Usage and Structure of continuous integration as configuration?

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

Abstract—**Continuos integration (CI) is becoming more popular as software development moves to an Agile fast paced development life cycle. Most CI is done automatically using a service which runs based off configuration. Our major questions is how much is CI actually being used? As well as how are these files being structured? We got 94,379 open source projects from Github to answer these questions. In doing so compared our results against (author?) [1] work to see if their has been a increase in usage. We found a shift in CI services being used and were able to get similar results to their study. In terms of structure we found that configuration files are written with no comments normally. We suggest at the end further research is needed to get a better understanding of this growing field.**

I. INTRODUCTION

Continuos 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 **(author?)** [2] which does a systematic mapping of research in that area. For Continuous Integration with **(author?)** [3] 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 **(author?)** [4] which demonstrate some of the depth that the papers go into.

Continuos Integration is a process of automatically compiling, running tests and checking that the product works. This is can be combined with Continuos 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 [1]. 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?
- **RQ 3.5** Do more recently contributed projects use CI?

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

It would have been really interesting to do a full in depth analysis of each CI configuration format like **(author?)** [5] does for Travis. However we can look at the general structure of all the CI configuration files allowing for comparisons to be made between configuration files. As that will allow comparisons to be made more easily otherwise comparing very specific features would have been harder.

- **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?

II. RELATED WORKS

A. Continuos Integration

Continuos 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 [6]. 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.

To enable to this to happen automation needs to put in place for build, testing and other aspects of the integration process in order that a clear piece of feedback (yes or no) can be given about the status of the build. If done with a version control system, if the same commit is built twice (so no changes have happened) it is vital that it produces the same result. Otherwise

it is hard for a team to be able to depend on CI result if they are getting flakey test results or flakey build results.

B. Usage of Continuous Integration

The actual usage of CI was looked at by [1]. In this they use three source of information Github repositories, Travis builds and a survey. The impact of CI usage on Github was looked as well in **(author?)** [7]. The key difference is that we will be looking at the configuration alongside the CI usage and leaving impact of CI on the project to further research or current existing research. In analysing that data they found that "The trends that we discovered point to an expected growth of CI. In the future, CI will have an even greater influence than it has today." As we are looking at the same question we will use four of the research questions out of the fourteen (IV-A, IV-B, IV-C). In order to see what difference four years has made to the growth of usage of CI.

In addition to comparing to the previous study we will also look into how CI usage changed over the years.

C. configuration as code

Configuration as code or Infrastructure as Code has been an increasing area of research over the last few years. There seems to be slightly more research in infrastructure as code, for example see **(author?)** [2]. The has been a focus on Puppet and Chef, for example **(author?)** [8] looked at code quality by the measure of "code smell" of Puppet code. This tackles the problem by defining by best practices and analyzing the code against that. In the case of **(author?)** [9] it uses the docker linter in order to be able to analyse the files. For the CI systems we pick we will look into the tooling around that to aid the analysis.

III. METHODOLOGY

In order to answer the research questions we needed to find projects for CI configuration files. This is because we needed to get the contents of the configuration in order to analyse the structure of it. We chose to scrape Github via their API as it was easy to setup and test whether or not it was working as there is a one-to-one mapping between the API and user interface. Using Ghtorrent **(author?)** [10] it may have been easier to gather more data because the rate limiting wasn't as strict. Yet harder to test whether or not it is working. Therefore we decide to use the Github API as the source for our corpus.

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

In terms of which configuration files to pick we based our list from Github Welcomes all CI Tools blog post in 2017 [12]. In addition we added Github Actions and Azure Pipelines to the list as they are new and potentially popular systems.

basically want to expand on IV-Cmight also move some stuff out of RQ3 to make it more of a straight up comparison

Fig. 1. Example Github repository that has multiple configuration types in it [11]. (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"]
```
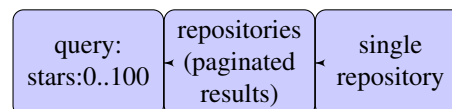
Fig. 2. Python configuration file used to specify what types of configuration to search for. The key specifies the name of the configuration and the value is the location in the repository the config should be found.

As can be seen in Figure 3 a query based on the number of stars a project has on Github. This is because we need a way of getting a large sample from Github without introducing too much bais into the sample. That is not too say that our method is perfect but it provides an easy way to get a large sample that includes projects with and without CI. Another potential solution would have been to use the "filename:travis.yaml" search API. However this did not provide information about
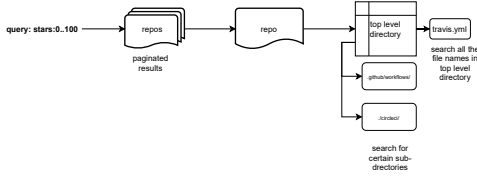
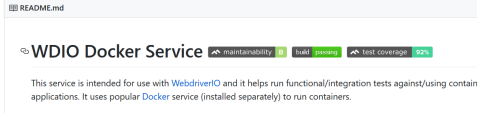Fig. 3. Diagram of the process used to search for projects with CI files in them

Fig. 5. Example Github project description and metadata[17]. The orange sections sections highlighted are the metadata that we scraped.

Fig. 4. Example of CI tag for Github ReadMe [13]

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

which projects did not use CI. Additionally for one unique search their can only be 1000 results returned by the Github API. To mitigate that limit we search based stars as we did do a search for a 1000 results per star count. The limitation of this 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 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.

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

We excluded Wrecker from the search because they represented a very small number of projects in comparison to the other projects. As it seems since the Github survey in 2017 they got bought by Oracle and from doing a search on Github for what we think based on the docs [14] and [15] for their configuration file naming convention. We were only able to find 20 results so did not include in the scraping script to speed up the process of searching for the other configuration file formats. Despite it being used in the **(author?)** [1] paper.

Along with information of what CI is being used for a project we also gathered metadata about the project. The available metadata through the API is largely what can been seen on a repository for example in Figure 5. We have the star count which is an indication how popular a project is as users can star projects that they like **(author?)** [16]. Then we have watchers which is users that have subscribed to the project to get notifications about the project.

A. Data corpus

This all produced a sample of 32,660 projects from open source projects on Github. As can be seen in Figure 6 we weren't able to scrape the whole star count range easily. This is because the script would crash when Github gave a 500 error
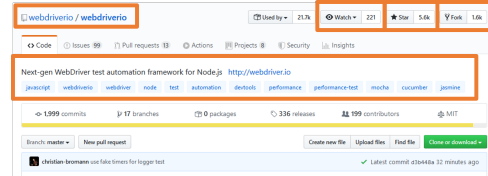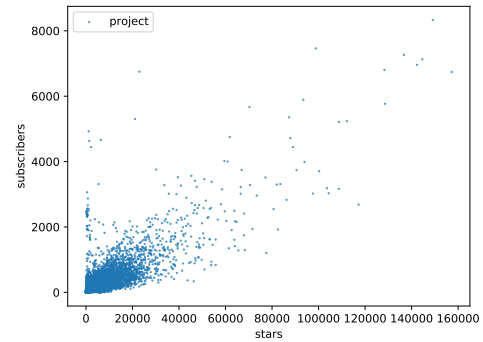

Fig. 6. Github stars against subscribers

B. Comparison corpus info

In **(author?)** [1] paper they use a similar method of using the Github API in order to create their corpus. Additionally they contacted Cloud Bees **(author?)** [18] to get a list of all open source projects that used their services. This helped them not to miss out on projects that they would otherwise missed out on. They kindly gave a copy of their final corpus.

However it does contain the data on the Cloud Bees projects which is 223 projects. As far as we can tell this only affects the comparison done in RQ2 IV-B. Additionally we found slight discrepancies between the paper and corpus in RQ1, RQ2 and RQ3, these being mainly 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 still be 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 two histograms to showing the density by the stars of the spread of data using the Sturge's rule (Figure 9). As we expected both corpses are skewed to the left. They use a logarithmic scale and initially it can look like there more data in 2020 corpus. However it is just more spread over the stars and potentially had a larger search range used. In Figure 7 there

is data after 40,000 stars potentially this could be down to the search method used or that most projects weren't dispersed over so many stars on Github.
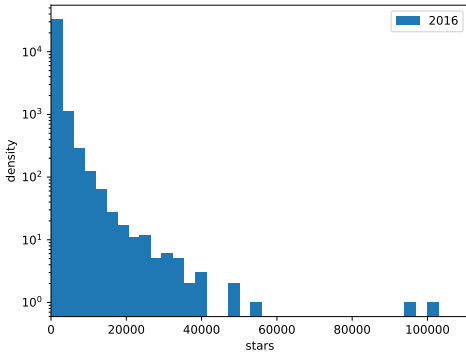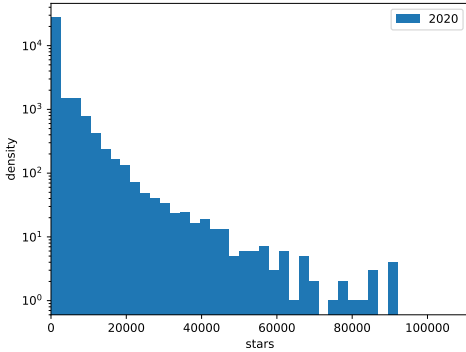

Fig. 7. 2016 corpus

Fig. 8. 2020 corpus

Fig. 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.

IV. USAGE OF CI

A. *RQ1: What percentage of open-source projects use CI?*

Out of the 94,379 projects 42.25% of them had CI configuration files in them indicating that they used CI in our dataset.

	2016		2020	
	count	percentage	count	percentage
CI	13,752	39.81%	37,417	39.65%
ReadMe has CI status	n/a		2462	2.61%
Multiple CI	n/a		1764	14.07%
No CI found	20,792	60.19%	54,500	57.75%
Total	34,544		94,379	

TABLE I
THIS TABLE SHOWS THE COMPARISON BETWEEN THE 2016 DATASET WHICH IS **(author?)** [1] 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 IV-A is the percentage of that 38% that has multiple CI in them. This is because configuration files can be used to CI or Continuos 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 (55.92%) projects with a ReadMe that had a CI status label that we could find. However this method is not perfect, for example "awesome-bootstrap-checkbox" by "flatlogic" [19] their ReadMe has the following line:

```
[![Dependency Status]
(https://img.shields.io/david/dev/flatlogic/awesome-bootstrap-checkbox
]
(https://www.npmjs.com/package/awesome-bootstrap-checkbox)
```

This contains `Status` and a url so we say it has got CI when the repository currently doesn't. Yet this is not the case for all of them as for example "SyncTrayzor" by "canton7" [20] uses AppVeyor but doesn't use a configuration file for it. Therefore we didn't find it as we searched for a configuration file only.

The percentage of CI projects in 2016 was 39.81%. If you look at Table 1 it shows that we got 42.25% CI projects. This shows a slight increase in CI usage however if we look at just projects where we can confirm to have found CI it is very close still at 39.65%.

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

REDO with new data to get the stats to line up but only metha

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

A further possible explanation Github's growth over the last 4 years (**author?**) [21] to 2019 (**author?**) [22]) so that Github is now at 40 million active users. It means that there are more projects that are using CD setups for building their static sites and in general Github is being used for more things that wouldn't require CI.

We think that the last two factors are the most likely contributors to why there is less CI usage now. Another important interesting aspect is that despite Github growing so much the CI usage rate has stayed relatively the same.

B. *RQ2: What CI systems are projects using?*

In Table II we, like all other research, found that Travis is the most popular CI system in use. However over the last 4 years the [12] CircleCi has lost out on it's rough quarter that it owned in the market. Notably the rise of Github Actions seems to have taken second place in the market share even though it is still very young in comparison as it was officially released November 13th 2019 but had a closed beta since the summer of 2019. This might not be down to the CircleCi losing out on their existing share, but maybe due to the rise in CI usage on Github. As well as because we searched Github for projects Github's CI product would be popular on their platform.

TABLE II
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%

double check this para- graph!

Our sample size of repositories is 31,494 that as it is a representation of projects on Github so won't account for the whole of it. This means that although Wrecker had the smallest count of CI when researching of 20 projects. In Table II we have configuration types that have lower counts. This is because that search for the 20 searched the whole of Github but the scraping was only able to do a small sample. Additionally there potentially could be faults in the scraping causing it show such low numbers for the last 3.

C. *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 (**author?**) [1] conducted this question. Then in this case we choose to categories via star count for each project.
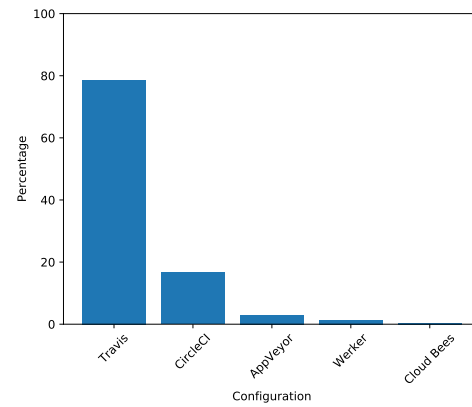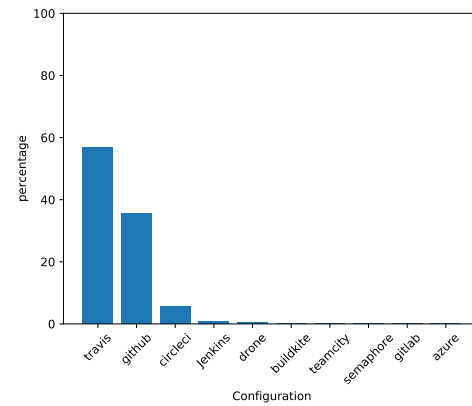

Fig. 10.   2016 corpus

Fig. 11.   2020 corpus

Fig. 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

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 7.

Figure 16 uses the same method as Figure 15 except it does it based the number of subscribers. Subscribers are users on Github who want to keep update on the changes to a project. This could range from core team members working on the project to people that want to be notified about a new release. In looking at this metric the hypothesis was that it would have a sharper rise in percentage of projects using CI per subscriber. However that was not the case overall as the gradient is not as strong. There is no comparison to [1] because their final corpus does not contain the subscriber count for each project.

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

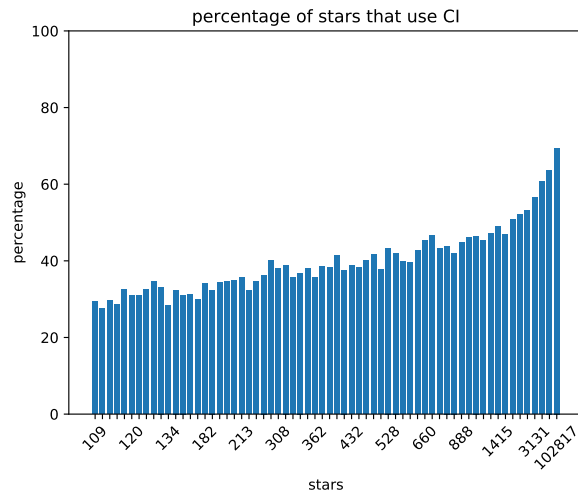Next we turn to what kind of programming languages are
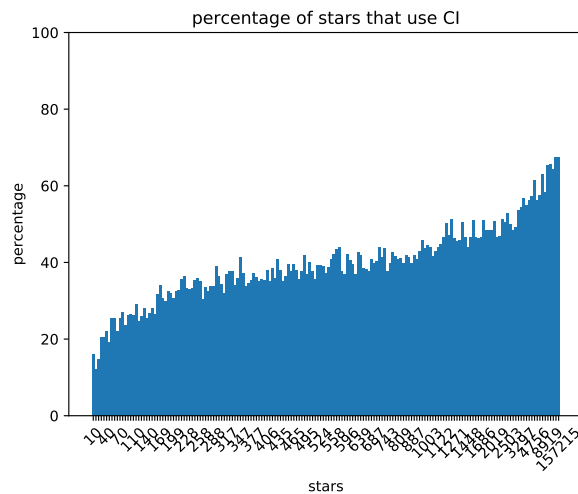
Fig. 13. 2016 dataset

Fig. 14. 2020 dataset

Fig. 15. In Figure 13 is the results from this research and in Figure 14 is the results from [1]. The results show the percentage chance of CI usage depending on the number of stars a group of 540 projects has on average.

Fig. 16. Subs graph

being used for CI. Figure 17 shows the top 20 programming languages by count and we can see that Javascript is the most common kind of project by a considerable margins. This was to be expected as it ties in with Github's annual report [22] on the platform they reported that Javascript has been the most popular programing language for the last 5 years. The interesting part is that our sample matches the rise in Python over Java. This is despite the fact that they are using "unique contributors to public and private repositories tagged with the appropriate primary language" and we are using the count of projects by primary programming language tag.

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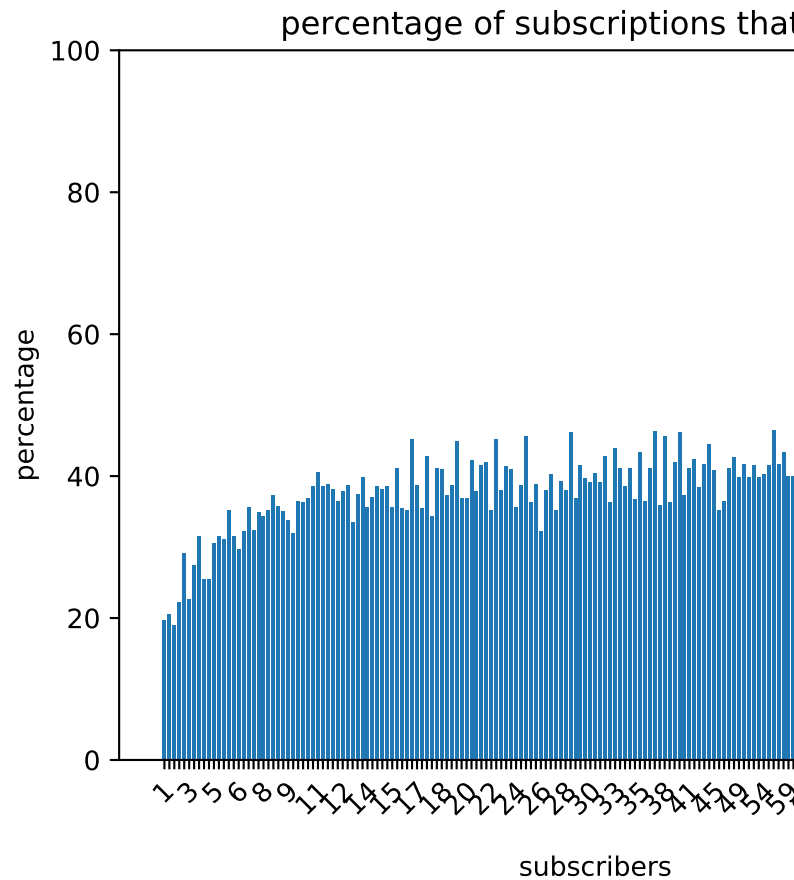
dot.

The most striking part of Figure 18 is the clear divide between different programing languages star count. The programming with the languages with the highest CI usage are Rust (94.6%), Typescript (86.56%) and Go (78.31%). This is interesting in that they are all fairly "new programming languages" in comparison to the others in the graph. They are all languages which are developed and open source on Github. In terms of Rust and Go it could be down to their tooling that comes built in to the language as that would lead to implementing CI to be a lot easier. Typescript is more a special case as it is a subset of Javascript so uses 'npm' to deal with dependency management which was some of the inspiration for Rust's tooling **(author?)** [23]. Older programming languages like Java and C# both have tooling for dependency management but the chances that they use CI is much lower. Therefore an area for further research would be whether or not the use of "modern" dependency management systems increases the chance of CI.

In figure 19 it shows the sample from 2016 in comparison to 18. The first major difference is in how the spread of languages by stars isn't as divided. This could be because of how the sample is not spread out as seen in Figure 9. The scatter graph is for the top 30 most common programming languages
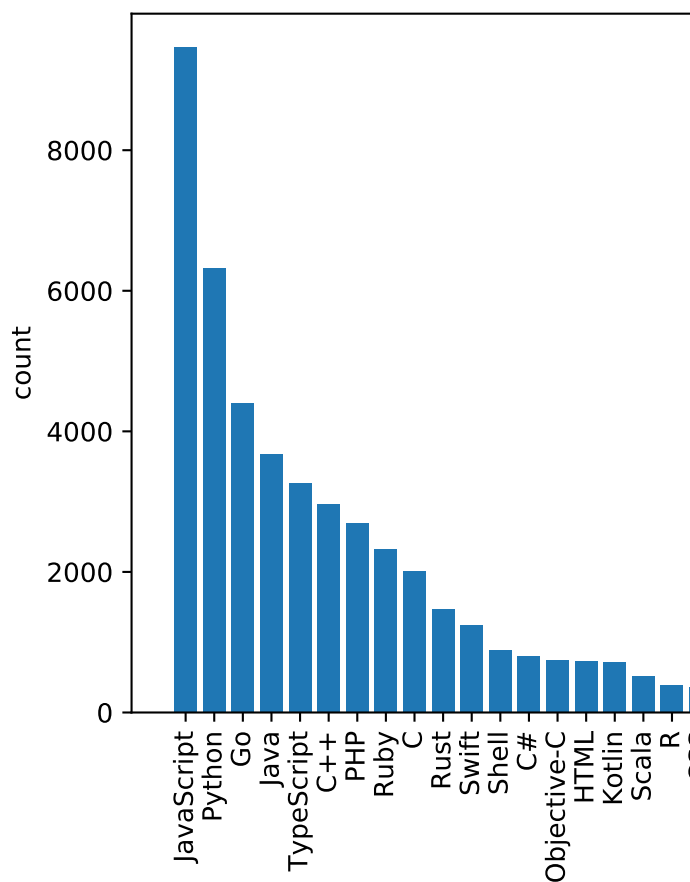
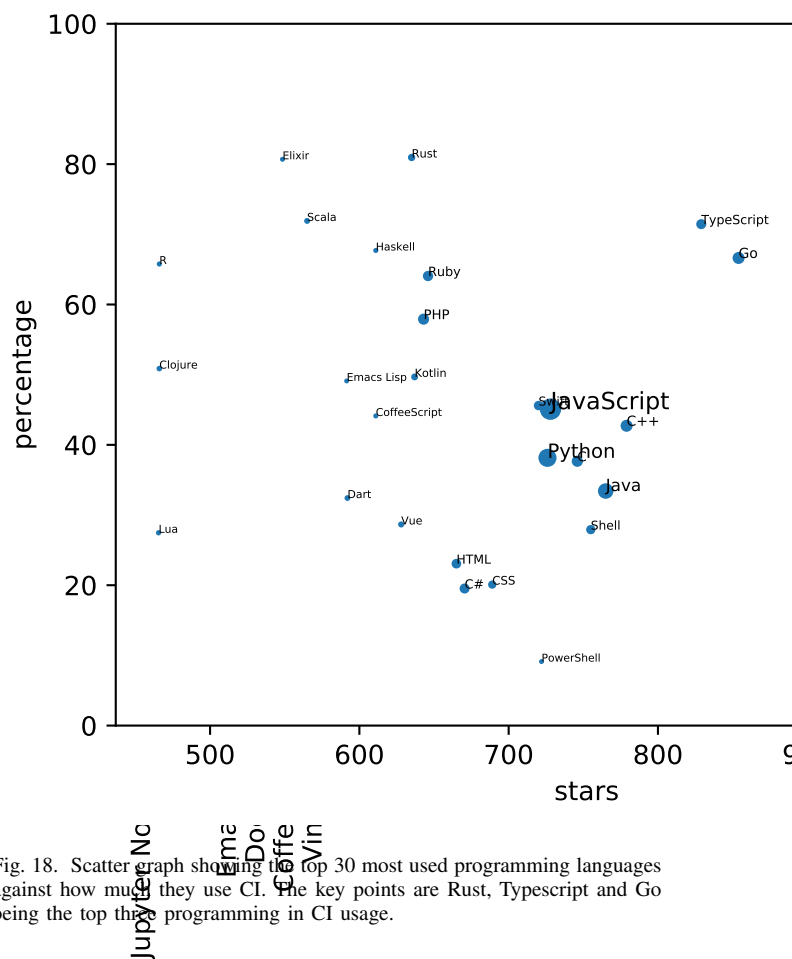Fig. 17. Count of top 20 programming languages used by projects using CI

Fig. 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.

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 **(author?)** [1] 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. 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.
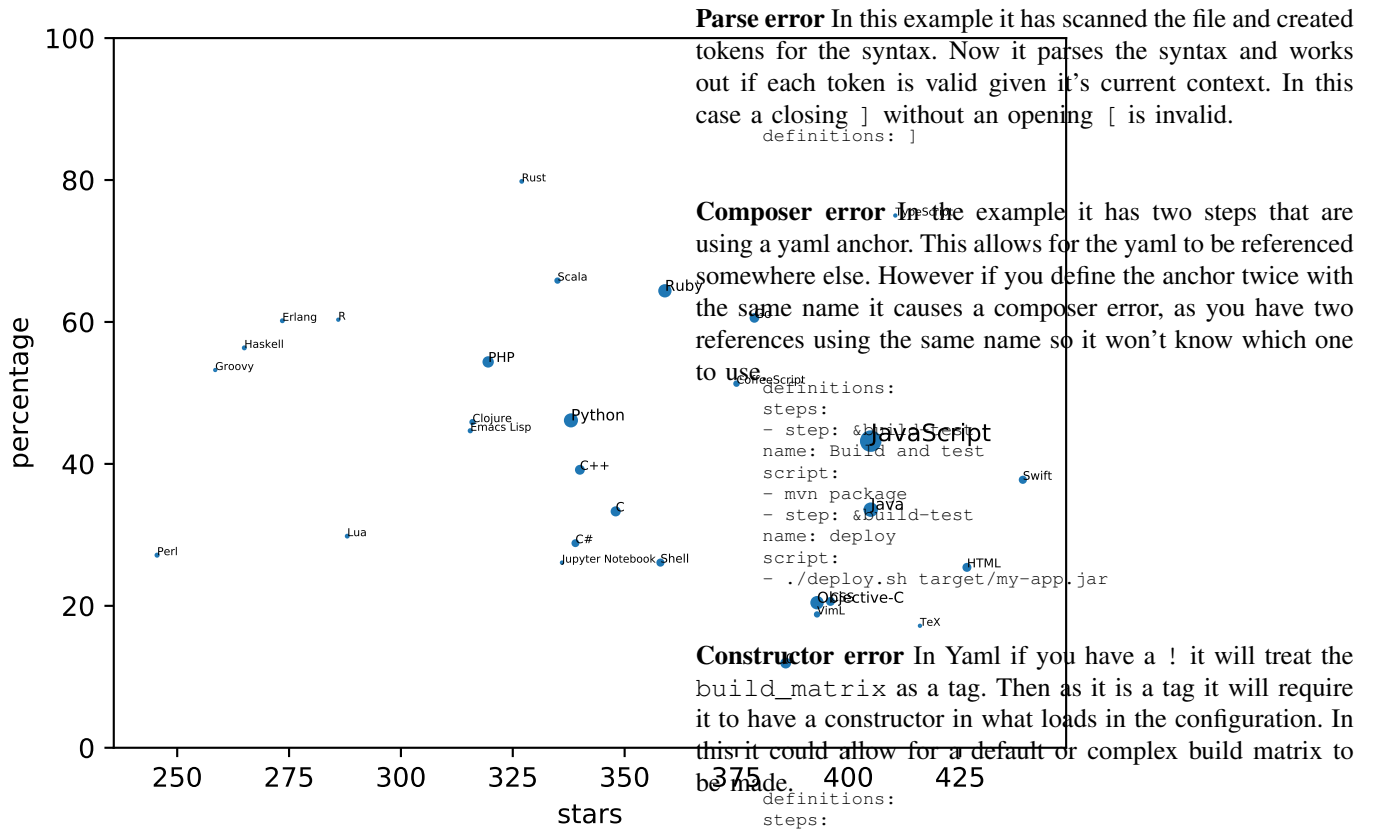
Fig. 19. Scatter graph showing the top 30 most used programming languages against how much they use CI for **(author?)** [1] from 2016. In comparison to Figure 18 the data is not as grouped as clearly by the star count. Rust, Typescript and then Scala have the highest programming CI usage.

V. STRUCTURE OF CONFIGURATION FILES

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

A. **RQ4**: What are the common errors when loading 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 **(author?)** [24] 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: ]
```

Composer error In this 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 III
YAML CONFIGURATION ERRORS

config	composer error	constructor error	parse error	scanner error	no. c
buildkite	0	0	0	1	100
circleci	6	0	3	3	2790
drone	70	1	0	0	211
github	0	1	4	28	1763
semaphore	0	0	0	3	28
travis	22	0	27	56	2814
azure	0	0	0	0	1
gitlab	0	0	0	0	1

As can be seen in the Table III there are 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 of Drone all the errors are for the same type of error. Potentially this could be because of how anchors are use a lot more in Drone configurations.

Travis is the largest configuration 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 with errors ones that are being changed when the scraping was being done.

This means that as the CI has been set up correctly for the other 99.632% as they are not needing to change because their our no yaml errors in it and presumably it is doing what they intend for it to do.

B. *RQ5: How are comments used in configuration?*

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.

```
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
```

Fig. 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 IV.

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.

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 IV
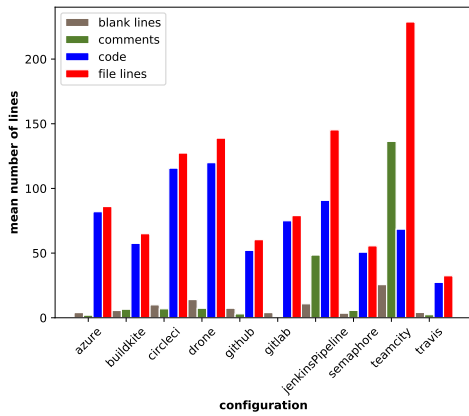LINE STRUCTURE ANALYSIS OF FIGURE 20

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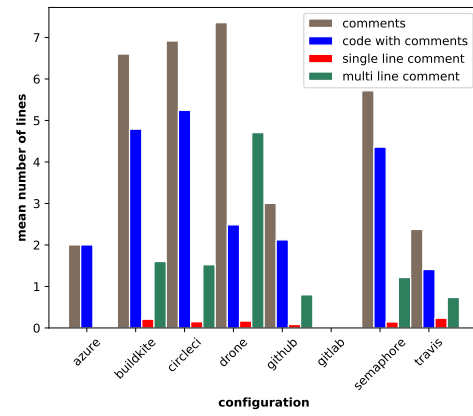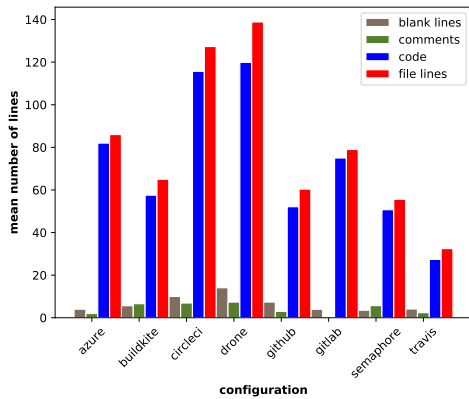

Fig. 22. Mean of line counts but only for yaml based configuration. Giving a clearer view on the breakdown of how little blank lines and comments are used.

Figure 22 shows how the comments are broken down for each CI service on average. In the case of Azure our sample size is only one project so that doesn't give us any significant insight. The blue line which represents code that has a comment after is the most commonly used kind of comment. Apart from for Drone which has more multiple line comments. This is really interesting as it highlights that comments tend to be tied to the code that is written. We had expected multiple line or single comments to be the most common kind of comment used. This is because code tend to follow a style guide to limit the maximum characters for a single line, for example in Python's **(author?)** [25] and Ruby's **(author?)** [26] which both use the same commenting syntax style guides. In doing so this rule also applies for comments as well therefore you don't normally have space for a code and a comment. However the code with comments is the most common type of comment shown in 23.

In the case of Jenkins pipelines and Teamcity there is a much higher usage of having code with comments. Therefore we have separated the analysis and comparison from Figure 21 to Figure 24 for the non yaml based configuration. Jenkins

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

and Teamcity configurations is Kotlins based and for TeamCity is also xml based. The key difference we find here is that multiple line comments are 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 multi-line comments to be done easily.
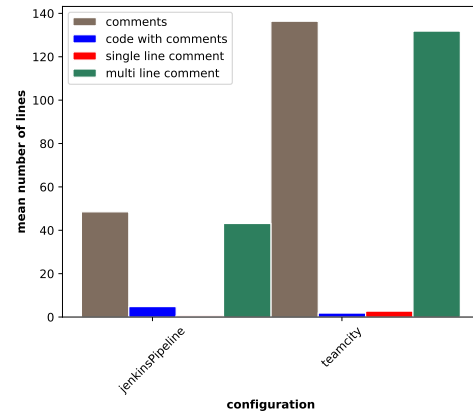

Fig. 24. Mean of line counts for comments, code with comments, single line comments and multiple line comments for none yaml configuration files

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

From labelling the comments as shown 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

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. //===).
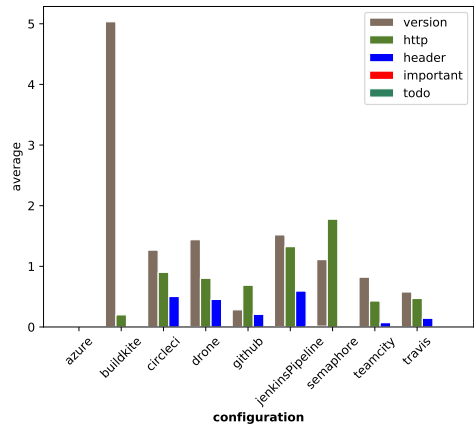
Fig. 25. Comment types

C. RQ6: Are external scripts used within the configuration?

An external script is typically a Bash or Powershell script, 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 is executed on a users machine. Therefore you can get some setups where you have scripts defined for building and deploying the code that the users and CI 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 know the reasoning for the high numbers further research needs to be done.
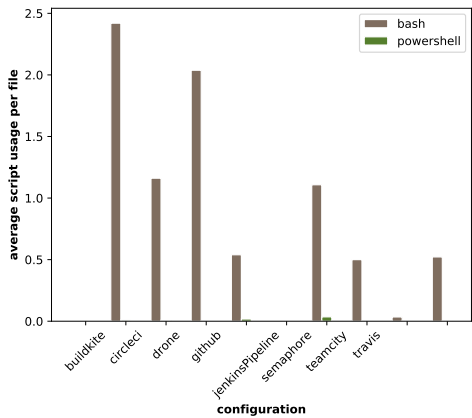
Fig. 26. Mean script usage for each file

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

likely to be from a configuration template or commenting out configuration.

Add a new conclusion if

	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 V

RAW COUNT AND PERCENTAGE CHANCE OF EACH CI SERVICE
CONTAINING AN EXTERNAL SCRIPT

very different from a Travis setup. Being able to get the correct data to be able to distinguish and analyse both situations.

VI. THREATS TO VALIDATING

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

Firstly if we assume that the scraping works perfectly then it only has a maximum of 1000 open source projects per star. That is excluding closed source projects which would range from personal projects to companies. Additionally we only used data from Github not from Gitlab, bitbucket or other version control hosting services. This leads to bais in the data for example if Gitlab was also scraped then we would get a lot more Gitlab CI files. However in order to get the best spread of data Github has the best API and most of their services do not tie you down to use only their service. Also although we could get a 1000 projects per star we were still able to get around 30,000 projects and a wide spread across Github. The key aspect being that because it was a sample we focused on getting a good spread of data.

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

Thirdly, identifying which projects are programming projects or would have a need for CI on Github and which are not. Based on the research [27] it is important to filter out repositories that aren't part of the question being asked. Therefore we could have looked to try and filter out Github static sites and other none software based projects. However if assume a certain type of project won't be using CI then we would be introducing bais when trying to answer how CI is used. For further research better categorising of projects would help a lot. The major difficulty would be if for example you had a static site that ran on Netlify that will be using CI/CD but

VII. Summary

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

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

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

A. Discussion and further research

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

Further research into usage that we would like to do is to look into how the size of the project affects the chance that it uses CI. Then looking at the usage of scripts within CI configuration, for example using a script tag to run a shell script, as while doing the research we found some projects use scripts a lot while others just used the CI configuration. This would lead to questions around which CI system have a higher amount of scripts used, but also looking at how much they enable them to be used and what is the size of those scripts. The data for the programming language and version(s) is in the configuration. Therefore it would be possible to work out how much 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. [28], [29], [30]) is used more for projects with CI and those that don't.

VIII. Acknowledgement

We wish to thank Michael Hilton in particular for providing the corpus for their research **(author?)** [1].

References

[1] K. H. Michael Hilton, Timothy Tunnell, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects | Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering," 2016. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/2970276.2970358

[2] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Information and Software Technology*, vol. 108, pp. 65–77, Apr. 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584918302507

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

[4] P. Copeland, "Google's Innovation Factory: Testing, Culture, and Infrastructure," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ser. ICST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 11–14. [Online]. Available: http://dx.doi.org/10.1109/ICST.2010.65

[5] K. Gallaba and S. McIntosh, "Use and Misuse of Continuous Integration Features: An Empirical Study of Projects that (mis)use Travis CI," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[6] M. Fowler, "Continous integration," 2010. [Online]. Available: https://www.martinfowler.com/articles/continuousIntegration.html

[7] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in GitHub," ser. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, Aug. 2015, pp. 805–816. [Online]. Available: https://doi.org/10.1145/2786805.2786850

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

[9] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall, "An Empirical Analysis of the Docker Container Ecosystem on GitHub," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, May 2017, pp. 323–333, iSSN: null.

[10] "Ghtorrent website," 2020. [Online]. Available: https://ghtorrent.org/

[11] J. Ling, "Cu worhsip song list creator - a repository taken over for testing," 2019. [Online]. Available: https://github.com/JosephLing/cuWorshipSongListCreator

[12] Github, "Github welcomes all ci tools," in *Github welcomes all ci tools*, github.com, Ed., 2017. [Online]. Available: https://github.blog/

2017-11-07-github-welcomes-all-ci-tools/

[13] S. Tsvilik, "wdio-docker-service," 2020. [Online]. Available: https://github.com/stsvilik/wdio-docker-service

[14] Wrecker and Oracle, "Wrecker ci development blog," 2018. [Online]. Available: https://devcenter.wercker.com/development/cli/usage/developing/

[15] GitHub, "github filename search for wrecker.yml files," 2020. [Online]. Available: https://github.com/search?q=filename%3Awrecker.yml

[16] H. Borges, A. Hora, and M. T. Valente, "Understanding the Factors That Impact the Popularity of GitHub Repositories," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct. 2016, pp. 334–344, iSSN: null.

[17] webdriverio, "webdriverio," 2020. [Online]. Available: https://github.com/webdriverio/webdriverio

[18] "Cloudbees website," 2020. [Online]. Available: https://www.cloudbees.com/

[19] "flatlogic/awesome-bootstrap-checkbox," 2020. [Online]. Available: https://github.com/flatlogic/awesome-bootstrap-checkbox

[20] A. Male, "canton7/SyncTrayzor," Apr. 2020, original-date: 2015-02-08T17:08:40Z. [Online]. Available: https://github.com/canton7/SyncTrayzor

[21] "GitHub State of the Octoverse: 2016," 2020. [Online]. Available: http://octoverse.github.com/2016

[22] Github, "Octoverse - top languages," p. Top languages, 2019. [Online]. Available: https://octoverse.github.com/

[23] "Cargo: Rust's community crate host | Rust Blog," 2020. [Online]. Available: https://blog.rust-lang.org/2014/11/20/Cargo.html

[24] "Yaml faq," 2020. [Online]. Available: https://yaml.org/faq.html

[25] "PEP 8 – Style Guide for Python Code," 2020. [Online]. Available: https://www.python.org/dev/peps/pep-0008/#maximum-line-length

[26] ">Ruby Style Guide," 2020. [Online]. Available: https://www.w3resource.com/ruby-programming/ruby-classes.php#line-length

[27] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining GitHub," ser. MSR 2014. Hyderabad, India: Association for Computing Machinery, May 2014, pp. 92–101. [Online]. Available: https://doi.org/10.1145/2597073.2597074

[28] V. Driessen, "A successful git branching model - git flow," 2013. [Online]. Available: https://nvie.com/posts/a-successful-git-branching-model/

[29] Github, "Github flow introduction," 2017. [Online]. Available: https://guides.github.com/introduction/flow/

[30] P. Hammant, "What is trunk based development," 2013. [Online]. Available: https://paulhammant.com/2013/04/05/what-is-trunk-based-development/