

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 5500

4 April 8, 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 run 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 [22] work to see if their has been a 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 started to or have been improving their CI products. In terms of research, Infrastructure as Code in Rahman, Mahdavi-Hezaveh and Williams [23] which does a systematic mapping of research in that area. For Continuous Integration with Shahin, Ali Babar and Zhu [24] 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 [13] which demonstrate some of the depth that the papers go into.

Continuous Integration is a process of automatically running compiling, running tests and checking that the product works. This is 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 [22]. They looked closely at 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?

However the two key differences is that we will be scraping a new data set for the comparison. In doing so gathering slightly more data on the

repositories but not none on pull requests. As well as we didn't conduct a survey. From that additional data we are going to look more closely at the first question of What percentage of open-source projects use CI? As we are asking the same questions, we will use their corpus to compare on what has changed over the last 4 years. For our second theme, structure of CI as configuration we wanted to pick structural components that would be similar between all CI files. It would have been really interesting to do a full in depth analysis of each like Gallaba and McIntosh [15]. However we would like to tie in how the files are structured to how they are used so won't following that style. This led to the following research questions:

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

2 Related Works

2.1 Continuous Integration

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

An early definition of CI was written up and then updated later by Martin Fowler [14]. A key part of the CI is that allows teams to work on the same code base which without CI could easily lead to integration bugs and broken builds.

73 To enable to this to happen automation needs to put in place for build,
74 testing and other aspects of the integration process in order that a clear
75 piece of feedback (yes or no) can be given about the status of the build. If
76 done with from a version control system if the same commit is built twice
77 (so no changes have happened) it is vital that it produces the same result.
78 Otherwise it is hard for a team to be able to depend on CI if they are getting
79 flakey test results or flakey build results.

80 **2.2 Usage of Continuous Integration**

81 The actual usage of CI as configuration was looked at by [22]. In this they
82 use three source of information Github repositories, Travis builds and a sur-
83 vey. In order to be do a more systematic study of CI usage than [27]. In
84 analysing that data they found that "The trends that we discovered point
85 to an expected growth of CI. In the future, CI will have an even greater
86 influence than it has today." As we are looking at the same question we will
87 use four of the research questions out of the fourteen. In order to see what
88 difference four years has made to the growth of usage of CI.

89 **2.3 Config 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 research
92 in infrastructure as code Rahman, Mahdavi-Hezaveh and Williams [23]. The
93 has been a focus on Puppet and Chef, for example in Sharma, Fragkoulis and
94 Spinellis [25] looks at code quality by the measure of "code smell" of Puppet
95 code. This tackles the problem by defining by best practices and analyzing
96 the code against that. In the case of Cito et al. [12] it uses the docker linter
97 in order to be able to analyse the files. For the CI systems we pick we will
98 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 121 mapping between the api and user interface.
105 However using Ghtorrent GhT [6] 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 config files we
110 would look for. If it was a directory then it would get all ".yaml" or ".yml"
111 along with any Teamcity ".kts" and ".xml" files. However the script did not
112 look into any of the sub directories which might be the cause for the low
113 number of Teamcity configuration files found. In the case that it was a file
114 that was on the top level directory we matched it the lowercase file name we
115 found against the query.

116 In terms 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 list as they are new potentially popular sys-
119 tems.

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

















Branch: master New pull request	
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 [20]. (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.

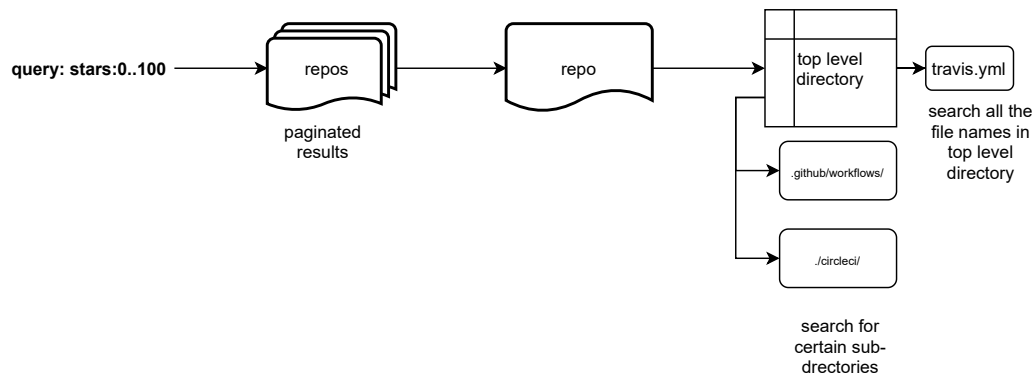


Figure 3: Diagram of the process used to search for projects with CI files in them

though was 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 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 save that data when we scrape a project.

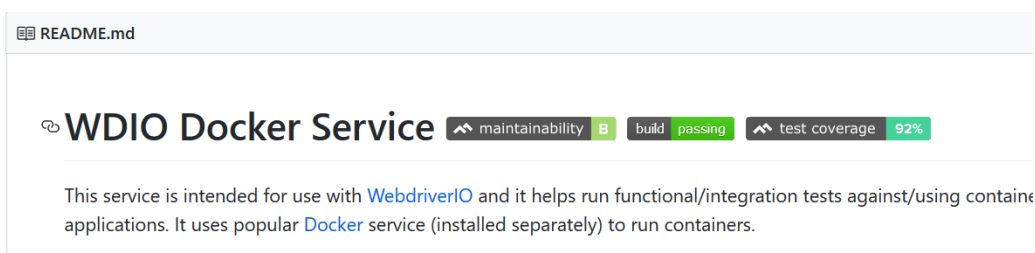


Figure 4: Example of CI tag for Github ReadMe [26]

We ended up with a config file with queries for configuration files 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 a
 145 search on Github for what we think based on the docs [29] and [18] for their
 146 config file naming convention. We were only able to find 20 results so did
 147 not include in the scraping script to speed up the process of searching for the
 148 other configuration file formats.

149 Along with information of what CI is being used for a project we also
 150 gathered metadata about the project. The available metadata through the
 151 api is largely what can be seen on a repository for example in Figure 5.
 152 We have the star count which is an indication how popular a project is as
 153 users can star projects that they like Borges, Hora and Valente [11]. Then
 154 we have watchers which is users that have subscribed to the project to get
 155 notifications about the project.

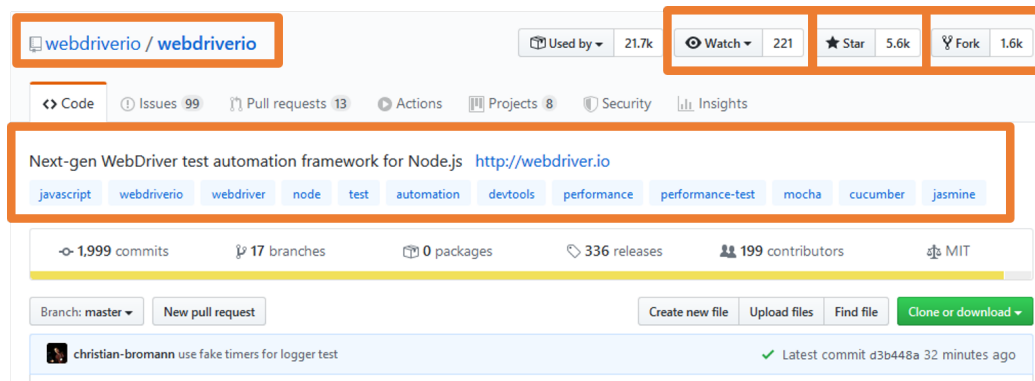


Figure 5: Example Github project description and metadata[28]. The orange sections sections highlighted are the metadata that we scraped.

156 3.1 Data corpus

157 This all produced a sample of 32,660 projects from open source projects on
 158 Github. As can be seen in Figure 6 we weren't able to scrape the whole
 159 star count range easily. This is because the script would crash when Github
 160 gave a 500 error code at us randomly. Along with empty repositories initially

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

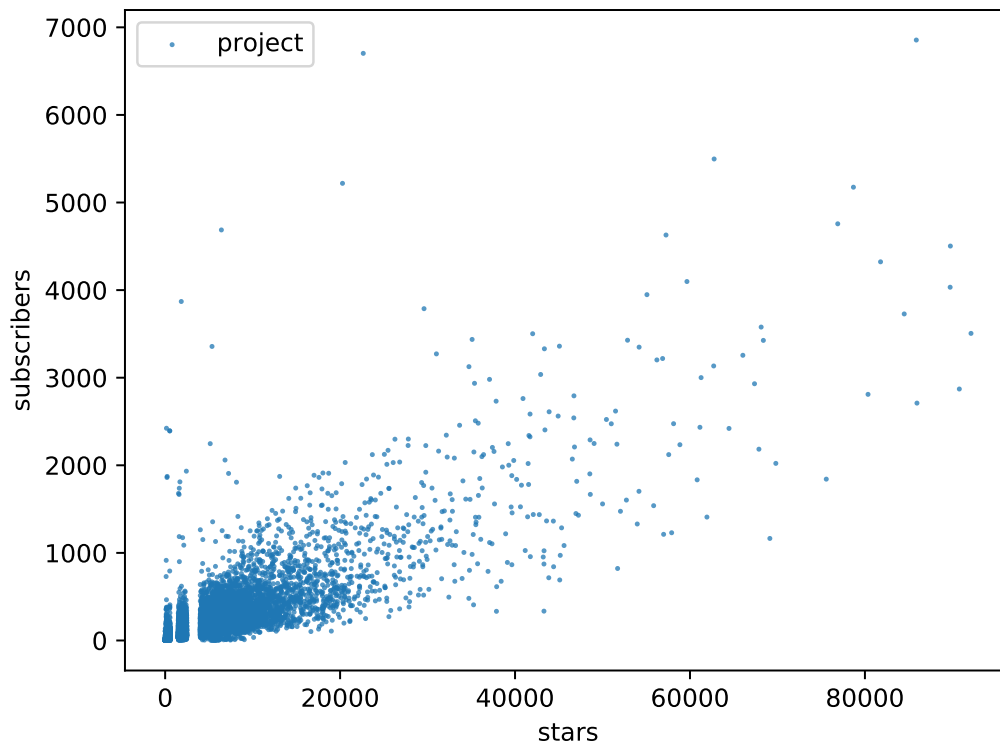


Figure 6: Github stars against subscribers

166 3.2 Comparison corpus info

167 In Michael Hilton, Marinov and Dig [22] paper they use a similar method
 168 of using the Github Api in order to create their corpus. Additionally they
 169 contacted Cloud Bees Clo [4] to get a list of all open source projects that
 170 used their services. This helped them not to miss out on projects that they
 171 would otherwise missed out on. They kindly gave a copy of their final corpus.
 172 However it does contain the data on the Cloud Bees projects which is

223 projects. As far as we can tell this only effects the comparison done in RQ2 4.2. Additionally we found slight discrepancies between the paper and corpus in RQ1, RQ2 and RQ3 mainly just a few numbers off in a few places. In order to do comparisons well and to keep it consistent we will be basing all our comparisons from the corpus. As the discrepancies are small we will using the conclusions from the paper where possible.

In order to get a better understanding of the results of the methodologies chosen in both cases. We created Figure 9 two histograms to showing the density by the stars of the spread of data using the Sturge's rule. As we expected both corpses are skewed to the left.

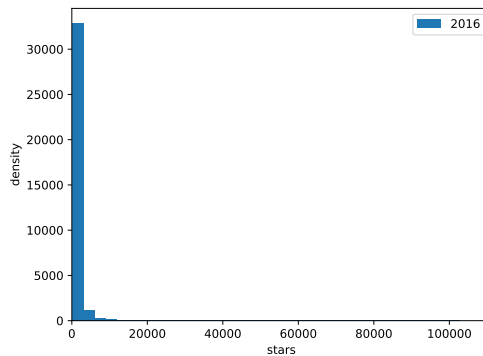


Figure 7: 2016 corpus

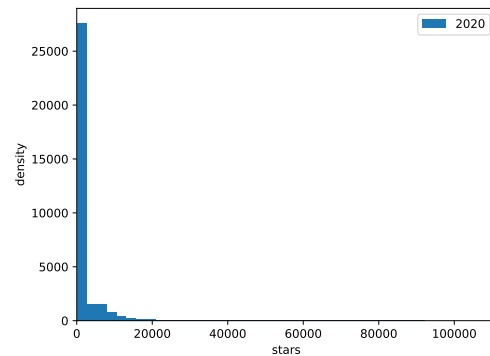


Figure 8: 2020 corpus

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
Found CI	13752	39.81%	12538	38.39%
No CI found	20792	60.19%	19214	58.38%
ReadMe has CI status	n/a		908	2.78%
Total	34,544		31,494	
Multiple CI	1796	12.91%	1764	14.07%

Table 1: This table shows the comparison between the 2016 dataset which is Michael Hilton, Marinov and Dig [22] 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 CD and some projects are run a monorepo which means that 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 (55.92%) projects with a ReadMe that had a CI status label that we could

201 find. However this method is not perfect, for example “awesome-bootstrap-
202 checkbox” by “flatlogic” [5] there ReadMe has the following line:

```
203     [![Dependency Status]  
204     (https://img.shields.io/david/dev/flatlogic/awesome-bootstrap-checkbox.svg?branch=master&style=flat)  
205     ]  
206     (https://www.npmjs.com/package/awesome-bootstrap-checkbox)
```

207 This contains **Status** and a url so we say it has got CI when the repository
208 currently doesn’t. Yet this is not the case for all of them as for example
209 “SyncTrayzor” by “canton7” [21] uses AppVeyor but doesn’t use a configu-
210 ration file for it. Therefore we didn’t find it as we searched for a config file
211 only.

212 The percentage of CI projects they had was 39.81%. If you look at Table
213 1 it shows that we got 38.51% CI projects. This is interesting as we searched
214 for more kinds of CI configuration so there was a potentially a higher chance
215 of having CI.

216 One possible reason could be because of in RQ3 4.3 it shows that the
217 more popular a project the higher chance it has of using CI. Therefore as
218 their sample contains a few more projects that are popular their they could
219 all be using CI. However that is a weak tangent to make in order to full
220 explain it.

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

225 Another possible reason is that because of Github’s growth over the last
226 4 years (Git [7] to 2019 Github [17]) so that Github is now at 40 million
227 active users. It means that there are more projects that are using CD setups
228 for building their static sites and in general Github is being used for more
229 things that wouldn’t require CI.

230 Therefore we think that the last two factors are the most likely contribu-
231 tors to why there is less CI usage now. Another important interesting part is
232 despite Github growing so much the CI usage rate has stayed relatively the

233 same.

234 4.2 RQ2: What CI systems are projects using?

235 In Table 2 we find like all other research Travis is the most popular CI system
236 in use. However over the last 4 years since the [16] CircleCi has lost out on it's
237 rough quarter that it owned. In particular the rise of Github Actions seems
238 to have taken second place even though it is still very young in comparison
239 as it was officially released November 13th 2019 but had a closed beta since
240 the summer of 2019. However this might not be down to the CircleCi loosing
241 out on their existing share. But potentially as the rise in CI usage goes up on
Github. Projects are more likely to pick in the built in solutions to Github.

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%

242

243 Our sample of repositories is 31,494 this means that as it is a representa-
244 tion of projects on Github so won't account for the whole of it. This means
245 that although Wrecker had the smallest count of CI when researching of 20
246 projects. In Table 2 we have configuration types that have lower counts.
247 This is because that search for the 20 searched the whole of Github but the
248 scraping was only able to do a small sample. Additionally their potentially
249 could be faults in the scraping causing it show such low numbers for the last
250 3.

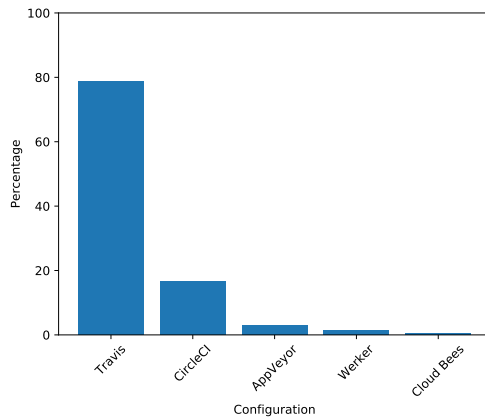


Figure 10: 2016 corpus

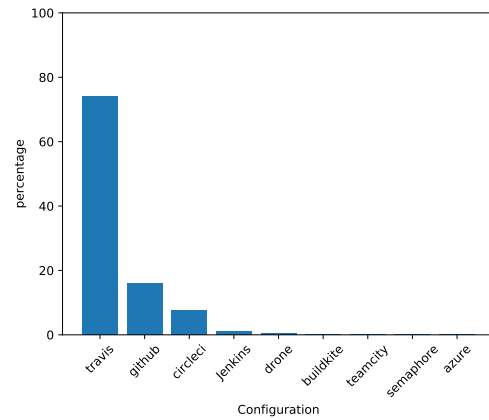


Figure 11: 2020 corpus

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

251 4.3 RQ3: Do certain types of projects use CI more 252 than others?

253 Below shows all the CI projects sorted then grouped together per 540 projects.
254 Then in this case we choose to categories via star count for each project.

255 Here in Figure 14 and 13 we are comparing whether or not in the last 4
256 years the number of stars increases the CI being used. It shows how the trend
257 in the more popular the project by how you have more stars for a project
258 increases the chances it uses CI has stayed the same. However the gradient
259 of that trend has changed to be slightly greater overall. Yet not quite as
260 sharp for the end of the graph this is most likely because the 2016 dataset
261 doesn't have as much data between 40000 and 90000 as seen in Figure 7.

262 Figure 16 uses the same method as Figure 15 except is does it based the
263 number of subscribers. Subscribers are used on Github to keep update on the
264 changes on the project. This could range from core team members working
265 on the project to people that want to be notified about a new release. In
266 looking at this metric the hypothesis was that it would have a sharper rise

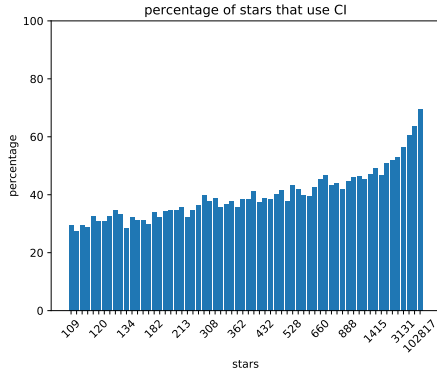


Figure 13: 2016 dataset

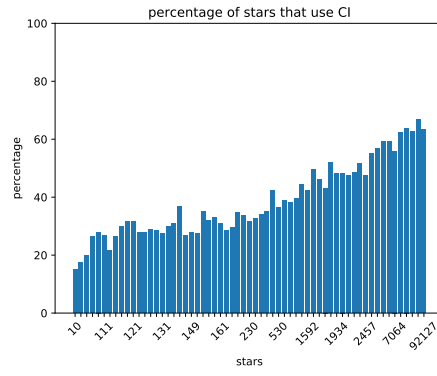


Figure 14: 2020 dataset

Figure 15: In Figure 14 is the results from this research and in Figure 13 is the results from [22]. The results show the percentage chance of CI usage depending on the number of stars a group of 540 projects has on average.

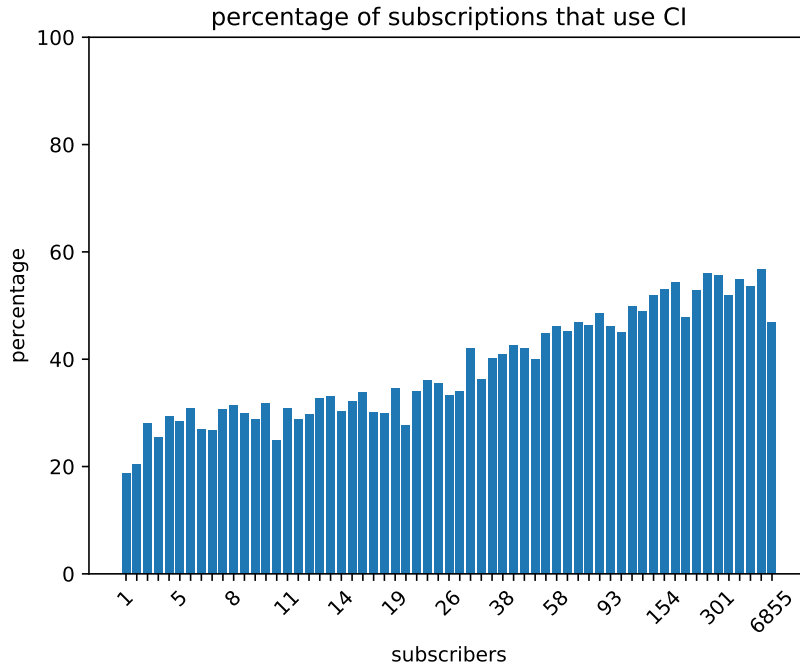


Figure 16: Subs graph

267 in percentage of projects using CI per subscriber. However that was not the
 268 case overall the gradient is not as strong. There is no comparison to [22]

269 because their final corpus does not contain subscriber count for each project.

270 That gives us a good look at how projects can be viewed through Github’s
271 metadata.

272 In terms of what kind of programming languages are being used for CI? As
273 well as what programming languages were found when creating the sam-
274 ple. We can see the top 20 results in Figure 17 in that we can see that
275 Javascript is the most common kind of project. This was too be expected as
276 in Github’s annual report [17] on the platform they reported that Javascript
277 has been the most popular for the last 5 years. The interesting part is that
278 our sample matches the rise in Python over Java. Despite the fact that they
279 are using “unique contributors to public and private repositories tagged with
280 the appropriate primary language” and we are using the count of projects by
281 primary programming language tag.

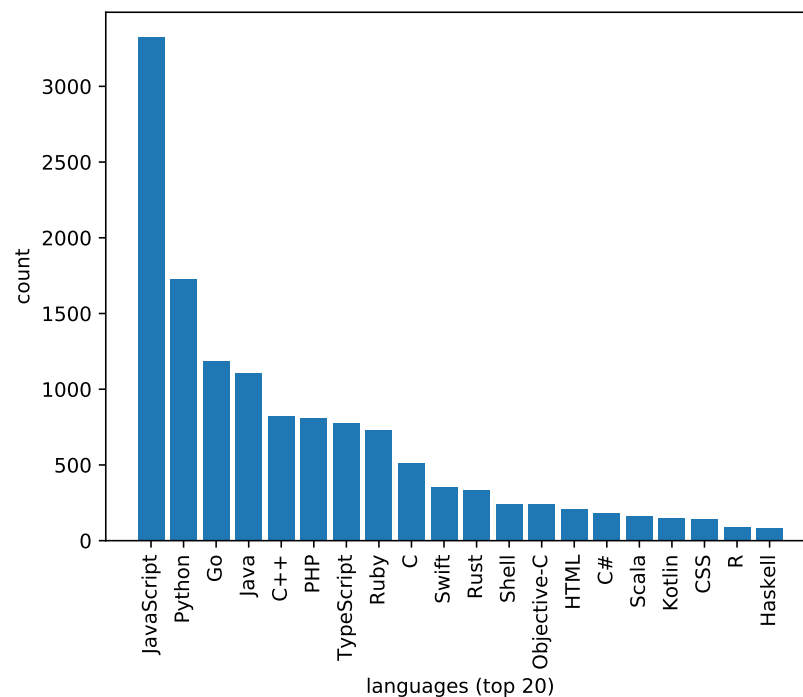


Figure 17: Count of top 20 programming languages used by projects using CI

282 In order to get a better idea of the breakdown of the effect programming
 283 languages have on CI usages. We created Figure 18 this shows three pieces
 284 of information the percentage of CI usage on the y axis, average star count
 285 on the x axis and then number of projects using the language by the size of
 286 the dot.

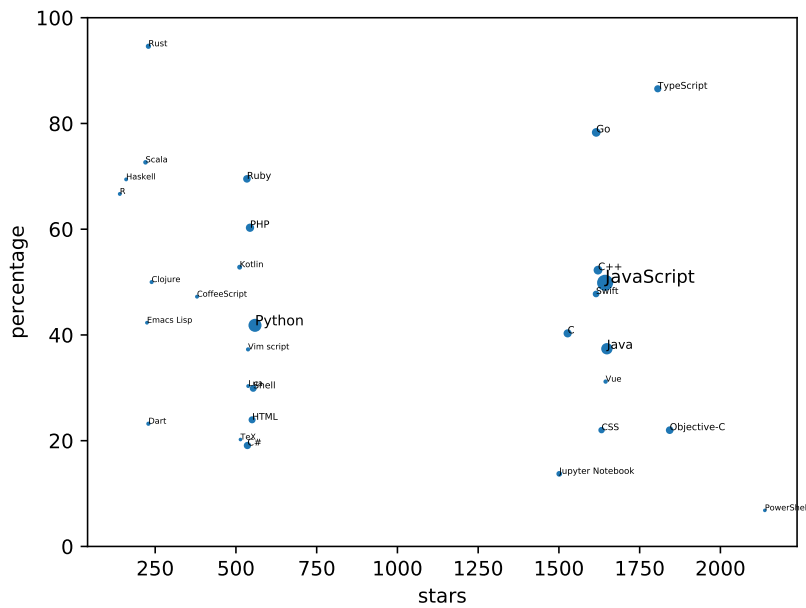


Figure 18: Scatter graph showing the top 30 most used programming languages against how much they use CI. The key points are Rust, Typescript and Go being the top three programming in CI usage.

287 The most striking part of Figure 18 is the clear divide between different
 288 programming languages star count. The programming with the languages with
 289 the highest CI usage are Rust (94.6%), Typescript (86.56%) and Go (78.31%).
 290 This is interesting in how they are all fairly “new programming languages”
 291 in comparison to the others in the graph. They are all languages which are
 292 developed and open source on Github. In terms of Rust and Go it could be
 293 down to their tooling that comes builtin to the language. As that would lead
 294 to implementing CI to be a lot easier. Yet Typescript is more a special case as
 295 it is a subset of Javascript so uses ‘npm’ to deal with dependency management

296 which was some of inspiration for Rust’s tooling Rus [10]. Older programming
 297 languages like Java and C# both have tooling for dependency management
 298 but the chances that they use CI is much lower. Therefore an area for further
 299 research would be whether or not the use “modern” dependency management
 300 systems increases the chance of CI.

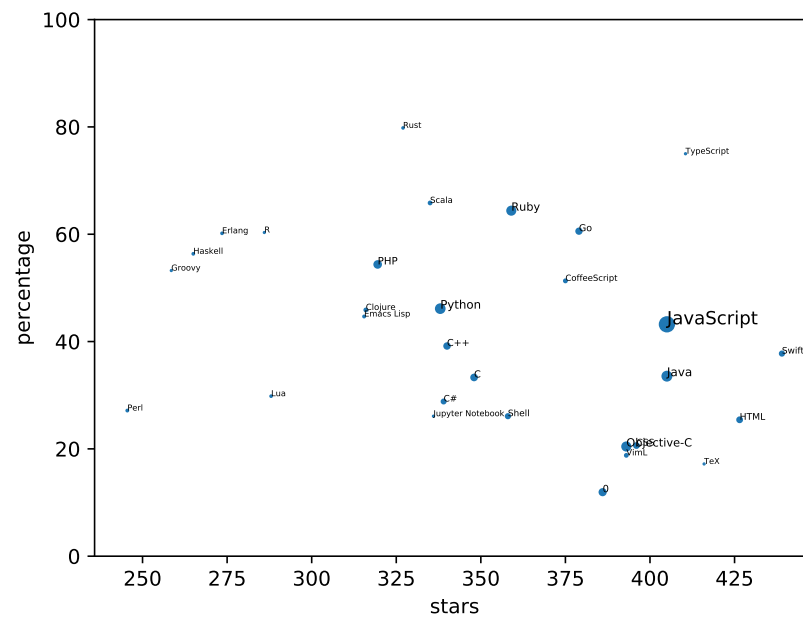


Figure 19: Scatter graph showing the top 30 most used programming lan-
 guages against how much they use CI for Michael Hilton, Marinov and Dig
 [22] from 2016. In comparison to Figure 18 the data is not as grouped as
 clearly by the star count. Rust, Typescript and then Scala have the highest
 programming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

301 In figure 19 it shows the sample from 2016 in comparison to 18. The first
 302 major difference is in how the spread languages by stars isn’t as divided. This
 303 could be because of how the sample is not as spread out as seen in Figure
 304 9. The scatter graph is for the top 30 most common programming languages
 305 found from this Rust, Typescript and Scalar have the highest chance of using
 306 CI. This is really interesting is that Rust and Typescript are still really within
 307 the top 3 after 4 years. Potentially this could be to do with the ecosystem

308 around the languages that lead this. However this an area for further research
309 of why different languages have a higher chance of using CI.

310 Finally one observation that was made in Michael Hilton, Marinov and
311 Dig [22] paper was that there was a higher chance of CI usage for dynami-
312 cally typed languages. We looked into analysing this as we found both Rust
313 and Typescript having really high CI usage chance. Yet at the same time
314 overall Javascript and Python had the most projects that used CI. So we
315 wanted to look at where the balance lied in the difference between the two.
316 However categorising the programming languages by their usage is difficult.
317 For example is a Javascript a project that is using Typescript's js checking
318 dynamically typed or statically typed? And then how do you tell? Or if
319 you have a similar situation where Python has static types. Therefore this
320 an area for further research as it is a question that would need to carefully
321 answered.

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

326 5 Structure of configuration files

327 The following three research questions will just be on the 14,302 CI projects
328 (found CI plus multiple CI 4.1). In order to be able to ask the questions
329 about the data we filter the sample to only include CI projects. Then we
330 created a csv table with a row per CI type in that project as some projects
331 had multiple versions of CI as shown in 4.1. Then we processed each CI file
332 to get the necessary data to be able to ask questions about it's structure. As
333 we wanted to be able to process files with or without errors in along with all
334 types of CI. We created a parser to go through each line of the configuration
335 file working out what that line is. For example is it a comment or blank line
336 or does it have code.

337 5.1 RQ4: What are the common errors when loading 338 yaml configuration?

Here our the

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 [9] 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: ]
```

339

Composer error In the example it has two steps that are using an 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 an 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
```

Table 3: yaml configuration errors

config	composer error	constructor error	parse error	scanner error	no. config
circleci	1	0	0	1	1109
drone	31	0	0	0	84
github	0	1	0	3	2301
travis	6	0	10	21	10607
buildkite	0	0	0	0	32
semaphore	0	0	0	0	2
azure	0	0	0	0	1

As can be seen in the Table 3 their our configuration files with yaml errors meaning that the CI for that project will not load correctly. Yet it seems that a very small percentage of projects that have them. For example the two highest configuration types with errors are Drone (36.90%) followed by Travis (0.348%).

In the case for Drone all the errors are for the same type of error. Potentially this could be because of how anchors are a lot more common in Drone.

For Travis as it is the largest config type out of the sample by a significant amount it is more likely to contain more errors. Yet with such a small amount it seems like yaml errors aren't a major problem in CI. Although as they are required to be fixed in order for the CI to run the chances are the ones

352 with errors ones that are being changed when the scraping was being done.
353 Meaning that as the CI has been set up correctly for the other 99.632% as
354 they are not needing to change because their our no yaml errors in it and
355 presumably it is doing what they intend for it to do.

5.2 RQ5: How are comments used in configuration?

356 The assumption was the as continuous integration setups can be compli-
357 cated and have edge cases. Therefore comments in the configuration would
358 be used to describe and handle that complexity.

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

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 right there is an example Github Action yaml file. If were it would be parsed we would get:

```
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: example configuration file

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

367 In Figure 22 is just all the yaml based configurations. In order to get

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

368 a better understanding of comment usage for them. As comments average
369 line numbers is so it wasn't visible in Figure 21. The other really interesting
370 finding was that average line count for Travis CI files was the smallest. This
371 is because Travis is the most popular CI service so there would be a higher
372 chance to have large configuration files.

373 In Figure 22 it shows how the comments are broken down for each CI
374 service on average. In the case of Azure our sample size is of only one
375 project so that doesn't give us any significant insight. The blue line which
376 represents code that has a comment after is the most commonly used kind
377 of comment. Apart from for Drone which has more multiple line comments.
378 This is really interesting as it highlights that comments tend tied to the code
379 that is written. We had expected for multiple line or single comments to
380 be the most common kind of comment used. This is because code tend to
381 follow a style guide to limit the maximum characters for a single line PEP
382 [8]. In doing so this rule also applies for comments as well therefore you don't
383 normally have space for a code and a comment. (BUT THAT IS NOT THE
384 CASE HERE THINKING FACE..)

385 In the case for Jenkins pipelines and Teamcity there is a much higher
386 usage of having code with comments. Therefore we have separated the anal-
387 ysis and comparison from Figure 21 to Figure 24 for the none yaml based

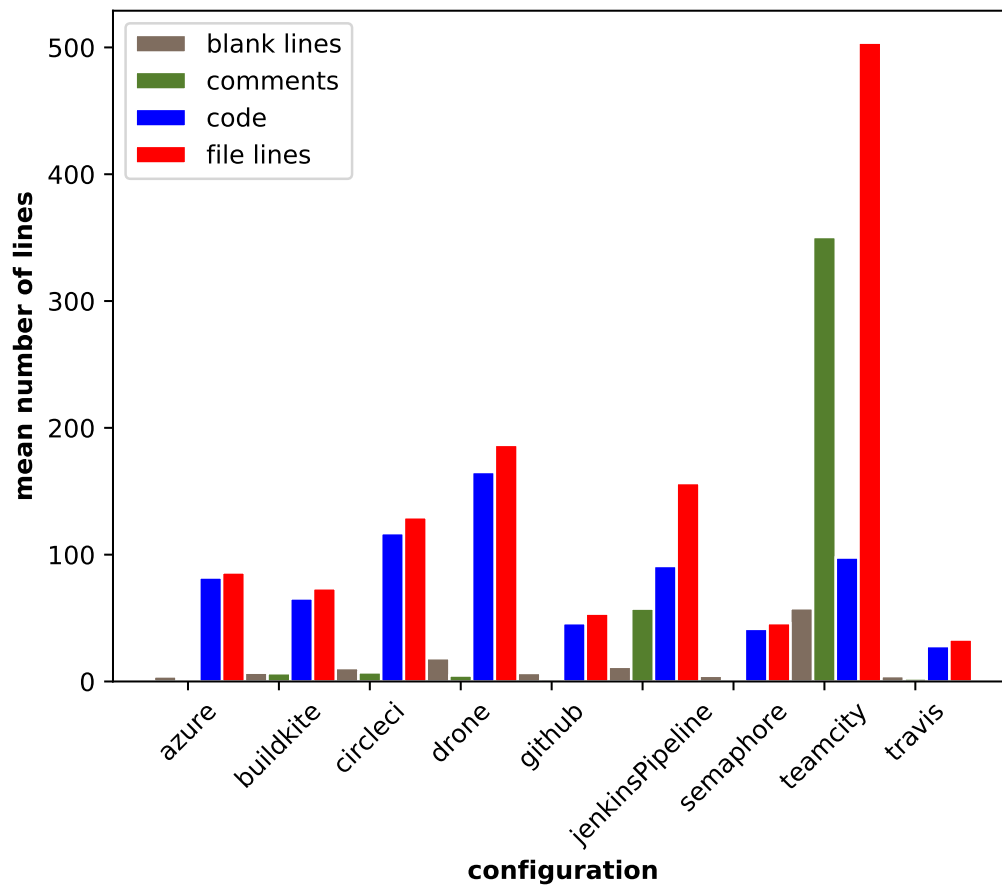


Figure 21: Mean of line counts for all CI configuration types. Showing the blank, coded and commented line average breakdown.

configuration. Jenkins and Teamcity configuration is Kotlins based and for TeamCity also xml based. The key difference we find here is that multiple line comments is more common than code with comments when compared to the yaml configuration. The second key difference is a much higher mean number of lines for comments. These two difference are probably combined because both Kotlins and xml allow block comments which allow for multiline comments to be done easily.

We have looked at the structure of the file and the what kind of comments are used. Then we looked at the structure of the comments. In to do this

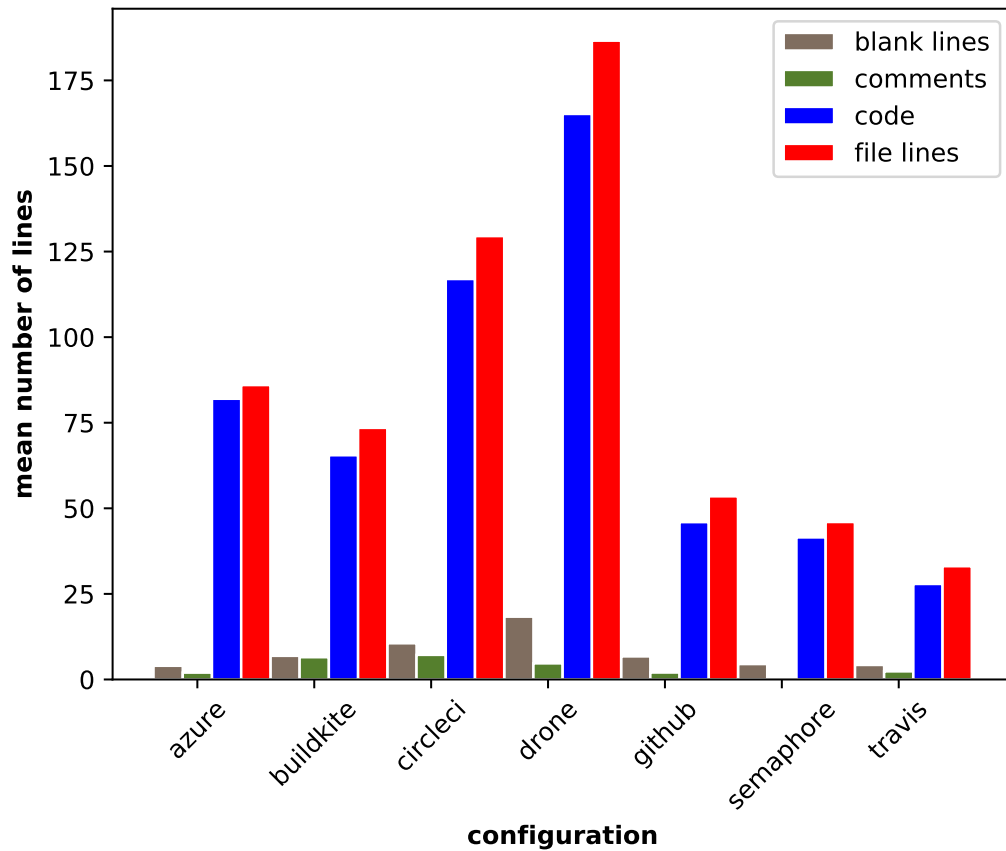


Figure 22: Mean of line counts but only for yaml based configuration. Giving a clearer view on the breakdown of how little blank lines and comments are used.

we created a list of regular expressions used to categorise how the comments were structured.

From labelling the comments in Figure 25 we can see that having comments with versions in and urls is most common. This could indicate comments from templates or how they are commented. Although yet again the amount of labels found on average is still very low.

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

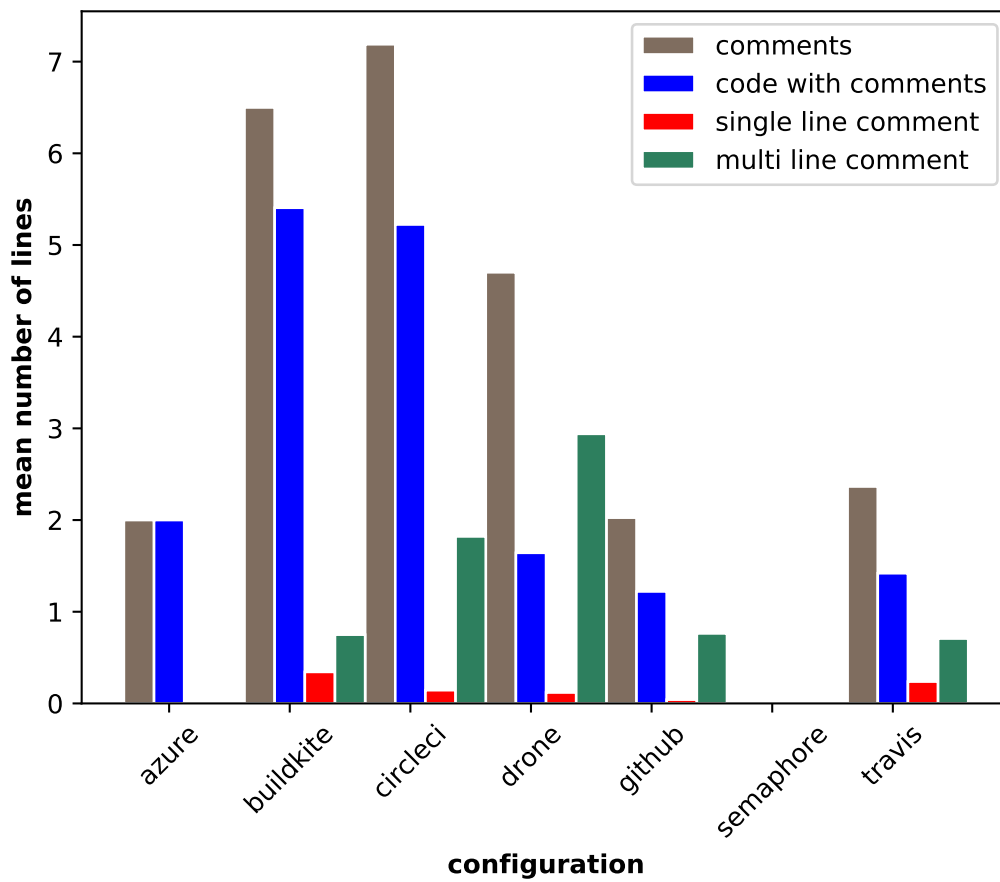


Figure 23: Mean of line counts for comments, code with comments, single line comments and multiple line comments for yaml configuration files

406 or commenting out configuration.

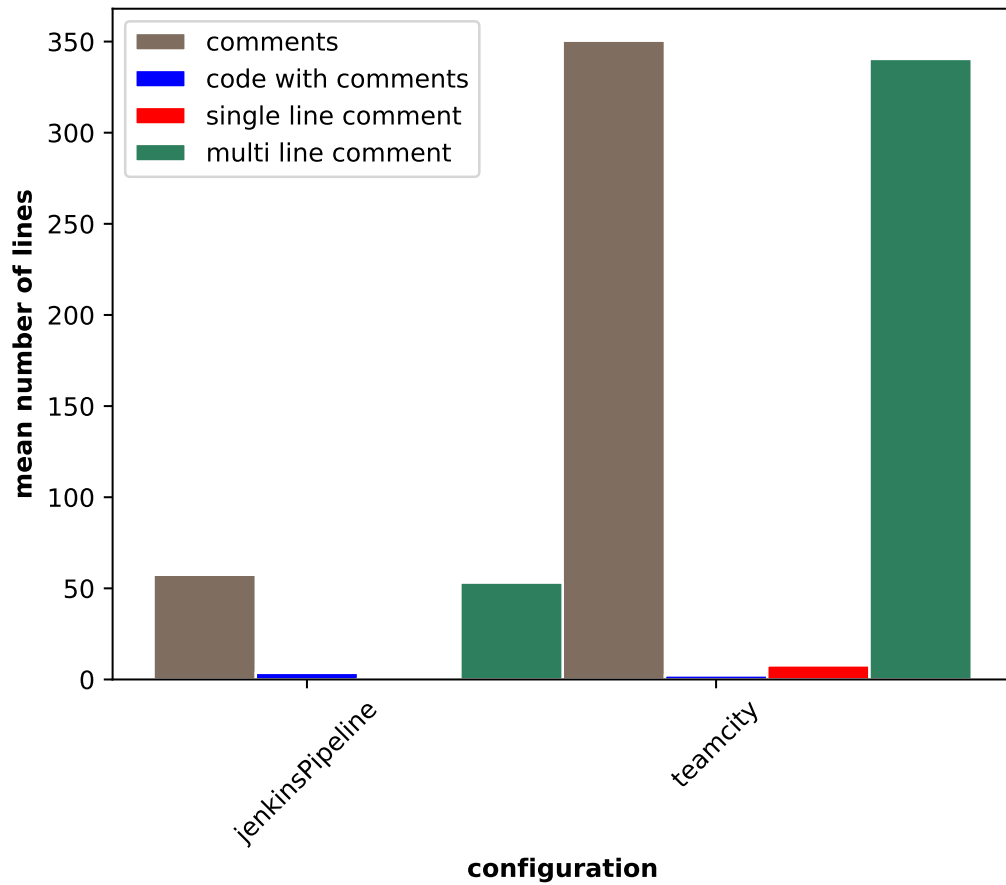


Figure 24: Mean of line counts for comments, code with comments, single line comments and multiple line comments for none yaml configuration files

5.3 RQ6: Are external scripts used within the configuration?

An external script is a bash or powershell script typically depending on the operating system. It can be used to build, deploy or do any step that CI takes. The key difference between it and the CI configuration is that it be executed on a users machine. Therefore you do get some setups where you 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. `//===`).

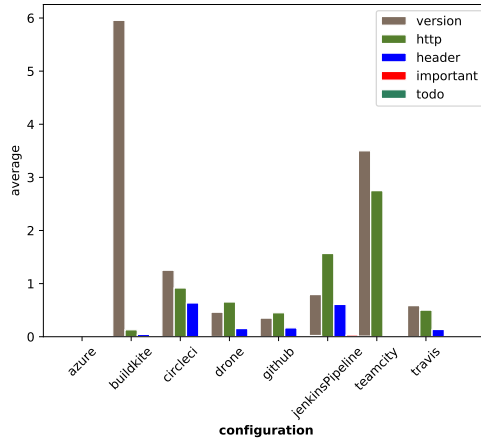


Figure 25: Comment types

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

In Figure 26 it 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 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 so 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 our interesting results and in order to

431 be able know the reasoning for the high numbers further research needs to
432 be done.

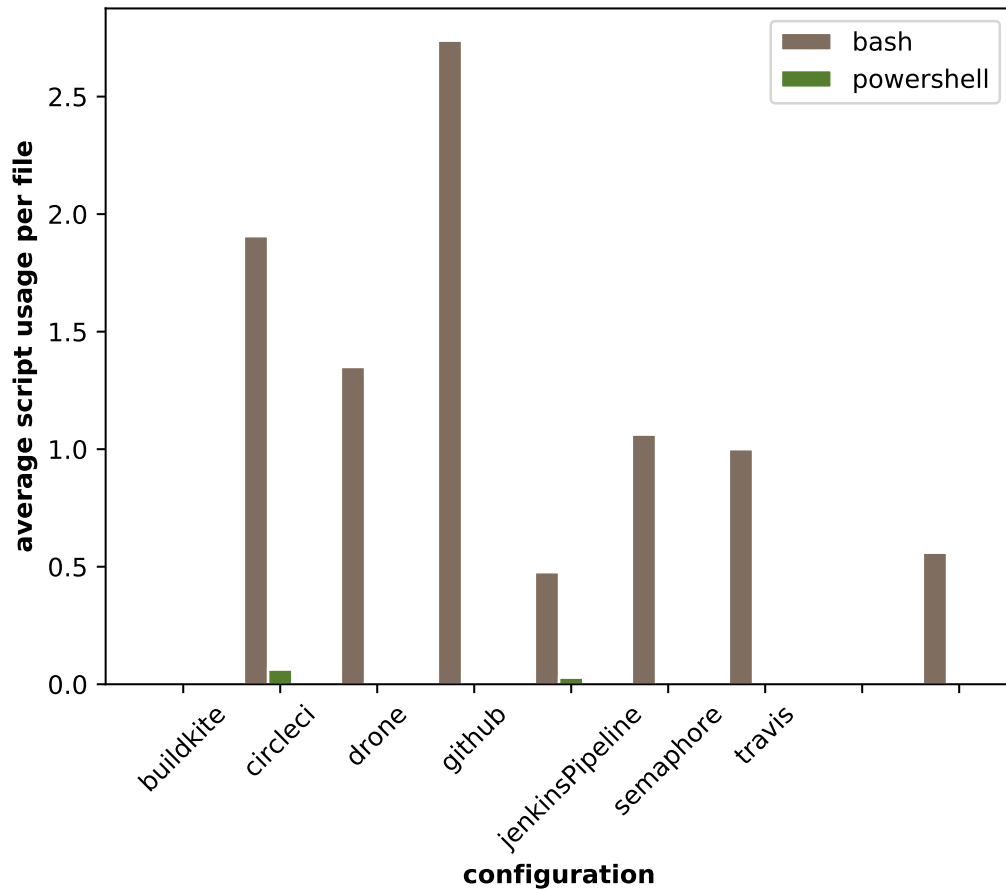


Figure 26: Mean script usage for each file

433 In Table 5 we can see the raw count found for each CI service for bash
434 and powershell. We were surprised to see the percentages for external scripts
435 to be so high. As we suspected that internal script usage was most likely
436 very high but

437 As some of the necessary actions are being done in the scripts and not in
438 the CI file. Potentially there could be less lines of code in the configuration
439 for files that use scripts. However in Figure 27 we can see that the data is
440 very spiky with outliers. Then in Figure 28 we can see the same affect when

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

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

trying to see if the more popular a project is affects the chances of it using CI.

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

In Figure 28 shows that there is no correlation between popularity and external script usage. The graph more closely follows the same shape as Figure 6. However much like that other graph there is no clear correlation to be made.

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

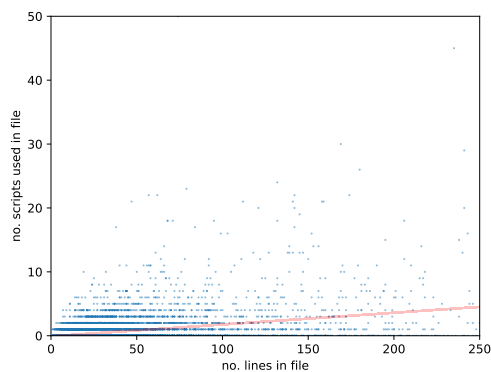


Figure 27: no. scripts to no. lines with the extreme values cropped out. This shows a slight trend in the more lines you have the more scripts you will use.

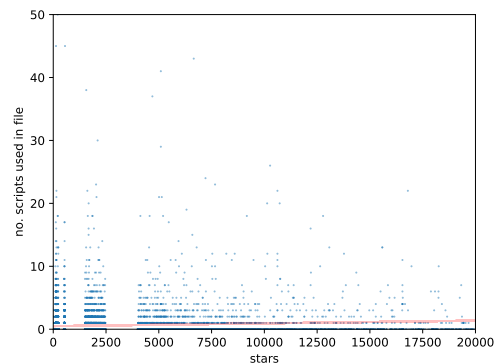


Figure 28: no. scripts to stars with the extreme values cropped out. As we are comparing stars along the x axis it looks similar too Figure 6. But focused on the first 20,000 stars as that is where the data is most dense. We found a slight increase in usage of scripts the more popular a project became. Yet this is only a correlation further research would need to be done to work out if their is a causation

454 6 Threats to validating

455 The major and most obvious threat is the sample gathered from scraping the
 456 data from Github. This has already been touched on in the 3 section but
 457 now we are going to look at it in more detail.

458 Firstly if we assume that the scraping works perfectly then it's only at
 459 maximum a 1000 open source projects per star. That is excluding closed
 460 source projects which would range from personal projects to companies. As
 461 well as it is only data from Github not from Gitlab, bitbucket or other version
 462 control hosting services. This leads to bais in the data for example if Gitlab
 463 was also scraped then we would get a lot more Gitlab ci files. However in

464 order to get best spread of data Github has the best api and most services
465 do not tie you down to use only their service. As well although we could get
466 a 1000 projects per star we were still able to get around 30,000 projects and
467 a wide spread across Github. The key aspect being that because it was a
468 sample we focused on getting a good spread of data.

469 Secondly the scraping script is not perfect in how it finds configuration
470 files. As it only looks in the top level directory for the file name pattern
471 described in their docs or unique folder. Therefore if the systems allowed
472 many different names or different names in past it wouldn't have picked it
473 CI system. Additionally we only decided to scrape for certain CI files. Yet
474 we chose a good scope based on previous research into the top CI files. As
475 well the scraping script has been tested worked on to try and minimise any
476 bugs. In the case that we did not pick up a CI file we ran a regexp against
477 the ReadMe file to get a better understanding of the error bounds.

478 Thirdly identifying which projects are programming projects or would
479 have a need for CI. Based on the research [19] it is important to filter out
480 repositories that aren't part of the question being asked. Therefore we could
481 have looked to try and filter out Github static sites and other none software
482 based projects. However if assume a certain type of project won't be using
483 CI then we would be introducing bias when trying to answer how CI is used.
484 For further research better labelling of what kind of projects are which would
485 potentially beneficial though.

486 7 Summary

487 We got a sample of 32,660 open source projects from Github and were able
488 to compare that to a previous study 4 years ago. In doing so we found that
489 usage of CI projects was similar and that more popular a project the higher
490 chance it would be using CI. This linked with the research from 4 years ago.
491 The major change was the increase in popularity of Github Actions taking
492 over second place from CircleCi. Additionally we look at whether or not the
493 number of people watching the project had the same effect it did but to a
494 lesser extent.

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

499 From this a better understanding of this topic could be gathered by look-
500 ing into the data gathered more. As we found we were faced with a lot more
501 questions while doing this research as we go into below.

502 7.1 Discussion and further research

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

506 Further research into usage that we would like to do is look into how
507 the size of the project affects the chance that it uses CI. Then looking at
508 the usage of scripts within CI configuration, for example using a script tag
509 to run a shell script. As while doing the research we found some projects
510 use scripts a lot while others just used the CI config. This would lead to
511 questions around which CI system has a higher amount of scripts used. But
512 also looking at how much they enable them to be used and what is the size
513 of those scripts. The data for the programming language and version(s) is in
514 the config. Therefore it would be possible to work out how much usage each
515 version is getting of a particular programming language.

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 an 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. [1], [2], [3]) is used more for projects with CI and those that don't.

In addition working on pruning our dataset using methods outlined in [19].

8 Acknowledgement

We wish to thank Michael Hilton in particular for providing the corpus for their research Michael Hilton, Marinov and Dig [22].

References

- [1] (???).
- [2] (???).
- [3] (???).
- [4] (???). Cloudbees website.
- [5] (???). flatlogic/awesome-bootstrap-checkbox.
- [6] (???). Ghtorrent website.
- [7] (???). GitHub State of the Octoverse: 2016.
- [8] (???). PEP 8 – Style Guide for Python Code.

- 540 [9] (????). Yaml faq.
- 541 [10] (2020). Cargo: Rust’s community crate host | Rust Blog.
- 542 [11] Borges, H., Hora, A. and Valente, M. T. (2016). Understanding the
543 Factors That Impact the Popularity of GitHub Repositories. In *2016*
544 *IEEE International Conference on Software Maintenance and Evolution*
545 *(ICSME)*, pp. 334–344, iSSN: null.
- 546 [12] Cito, J., Schermann, G., Wittern, J. E., Leitner, P., Zumberi, S. and
547 Gall, H. C. (2017). An Empirical Analysis of the Docker Container
548 Ecosystem on GitHub. In *2017 IEEE/ACM 14th International Confer-*
549 *ence on Mining Software Repositories (MSR)*, pp. 323–333, iSSN: null.
- 550 [13] Copeland, P. (2010). Google’s Innovation Factory: Testing, Culture, and
551 Infrastructure. In *Proceedings of the 2010 Third International Confer-*
552 *ence on Software Testing, Verification and Validation*, Washington, DC,
553 USA: IEEE Computer Society, ICST ’10, pp. 11–14.
- 554 [14] Fowler, M. (2010). Continuous integration.
- 555 [15] Gallaba, K. and McIntosh, S. (2018). Use and Misuse of Continuous In-
556 tegration Features: An Empirical Study of Projects that (mis)use Travis
557 CI. *IEEE Transactions on Software Engineering*, pp. 1–1.
- 558 [16] Github (2017). Github welcomes all ci tools. In github.com, ed., *Github*
559 *welcomes all ci tools*.
- 560 [17] Github (2019). Octoverse - top languages.
- 561 [18] GitHub (2020). github filename search for wrecker.yml files.
- 562 [19] Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M.
563 and Damian, D. (2014). The promises and perils of mining GitHub.
564 Hyderabad, India: Association for Computing Machinery, MSR 2014,
565 pp. 92–101.

- 566 [20] Ling, J. (2019). Cu worhsip song list creator - a repository taken over
567 for testing.
- 568 [21] Male, A. (2020). canton7/SyncTrayzor. Original-date: 2015-02-
569 08T17:08:40Z.
- 570 [22] Michael Hilton, K. H., Timothy Tunnell, Marinov, D. and Dig, D.
571 (2016). Usage, costs, and benefits of continuous integration in open-
572 source projects | Proceedings of the 31st IEEE/ACM International Con-
573 ference on Automated Software Engineering.
- 574 [23] Rahman, A., Mahdavi-Hezaveh, R. and Williams, L. (2019). A system-
575 atic mapping study of infrastructure as code research. *Information and*
576 *Software Technology*, 108, pp. 65–77.
- 577 [24] Shahin, M., Ali Babar, M. and Zhu, L. (2017). Continuous Integration,
578 Delivery and Deployment: A Systematic Review on Approaches, Tools,
579 Challenges and Practices. *IEEE Access*, 5, pp. 3909–3943.
- 580 [25] Sharma, T., Fragkoulis, M. and Spinellis, D. (2016). Does Your Config-
581 uration Code Smell? In *2016 IEEE/ACM 13th Working Conference on*
582 *Mining Software Repositories (MSR)*, pp. 189–200, iSSN: null.
- 583 [26] Tsvilik, S. (2020). wdio-docker-service.
- 584 [27] Vasilescu, B., Yu, Y., Wang, H., Devanbu, P. and Filkov, V. (2015).
585 Quality and productivity outcomes relating to continuous integration
586 in GitHub. Bergamo, Italy: Association for Computing Machinery,
587 ESEC/FSE 2015, pp. 805–816.
- 588 [28] webdriverio (2020). webdriverio.
- 589 [29] Wrecker and Oracle (2018). Wrecker ci development blog.