICSME)*, pp. 334–344, iSSN: null.
- 564 [10] Cito, J., Schermann, G., Wittern, J. E., Leitner, P., Zumberi, S. and  
565 Gall, H. C. (2017). An Empirical Analysis of the Docker Container  
566 Ecosystem on GitHub. In *2017 IEEE/ACM 14th International Confer-*  
567 *ence on Mining Software Repositories (MSR)*, pp. 323–333, iSSN: null.
- 568 [11] Copeland, P. (2010). Google’s Innovation Factory: Testing, Culture, and  
569 Infrastructure. In *Proceedings of the 2010 Third International Confer-*  
570 *ence on Software Testing, Verification and Validation*, Washington, DC,  
571 USA: IEEE Computer Society, ICST ’10, pp. 11–14.
- 572 [12] Driessen, V. (2013). A successful git branching model - git flow.
- 573 [13] Fowler, M. (2010). Continuous integration.
- 574 [14] Gallaba, K. and McIntosh, S. (2018). Use and Misuse of Continuous In-  
575 tegration Features: An Empirical Study of Projects that (mis)use Travis  
576 CI. *IEEE Transactions on Software Engineering*, pp. 1–1.
- 577 [15] Github (2017). Github flow introduction.
- 578 [16] Github (2017). Github welcomes all ci tools. In github.com, ed., *Github*  
579 *welcomes all ci tools*.
- 580 [17] Github (2019). Octoverse - top languages.
- 581 [18] GitHub (2020). github filename search for wrecker.yml files.
- 582 [19] Hammant, P. (2013). What is trunk based development.
- 583 [20] Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M.  
584 and Damian, D. (2014). The promises and perils of mining GitHub.

Hyderabad, India: Association for Computing Machinery, MSR 2014, pp. 92–101.

[21] Ling, J. (2019). Cu worhsip song list creator - a repository taken over for testing.

[22] Male, A. (2020). canton7/SyncTrayzor. Original-date: 2015-02-08T17:08:40Z.

[23] Michael Hilton, K. H., Timothy Tunnell, Marinov, D. and Dig, D. (2016). Usage, costs, and benefits of continuous integration in open-source projects | Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.

[24] Rahman, A., Mahdavi-Hezaveh, R. and Williams, L. (2019). A systematic mapping study of infrastructure as code research. *Information and Software Technology*, 108, pp. 65–77.

[25] Shahin, M., Ali Babar, M. and Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, 5, pp. 3909–3943.

[26] Sharma, T., Fragkoulis, M. and Spinellis, D. (2016). Does Your Configuration Code Smell? In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pp. 189–200, iSSN: null.

[27] Tsvilik, S. (2020). wdio-docker-service.

[28] Vasilescu, B., Yu, Y., Wang, H., Devanbu, P. and Filkov, V. (2015). Quality and productivity outcomes relating to continuous integration in GitHub. Bergamo, Italy: Association for Computing Machinery, ESEC/FSE 2015, pp. 805–816.

[29] webdriverio (2020). webdriverio.

[30] Wrecker and Oracle (2018). Wrecker ci development blog.