Characterizing the usage of CI tools in ML projects

Dhia Elhaq Rzig
University of Michigan - Dearborn
Dearborn, MI, USA
dhiarzig@umich.edu

Foyzul Hassan
University of Michigan - Dearborn
Dearborn, MI, USA
foyzul@umich.edu

Chetan Bansal
Microsoft Research
Redmond, WA, USA
chetanb@microsoft.com

Nachiappan Nagappan
Microsoft Research✳
Redmond, WA, USA
nnagappan@acm.org

ABSTRACT

Background: Continuous Integration (CI) has become widely adopted to enable faster code change integration. Meanwhile, Machine Learning (ML) is being used by software applications for previously unsolvable real-world scenarios. ML projects employ development processes different from those of traditional software projects, but they too require multiple iterations in their development, and may benefit from CI. **Aims:** While there are many works covering CI within traditional software, none of them empirically explored the adoption of CI and its associated issues within ML projects. To address this knowledge gap, we performed an empirical analysis comparing CI adoption between ML and Non-ML projects. **Method:** We developed `TraVanalyzer`, the first Travis CI configuration analyzer, to analyze the CI practices of ML projects, and developed a CI log analyzer to identify the different CI problems of ML projects. **Results:** We found that Travis CI is the most popular CI tool for ML projects, and that their CI adoption lags behind that of Non-ML projects, but that ML projects which adopted CI, used it for building, testing, code analysis, and automatic deployment more than Non-ML projects. Furthermore, while CI in ML projects is as likely to experience problems as CI in Non-ML projects, it has more varied reasons for build-breakage. The most frequent CI failures of ML projects are due to testing-related problems, similar to Non-ML and OSS CI failures. **Conclusion:** To the best of our knowledge, this is the first work that has analyzed ML projects' CI usage, practices, and issues, and contextualized its results by comparing them with similar Non-ML projects. It provides findings for researchers and ML developers to identify possible improvement scopes for CI in ML projects.

CCS CONCEPTS

• **Software and its engineering**; • **Computing methodologies** → *Machine learning*;

1 INTRODUCTION

"The whole point of Continuous Integration is to provide rapid feedback". This is how Martin Fowler [1], who helped popularize CI, describes it. CI is a software development process for shared repositories that automatically integrates the changes committed by their developers. CI allows its adopters to catch bugs earlier, increase the frequency of their releases, and integrate pull requests faster [39]. Its adoption has grown from 40.27% in 2016 [39] to 68% within larger teams in 2018 [3]. Machine Learning (ML) projects have also seen an explosion in both usage and importance in recent years [37, 47]. Instead of being given an explicit solution like traditional software, ML projects, attempt to solve a problem by analyzing data, testing their findings, evaluating their results, and iterating on these phases. However, similar to traditional software, ML projects rely on iteration within their development. Indeed, the automation prowess of a CI system may be a great fit for ML projects' need for iteration. However, it's notable that most CI tools were conceived before ML project development became mainstream, and that both CI tools and ML projects have their specific problems. For example, debugging CI build failures and errors can be non-trivial due to complex logs [54], and ML projects require new development processes and practices such as data engineering and model management [43], or require a different approach to existing processes in comparison to traditional software, such as the example of traditional testing being ineffective on ML projects [40]. Yet, there is a gap in research concerning the adoption of CI within ML projects, the tasks performed by CI within them, as well as the problems CI tools face when they are used in these projects. In this paper, we aim to fill these knowledge gaps by identifying the adoption rates, current practices, and common failures and errors related to CI within ML projects. We used a triangulation-based [29] method to estimate the adoption rate of CI on a set of 4031 ML projects and 4076 Non-ML projects. Then, for our detailed CI analysis, we selected 476 ML and 202 Non-ML projects out of the larger set, using the same criteria of CI adoption and main programming language on both project categories. Using `TraVanalyzer`, our Travis CI AST analyzer, we determined the CI build goals of these projects. Using our CI log analyzer, we analyzed these ML and Non-ML projects' builds, and their associated job failure and error logs, from which we determined the CI problems these projects encountered. With our analysis, we answered 3 key questions:

RQ1: *What is the adoption rate of CI among ML projects?* Around 37.22% of ML projects have adopted CI, which is below CI's adoption rate by our set of Non-ML projects, estimated at 45.12%, as well as that of Open Source Software (OSS) overall, estimated to be between 45% and 68% [3]. We also found that Travis CI, the most popular CI tool for our Non-ML project-set and Open-source software on GitHub [39], is the most popular tool for ML projects as well, with no sign of adoption of ML-projects-specific CI tools.

RQ2: *What tasks does CI perform for ML projects?* Similar to traditional software [30], Testing and building software are the most

✳ Nachiappan Nagappan was with Microsoft when this work was done. He is now with Meta Platforms

common tasks. Code analysis is the third most common, and deployment is the least common. Surprisingly, these tasks are used more often within CI of ML projects than CI of Non-ML projects.
RQ3: *How often and Why do CI builds break in ML projects?* On average, 23.87% of the CI builds of a project fail, and 14.09% of the builds are errored, both of these build types are considered non-successful. Build breakages in general occur at similar levels between our sets of ML and Non-ML projects, but are more common in both than in Open Source Software (OSS) overall. The most common failures were caused by failed tests, errored tests, and failures related to Code Analysis, which are also common within our comparison set of Non-ML projects. However, ML failures show more variability in terms of causes linked to failure of a project's build.
In summary, we make the following contributions:

- The first comprehensive analysis of CI adoption by ML projects on GitHub.
- The first Travis CI configuration AST analyzer `TraVanalyzer` which determines CI tasks.
- The first CI log analyzer specifically-designed for the detection and classification of CI problems within Python-based ML and Non-ML projects.
- A comprehensive taxonomy of CI problems in Python-based ML and Non-ML projects, that can facilitate the fix pattern analysis.

2 BACKGROUND

Continuous integration (CI) was first introduced by Grady Booch in 1991 [28], and this concept began to gain popularity in the early 2000s partially due to support from Martin Fowler [1]. The founding principle behind CI is frequent integration of code from the different developers of a shared repository, arising from the time-consuming and difficult task of code integration that software projects without CI need to perform [1, 28]. In general, Continuous Integration servers and tools automate integration by automatically validating newly-pushed commits via the execution of building and testing processes. CI can also include other automated tasks like code analysis tasks, such as linting and code coverage, or deployment tasks. A variety of CI tools and services exist, and more recently, a few CI tools and services have been specifically designed for ML projects or their components. Tools and services that provide testing and versioning for ML projects, such as Kuberflow [19] and Amazon Sagemaker [5], are generally only concerned with ML models, and the corresponding ML project's code base is managed via a traditional CI tool such as Travis CI or GitHub Actions. As a result, the majority of these aforementioned tools are generally used in conjunction with traditional CI tools and cannot fully replace them. Furthermore, we found no evidence of their usage within our project set. In fact, Travis CI is also the tool that enjoys the highest usage within our set of ML projects, as detailed in Section 5.1.

Indeed, The majority of CI-related processes, such as building and testing, and their corresponding tools were established and designed for traditional software projects, and may not be well-suited for ML projects. For example, Unit Testing is a well-established practice of testing functional code by comparing its results against the expected results as defined by the test's author [45]. Applying this same approach to ML projects by evaluating their results on the same testing set repeatedly can cause problems with their accuracy [40]. Another example is automatic deployment, where for traditional projects, the software and its configuration are bundled into a deployable archive, such as a JAR, or deployable image, such as a Docker-image, and then are pushed to their respective endpoints. For ML projects, the most frequently updated component is the ML model [4], the deployment strategies of which may differ from those of other components. In spite of these differences, Travis CI, one of the most widely used CI tools [39], is the most popular CI tool of the ML and Non-ML projects we analyzed.

A Travis CI workflow is described via a *.travis.yml* file, written in YAML-based [24] Domain Specific Language (DSL), where certain settings-keywords can configure the environment or execute a certain process. A workflow is generally referred to as a build and can be composed of one or more stages that run sequentially, and each stage forms a specific subset of the overall build. Stages can be configured with the `stage` keyword, and by default, a build is composed of only one stage. Each stage may be composed of one or more Jobs that run in parallel, each executing the same sub-script in a specific environment different from the other jobs. This is configured via the *build matrix*, where two or more jobs can be set to run in parallel in each stage, by specifying a different environment for each. The keywords `OS` and `language`, which respectively set the Operating System of a job's container and prepare it by installing the tools of a specific programming language, can be used multiple times and with different values. For example, `OS:linux` with `language:Java` and `OS:linux` with `language:ruby` will configure two Jobs that run in Linux containers, one of which is configured for Java projects and the other one is configured for Ruby projects.

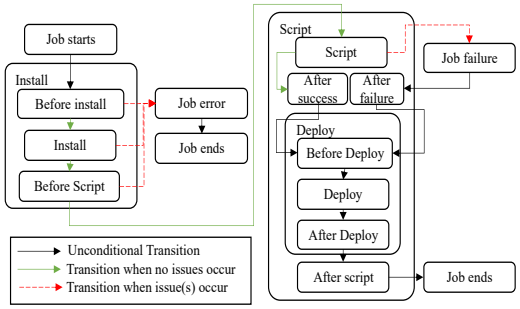

Figure 1: Travis CI job state machine

In further detail, a generic job's state machine is illustrated in Figure 1. A job has two phases, the install phase, meant for the installation of any dependencies and preparation of the environment, and the script phase, which runs the build, test, and other tasks specified by the developer. An **errored job** is a job that experiences an issue during the install phase of its execution, after which it immediately stops executing. A **failed job** is a job that experiences an issue during its script phase, after which it executes its "after failure" section, and the rest of its script phase. A **failed build** is a build with 1 or more failed jobs and no errored jobs. An **errored build** has one or more errored jobs. If no problems occur, the job(s) and the build are considered **passed**. A notable exception is that if a job experiences an issue only during its `Deploy` or `After Script` phase, it's still labeled as passed. A build can be manually canceled by the developer, giving it and all its associated jobs the **canceled** state.

3 RESEARCH METHODOLOGY

3.1 Dataset

In this work, we used two data sets: the larger one is referred to as the *Breadth Corpus*, on which we performed analysis to inform us about the state of CI adoption in ML projects, and the smaller one is referred to as the *Depth Corpus*, a subset of the breadth corpus on which we performed detailed analysis to inform us about CI usage goals, and CI problems within ML projects.

3.1.1 Breadth Corpus. Our goal was to analyze CI adoption within a set of open-source and active Machine Learning (ML) projects, and a similar comparison set of Non-ML projects. We define ML projects, also referred to as ML-enabled systems, as those with the goal of producing systems that have ML capabilities as part of their features. For this task, we initially chose to analyze the data set of projects proposed by Gonzalez et al. [37]. This data set is composed of 5224 ML projects and 4101 Non-ML projects. However, we found several problems with it via manual inspection, such as the inclusion of toy projects and study guides among others. To resolve this problem, two of the authors re-curated the ML projects by reading the descriptions on their main GitHub pages and any websites linked to by those pages, and they removed 1193 projects. These projects either did not use ML, or were toy projects, or study guides, or another type of repository which did not constitute a software project. We were unable to obtain 25 Non-ML projects from the original dataset due to delisting. The new set of projects which forms our *Breadth Corpus* contains:

- **4031 ML projects:** These projects are composed of ML frameworks and libraries such as Tensorflow, as well as ML applications such as Faceswap. All of these projects employ Machine Learning based techniques or components, and they either meet a specific need for the user, or have a general-purpose usage and can be used by other developers for a specific goal.
- **4076 Non-ML projects:** These projects are considered traditional software applications, such as websites, desktop or mobile applications etc., which do not contain or use ML-based components or technologies.

3.1.2 Depth Corpus. After estimating the adoption rates of different CI tools as detailed in Section 3.2.1, we found that Travis CI was the most popular CI tool for ML projects, as detailed in Section 5.1, which is also true for the Non-ML projects we analyzed and open-source software in general [39]. Furthermore, for the ML projects that used Travis CI, Python is the main language for 51.06% of them, as reported by the GitHub API [16], and no other language was the main language for more than 9% of them. This aligns with the results found by Gonzalez et al. [37] concerning Python being the most popular language for ML projects. As a result, we selected Python-based ML projects with one or more Travis CI builds since they represent the majority of CI-using ML projects. We then applied the same selection criteria of Python as a main programming language and CI-usage to our Non-ML set of projects to obtain a comparison set, which produced a smaller number of projects. This is expected since Python is a main programming language of only 14.95% of Non-ML CI-using projects. Our depth corpus is composed of:

- **476 ML** Travis CI-using Python ML projects.
- **202 Non-ML** Travis CI-using Python Non-ML projects.

3.2 Approach

In this section, we illustrate the different steps we took to select and analyze our project set. An overview of our approach is in Figure 2.

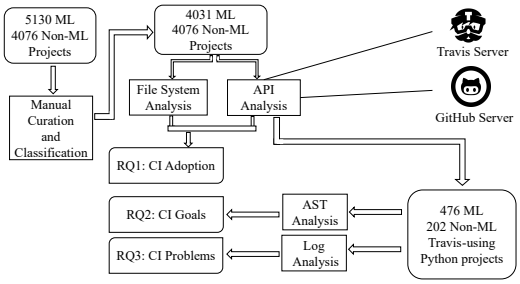

Figure 2: Overview of Research Methodology

3.2.1 CI Adoption Analysis. Currently, there is no standardized approach to determine if a GitHub repository is using a CI tool. In order to determine the adoption of different CI tools within our set of ML projects, we developed a two-pronged approach to estimate the adoption rates of the different continuous integration tools, based on methods followed by Hilton et al. [39] and Gallaba & Mcintosh [36]. First, we considered a list of CI tools proposed by Leite et al. [42], but this list included only 4 tools and was developed by examining traditional software projects. To enrich it, we selected the top 1000 ML and the top 1000 Non-ML projects in our *breadth corpus* and listed their filenames which had non-source-code extensions. Then, 2 authors attempted to match their names with the naming conventions of configuration files of CI tools in order to identify any other CI tools in our project set. We also added support for tools such as ease.ml\ci, that rely on the same configuration files of other tools by introducing their own segments, since the filename-based approach would not allow us to detect their usage. Second, we scanned the projects' repositories [1] to determine if they had files or file segments indicative of the usage of a specific CI tool, which we collected in the previous step. Finally, we selected the CI tools with an adoption rate of 5% or more as measured by our first File-system-based approach, which was GitHub Actions and Travis-CI, and then queried their APIs and used the existence of one or more builds to establish a project's adoption of a specific CI tool. However, we found conflicts between the two data sources: some projects had CI configuration files without any builds in the corresponding CI tool's server, or vice-versa. For example, while we found 852 ML projects with Travis files in their repositories, only 559 of them also had builds on Travis's server, and we found an additional 376 ML projects which had Travis builds but did not currently have a Travis configuration file in their repository. Even more notable differences were found in the case of GitHub actions, where 858 ML projects had configuration files for this tool, but only 477 of them had GitHub Actions builds, and we did not find any ML projects which had GitHub Actions builds but no configuration files. To resolve these conflicts and avoid false positives or false negatives regarding CI adoption, we applied a triangulation-based method [29] and defined two types of CI adoption:

[1] Last updated on 08/13/2021

Historical Adoption: A project that has builds on a specific CI tool's server, but does not have the CI tool's configuration files in its current repository, is assumed to have used the CI tool in the past.

Current Adoption: A project that has builds on a specific CI tool's server and has the tool's configuration file(s) within its current repository, is assumed to be currently using the CI tool.

This allowed us to resolve any data conflicts and answer **RQ1** regarding the popularity of CI within ML projects. The complete list of CI tools we considered is: AppVeyor [10], Buildbot [7], CircleCI [8], Cloud Build [9], CodeBuild [25], GitLab CI [17], Jenkins [20], Travis CI [22], GitHub Actions [15], VSTS [11] and ease.ml\ci [14].

3.2.2 CI Task Analysis. CI configuration files (e.g., .travis.yml) define continuous integration tasks such as build, test, deployment, etc. Travis CI adopted a DSL that is based on YAML. However, analyzing a Travis CI configuration file is not trivial as it can invoke system commands, external shell scripts, Python scripts, among others, and can also include steps to integrate ML components (e.g., data, model, etc.). To overcome these challenges, we developed the first .travis.yml AST [44] analyzer, `TraVanalyzer`, to parse Travis configuration files from our *Depth Corpus* projects, and applied a command clustering approach to group commands to allow for manual annotation and analysis. Details of the parser and the AST-based command clustering approach are discussed in the subsequent paragraphs.

AST Parsing of CI Configuration File:

Since Travis CI configuration files are written in a domain-specific language (DSL) language extended from YAML, we chose to extend the Java-based YAML parser `DocConverter` [12] to extract top-level entities of the configuration file. Figure 3 shows such an example where, in Phase I, top-level configurations are parsed in an AST format. However, based on Travis CI Documentation [6], `install, script, before_install, before_script, after_script, after_success, after_failure` job phases can invoke external system commands and bash syntax that includes if-else conditions, looping, variable usage, etc. Figure 3 Phase I shows an example of that with a script-block that contains an embedded bash script that uses an if-condition to invoke the `make flake` command. We developed a Bash script parser that can parse and extract such embedded scripts, and generate their ASTs. Since Bash scripts support variable assignment and usages, we applied a data-flow analysis [34] on the extracted AST to analyze its condition checks. However, in many cases, the scripts use system environment variables such as `TRAVIS_OS_NAME` for if-else conditions. Since system variables cannot be determined by just analyzing the embedded scripts, we considered these conditions as always true. After generating the AST for the embedded script, we extended the Phase I AST to generate the Phase II AST with Bash annotations. We used the Phase II AST as depicted in Figure 3(c) for the CI task analysis of ML projects. On a programmatic level, `DocConverter` allows us to ingest the file into an object, which we then convert to a Tree object, as represented in Phase I. We then apply the embedded script parsing to generate the object represented in Phase II.

Command Clustering and Task Analysis:

With the ASTs generated for the Travis CI configuration file, we can analyze the different stages and properties described in it, such as the language and OS. However, the purpose of external tools and commands invoked is difficult to determine. To solve this problem, we extracted the commands from the Travis AST objects, generated from the .travis.yml files belonging to the ML and Non-ML projects within our depth corpus, and applied AST-clustering based on the commands' names. We chose to omit the commands' parameters as they are often project-specific. In total, we extracted 258 distinct commands and/or tools, corresponding each to one cluster. Two of the authors then manually reviewed the tools and commands documentations' to categorize them into build, test, code analysis, and deployment tools. This categorization is used in order to answer **RQ2** regarding the CI tasks of our *Depth Corpus* projects.

3.2.3 CI Problem Frequency and Taxonomy Study. Continuous integration workflows may experience failures or errors due to a variety of reasons. To determine the build breakage[2] and success rates of the projects in our *Depth Corpus*, we used the Travis CI API via PyTravisCI [21] to obtain information about their respective builds and jobs, as well as the logs pertaining to their failed and errored jobs. Within this analysis, we opted to use the CI information of each project within one year of a project's most recent build[3]. We opted for this moving window of dates in order to consider only the most recent builds of each project, and we chose this relatively large window to minimize the chance of excluding information from less active projects. We found a total of 79868 builds of our ML projects and 7519 builds of our Non-ML projects using this moving-window criterion. The higher total number of builds for ML projects can be explained by the inclusion of projects such `RasaHQ/rasa_core` and `Cloud-CV/EvalAI` with builds as high as 4715 and 1680 in the selected window, while the highest number of builds for a single project was 733 for Non-ML projects within `numenta/nupic`. We then applied the filtering process proposed by Gallaba et al. [35] to avoid including duplicate builds in our analysis, which resulted in the removal of 6157 passed, 2144 failing, and 1163 errored builds of ML projects, as well as the removal of 694 passed, 708 failing, 357 errored builds of Non-ML projects. Finally, we generated the average percentages of successful, failed, errored, and canceled builds for each project, considering only their filtered builds.

Examination of existing taxonomies:

When it comes to classifying the issues causing a job failure or a job error, a number of taxonomies exist: Beller et al. [27] classify build failures into two categories depending on whether or not a build failed because of a test, but this has limited usage since it does not clarify the reasons behind a build failure when tests are not the cause. Durieux et al. [31] analyzed build failures to extract the issue causing a job failure. But some of their issue categories were overly specific, such as the "Gem file not found" category that does not generalize to projects not using Ruby, and some of them lacked detail, such as the "Travis Limitation" category which can be a log file length restriction, a timeout, or another Travis-related limitation. Rausch et al.'s work [48] classifies failures into a variety of categories related to Java projects but doesn't generalize to other languages or projects other than those studied. For example, their "androidsdk" category would only occur in projects using the Android SDK. Finally, none of these aforementioned taxonomies focus

[2] a broken build refers to a failed or errored build in the context of CI

[3] before August 13th, 2021

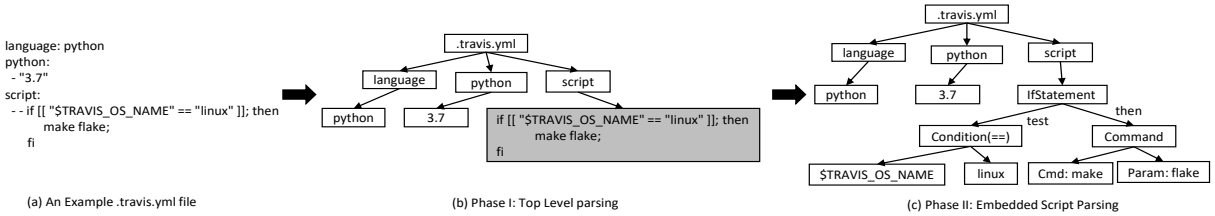

Figure 3: Overview of Parsing an example .travis.yml file in AST Format

on ML projects, as they were developed for traditional software. When testing the preexisting log analyzers associated with the aforementioned taxonomies, we found none of them reached a satisfactory saturation rate, which in our context is the percentage of failure logs within which a failure type was detected. For example. Travis Listener's Log Parser [31] failed to detect the failure type within 71% of 7655 randomly selected job failure logs pertaining to projects from our *Depth Corpus*, and Rausch et al.'s [48] classifier would not work since it is intended for Java projects, while the projects we are analyzing are Python-based. Overall, we found these existing taxonomies either have a very broad categorization that lacks detail, or a narrow categorization that does not directly relate to the failure and error classifications of the Travis CI builds that we specified. Furthermore, none of these taxonomies or their associated tools were designed for Python-based ML projects, and we found them inadequate for the job logs we aimed to analyze.

Creation of a new taxonomy:

In order to resolve the aforementioned problems and answer **RQ3** regarding the underlying reasons behind job failures and job errors, we created a new taxonomy and its associated log analyzer. We used open coding [41] to build our taxonomy of failures and errors and their associated log analyzer composed of regexes and scripts. During this step, we considered the following builds, selected via the filtration process described earlier: 1262 failed builds and 1144 errored builds belonging to Non-ML projects, and 19621 failed build and 7014 errored builds belonging to ML projects. Then, we attempted to obtain the logs pertaining to job failures connected to these broken builds, and succeeded at downloading 35965 and 7153 job failure logs of ML and Non-ML projects respectively. In the following step, we randomly selected our first set of 100 failure log files, which belonged to 71 different projects[4], from the set of all the failure logs. Then, we manually analyzed them to extract regular expressions belonging to different failure types, each of which is associated with one CI task. Fourth, we wrote a log analysis script that uses these regexes for classifying failures within their respective sub-types. Finally, we tested our log analyzer on the remaining set of failure logs to estimate our saturation rate. The rate was lower than 90% for both ML and Non-ML projects, so we repeated the previously-described process on a second set of 100 randomly-selected failure log files from 33 different projects to enrich our initial set of regexes. This allowed us to reach a saturation rate of 91.59% and 91.15% on ML and Non-ML job failure logs, respectively. In total, we used 2 sets each containing 100 log files to construct our failure taxonomy. For the error taxonomy, we repeated this same process by analyzing 2 sets of 100 error log files from 90 different projects, randomly selected from 18857 error logs, 14121 of which belonged to ML projects and 4736 of which belonged to Non-ML projects. This allowed us to reach a saturation rate of 95.05% and 96.16% on ML and Non-ML on job error logs, respectively.

This process has proven complex and time-consuming due to the variety of tools being used across Python projects, and their different logging conventions. For example, a test failure may be reported in different ways across different projects all using the same Pytest framework. It may be a line showing a test summary such as `=== 2 failed, 91 passed ===`, or instead, listing each failed test e.g: `FAIL Test1, FAIL Test2`, or a combination of both outputs or others. This output variability is also noted within other tools and other processes, such as linting and code coverage. To resolve these problems, our log analyzer had to rely on a high number of regexes to identify a large amount of different output-patterns, it relies on 110 regexes to analyze job failure logs, and 33 regexes to analyze job error logs. These regexes correspond to popular python tools that are used by both the ML and Non-ML projects we studied, making it possible to reuse the log analyzer for other Python-based CI-using software projects.

To better organize the different types of failures we identified via our log analyzer, we constructed our job failure taxonomy via the following process: 2 authors followed the card sorting [32] method of grouping similar failures and errors as identified by the regexes into multiple main and subgroups. Then, in the case of job failures, this hierarchy was simplified by merging most similar subgroups leaving only the ML-specific failure types at the 3rd level of the taxonomy tree, to help increase the ease of understanding and generalization of this taxonomy, after which 6 main and 18 subgroups remained. We constructed the error taxonomy using the same method, but we chose to only leave 4 main categories as they contained less internal variability within them in comparison to those of the failure taxonomy.

4 EVALUATION OF ANALYSIS TOOLS

Evaluation of TraVanalyzer: To evaluate `TraVanalyzer`'s AST generation, 2 authors manually evaluated 100 ASTs generated from 100 randomly selected .travis.yml files belonging to projects from our *Depth Corpus*, and found that the files were parsed correctly. This confirms the robustness of our tool regarding the generation of accurate ASTs. Regarding `TraVanalyzer`'s efficacy in correctly allowing us to determine the CI usage goals, 3 of the co-authors manually inspected Travis CI configuration files separately to categorize different tasks executed by the CI. Fleiss's kappa coefficient [33] was used to find inter-annotator agreement prior to discussion, and was on average 0.78 across the different categories, indicating substantial agreement. The disagreements of analysis were resolved by discussion. After evaluating the performance of `TraVanalyzer`

[4]no uniqueness criterionwas applied regarding projects or their categories during the process of log-selection as to not influence the randomness of the process

on the manually labeled data, we found an average Precision, Recall, and F1-score of 98.44%, 95.45%, and 96.92%, respectively, for identifying the different build, test, code analyzer, and deployment tasks configured within the CI configurations file(s). These results confirm the correctness of the proposed tool for our purposes.

Evaluation of the Log analyzer: We tested our CI Log analyzer on a set of randomly selected logs. Two authors manually labeled 100 errored job logs and 100 failed job logs, belonging to jobs from our Depth Corpus. These logs were not used to construct the log analyzer. Fleiss's kappa coefficient [33] was on average 0.73 across the 4 Job Error types, and 0.78 across the six main Job Failure types, indicating substantial agreement, and any disagreements were then resolved by discussion. Our log analyzer achieved an average Precision of 95.42%, average Recall of 92.84%, and F-1 score of 94.11% across the main Job Failure types. It also achieved an average Precision of 99.1%, an average Recall of 96.4%, and an F-1 score of 97.73 % across the Job Error types.

5 RESULTS

5.1 CI Adoption rates

Research Question 1: What is the adoption rate of CI among ML projects?

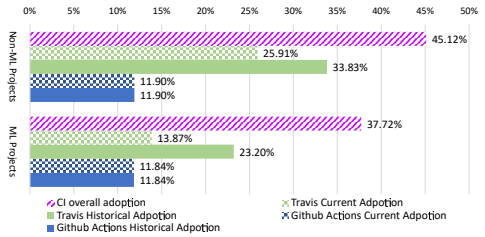

Figure 4: CI Tools Adoption rates (Excluding CI Tools with less than 5% adoption)

To determine the prevalence of CI within ML projects, we used the triangulation-based method detailed in Section 3.2.1, and applied it to our **Breadth Corpus**. This allowed us to estimate the adoption rates of the CI tools outlined in Section 3.2.1. The first step of our triangulation process was the File-System (FS) based process. Overall, the CI adoption rate through this method is estimated at 37.22% for all ML projects, and the CI adoption for Non-ML projects is estimated at 45.12%. Focusing on individual tools, the top 3 tools adopted by ML projects were Travis CI at 21.14%, GitHub Actions at 21.29%, and Circle CI as a distant third at 3.28%. For Non-ML projects, the top 3 tools adopted by ML projects were Travis CI at 33.98%, GitHub Actions at 13.62%, and AppVeyor as a distant third at 3.04%.

However, when applying the triangulation-based method, via querying the APIs of the top 2 tools per the FS-based method for both categories of projects, Travis CI and GitHub Actions, and then consolidating them with the FS-based findings, we found that there is a mismatch between the two data sources. Details about this mismatch and the definitions of *Historical Adoption* and *Current Adoption* we chose to resolve it are in Section 3.2.1. The current and historical adoption rates, illustrated within Figure 4, reflect that the popularity of Travis CI for open-source software [39] is also evident in ML projects and our comparison set of Non-ML projects, with GitHub Actions being a close contender in terms of current adoption for ML projects. This is surprising given the age of GitHub Actions, as support for CI was only added to it in public beta in August 2019 [15]. Overall the adoption of CI by ML projects is between 24.46% per the triangulation-based approach, and 37.22%, per the FS-based approach, which trail behind those of our comparison set of Non-ML projects, estimated at 38.84% per the triangulation-based approach, and 45.12% per the FS-based approach. The adoption rates of ML projects are also less than those of Open source projects in general, which is estimated at 40.27% by Hilton et al. [39] and more recently between 45% and 68% by Digital Ocean [3], which are consistent with our findings regarding CI adoption by our Non-ML comparison set.

5.2 CI Task Analysis

Research Question 2: What tasks does CI perform for ML projects?

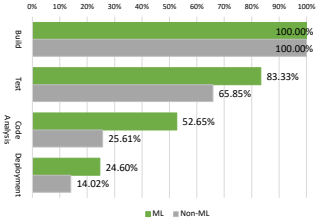

Figure 5: CI Task Adoption Percentage

We applied `TraVanalyzer` on the subset of projects from our **Depth Corpus** with the *current adoption* of Travis: 378 ML projects and 164 Non-ML projects in order to categorize the CI tasks of ML projects and compare them with those of Non-ML projects. We classified the tasks into four categories: build, test, code analysis, and deployment. The build category includes both build environment preparation and build configuration and execution commands. All the ML projects we analyzed include configuration and commands related to build environment preparation and execution in their Travis configuration files. However, only 83.33% of all of the ML projects we analyzed adopted testing in their CI process, and surprisingly, only 65.85% of Non-ML projects adopted testing, even though recommended testing practices are well established for these projects [18]. For code analysis, which includes static analysis, linting, and code coverage tools, overall adoption is lower than build and test adoption. Another surprising result is that while 52.65% of ML projects adopted code analysis in their CI workflow, only 25.61% of Non-ML projects adopted it. Concerning deployment, its overall adoption rate is only 24.6% by ML projects, even though automatic deployment is considered a key component of their workflows [4]. Meanwhile, only 14.02% of Non-ML projects adopted automatic deployment. In addition, as will be illustrated within the rest of this section, along with the higher adoption of the different functions of CI by ML projects, there is a higher diversity of tools being used by these projects to achieve those goals in comparison to Non-ML projects. Comparing these results to those found by Durieux et al. [30] when analysing Travis jobs of open source software in-general, building and testing are also the most common concerns of CI within OSS.

Build Tasks: Travis CI allows the definition of build environment in its configuration. For example, `language:python` prepares an environment for Python-based projects. Other properties can be set, such as the Python version, OS, etc. With `TraVanalyzer`, we extracted build configuration features. The 10 most frequently-used Tavis CI configurations for ML projects are: `language`, `python`, `dist`, `sudo`, `env`, `cache`, `include`, `os`, and `apt`. These are also the most popular keywords for Non-ML build configuration. Developers can also prepare their build environments by using bash commands to download dependencies, set environment variables, etc. Based on our approach detailed in Section 3.2.2, the 10 most frequently used external commands for preparing ML projects' build steps are: `pip`, `export`, `wget`, `apt-get`, `setup.py`, `conda`, `hash`, `cd`, `make`, and `curl`. The top 10 commands used for Non-ML projects are similar, but instead of `hash` and `curl` we find the `git` and `npm` commands.

Test Tasks: Testing confirms the correctness of the code before its integration. Since Travis CI does not provide built-in test configuration, test cases are generally executed by external tools and scripts invoked from the Travis CI configuration file. Based on our analysis, `pytest`, `activate`, external shell scripts, external Python scripts, `nosetests`, `py.test`, `coverage`, `tox`, `unittest` and, `green` are the 10 most frequently used tools and scripts used for ML projects' testing. Most projects invoke python unit test frameworks directly, but some projects invoke external scripts to execute their tests. No usage of ML-specific testing frameworks was noted. The same list was found for Non-ML projects, substituting `green` with the `make_test` command.

Code Analysis Tasks: Some ML projects use code analyzers for code quality and code style checking. These analyzers are generally configured through the Travis CI configuration file to ensure continuous code quality checking. `coveralls`, `codecov`, `flake8`, `coverage`, `pylint`, `bandit`, `mypy`, `autopep8`, `python-codacy-coverage`, and `black` are the 10 most frequently used code analysis tools and commands. Most of these are code coverage analyzers, but some are used for code style checking and other types of static analysis, such as `bandit`, a tool for checking security vulnerabilities of Python code. The first 5 tools were also among the most frequent Non-ML tools for code analysis, however,`ninja`,`pep8`,`pyflakes`,`codeclimate-test-reporter`,`luacheck` were the other 5 most-frequently-used coverage tools.

Deployment Tasks: Most CI services including Travis CI provide built-in support for deployment automation. Developers can also invoke external tools such as Docker [13] to automate deployment. Based on our analysis, `provider:pypi`, `docker`, `provider:pages` [23], `provider:script`, external shell script,`provider:releases` [23], `twine`, `provider: pages:git`, `doctr`, and `provider:aws` make up the top 10 providers and tools used for deployment. Meanwhile, the only deployment tool we found was beind used by Non-ML projects was `docker`

5.3 CI Problem Frequency and Taxonomy

> ***Research Question 3:*** How often and Why do CI builds break in ML projects?

Figure 6 shows the average percentages of the different types of build-outcomes per project within our **Depth Corpus** of projects. While ML projects seem to have a higher average rate of passing builds in comparison to Non-ML projects, it's important to also note the higher internal variability in terms of build-outcomes shown by the Non-ML projects within our **Depth Corpus**, when

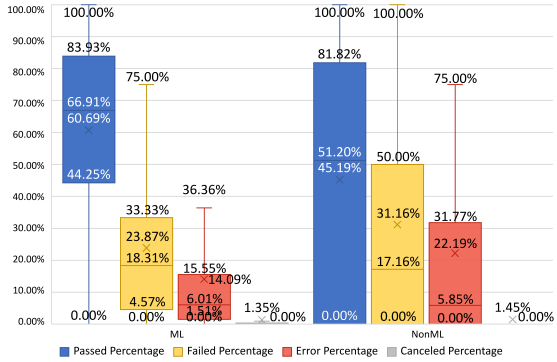

Figure 6: Build status average percentages

comparing their quartiles to those of ML projects. The results for Non-ML comparison set are surprising, especially when comparing our findings to Beller et al.'s [27] findings regarding open-source software, where an average of 82.4% of the builds for Java projects and an average of 72.7% of the builds for Ruby projects are passed, but it's important to note that our comparison set is composed of Python-based projects. Delving deeper into the different factors behind build breakage, we chose to analyze the logs corresponding to the failed jobs and errored jobs which compose the non-duplicate broken builds of the projects within our Depth Corpus, that we identified using the process detailed in Section 3.2.3. Gallaba et al. [35] showed that logs from Travis CI are an imperfect source of information since they can be incomplete, malformed, or not present within the Travis CI server. Indeed, we encountered problems obtaining the failure logs corresponding to the failed jobs we specified. For ML projects, this totaled 45590 jobs, 13.87% out of which were either empty or not found on Travis CI's server. From the 35965 obtained job failure logs, we achieved a 91.59% saturation rate of detecting at least one job failure type. For Non-ML projects, we succeeded at obtaining 7153 logs pertaining to the job failures we identified, on which we obtained a saturation rate of 91.15%, and only 11 logs were unobtainable. Moving on to error logs, We also attempted to download 15748 of the errored jobs selected from the jobs of ML projects. 10.33% of them were empty or not found on the Travis server. We achieved a 95.05% saturation rate of detecting the error type of the 14121 job error logs we obtained. For Non-ML projects, we attempted to download 4737 error logs only 1 of which were empty or not found, and we achieved a saturation rate of 96.16% on classifying the error types.

Description of Error Taxonomy: Moving on to the results we obtained, specifically the error taxonomy, the categories of the job errors which we determined are:

Script Error: in 306 jobs of ML projects and 0 jobs of Non-ML projects, constituting respectively 2.17% and 0%. They contain one or more errors within the shell script being executed. For example, an error occurs during copying or deleting a specific item or trying to execute a command that's not available.

Dependency Install problem: in 9989 jobs of ML projects and 3369 jobs of Non-ML projects, constituting respectively 70.74% and 73.96%. They contain problems directly related to the installation of dependencies. For example, if the package manager does not find a package, or there's a problem cloning a git repository required by the project.

Travis CI Error: in 2728 jobs of ML projects and 1157 jobs of Non-ML projects, constituting respectively 19.32% and 25.40%. They contain errors specific to the CI environment. In the case of Travis CI, logs exceeding the maximum size are an example of such errors.
Install phase misuse: in 2732 jobs of ML projects and 864 jobs of Non-ML projects, constituting respectively 19.35% and 18.97%. They contain the usage of the install phases for purposes other than installing dependencies. For example, running Testing, Code analysis, Deployment, or other processes within this phase.

Interpretation of Error Taxonomy results: It is evident that Dependency Install problems are the most common type of problems within the install phase, since they are detected within 70.74% and 73.96% of ML and Non-ML job errors. In fact, all the job error categories we identified, except for the install phase misuse category, are linked directly or indirectly to issues that can occur during dependency installation. This is unsurprising as the main goal of the `install` phase of a Travis job is to install the dependencies needed in order to run the configured scripts for the `script` phase correctly. It's important to note that an errored job can be linked to one or more of the issues within the taxonomy, hence a job can belong to one or more error categories. The frequency of job errors confirms that some of the approaches that developers use to install their dependencies, such as cloning from git or installing a specific version from package managers, are not 100% reliable. A git repository may change location or be removed, and a specific version of a tool may be removed from the package manager. Similar to Durieux et al.'s work [31] on traditional software, we found git cloning for dependency installation is a problem-causing practice, with 1065 jobs linked to ML projects failing due to a git-related error. While CI Tool errors may be due to limitations with Travis itself, the Install phase misuse being present in 19.35% of ML errored jobs and 18.97% of Non-ML errored jobs is concerning, as it possibly indicates a disregard of developers for Travis conventions, which can make diagnosing and resolving subsequent issues harder. These findings are similar to those of Gallaba et al.[36] concerning the prevalence of misuse of Travis files in open-source software.

Overall, the different job error categories occur with similar percentages in our ML set of projects as well as our Non-ML comparison set. It's important to note however that script errors as well as install phase misuses are present within a larger percentage of ML job errors than Non-ML ones, but Non-ML job errors are more likely to be related to Dependency Install problems and Travis CI related errors than ML job errors. Our findings align with those of Pinto et al. [46] concerning CI build breakages for traditional OSS. Indeed, they also found that issues related to dependency management were common reasons behind build errors. Furthermore, other works such as those of Sulír & Porubän [50], Tufano et al. [51] and Seo at al. [49] who respectively estimated that dependency-related issues accounted for 39% , 58% , and 65% of OSS build breakages.

Description of Failure Taxonomy: Following the `install` phase, the `script` phase which involves multiple CI processes is executed. A failed job may belong to 1 or more of the main categories or sub-categories. The main types of failures are:
Script Failure: This failure is the result of an error during the execution of the shell script or the python script as it attempts to execute tasks related to preparing the environment for CI processes and execute each one of them. This failure's sub-types are:

⇨*Dependency Install Problems*: this type of failure is encountered when Travis encounters a problem while trying to acquire and install a certain dependency. This is a sign of misuse of the script phase, as the best practices recommend that all dependencies be installed during the install phase of a Travis job [36].

⇨*Resource Not Found*: If the script tries to access a module, file, or program during its execution (outside of the other CI processes) and is unable to find it, this type of error occurs.

⇨*Other Commands Failure*: A general command failure resulting in the failure of the execution of a command, which in turn may stop the execution of the entire script or the execution of a specific CI task.

Test Failure: One or more tests ran correctly, but identified problems within the functional code being tested. The functional code needs to be modified to resolve this problem. Its sub-types are:

⇨*Assertion Exception*: An exception occurs when an assertion fails within a test execution. This exception occurs when the code fails to establish the functional requirements specified by the developer and is indicative of good test-writing practices being followed. A sub-type of this exception is the **ML-specific Assertion**, which indicates that an exception specific to the context of ML projects has occurred, such as the following example where the model prediction result does not meet the accuracy threshold.

```
Example: FAIL: test_recalculate_user (tests.als_test.
↪ ALSTest)
AssertionError: 1.0 != 0.0 within 0.0001 delta
```

⇨*Other Exception*: This type of exception occurs when an exception unforeseen by the test occurs. It indicates that the test did not account for this specific type of failure, and thus is not following the best practices of test-writing [18].

Test Error: One or more tests did not run correctly, due to problems within the test build up or tear down, or other processes related to preparing the environment to execute a test. Its sub-types are:

⇨*Error During Test Collection*: This indicates an error occurred while the testing framework was attempting to collect the tests across the different test-script files. This is usually due to an error in the tests which prevents them from being loaded into Pytest, such as missing modules or indentation errors within the test files.

⇨*Test Fixture Error*: An error that occurs during the execution of a pre-test or a post-test method, also known as fixtures, due to a programmatic problem during the execution of these methods such as a resource not being found. An ML-specific sub-type of this exception is the **CUDA Problems**, which indicates an exception related to the CUDA ML framework, and the other one is **ML Module Not found**, which specifies that the test fixture was unable to import a module generally associated with ML projects such as Tensorflow.

```
Example:
Could not load dynamic library 'libcuda.so.1'; dlerror:
↪ libcuda.so.1: cannot open shared object file: No such
↪ file or directory
```

⇨*Test Environment Problem*: This indicates a problem linked to the testing environment was found by the testing framework. For example, it occurs when certain environment variables or dependencies that were expected by the testing framework were not found, or the testing framework was unable to find or create the testing environment.

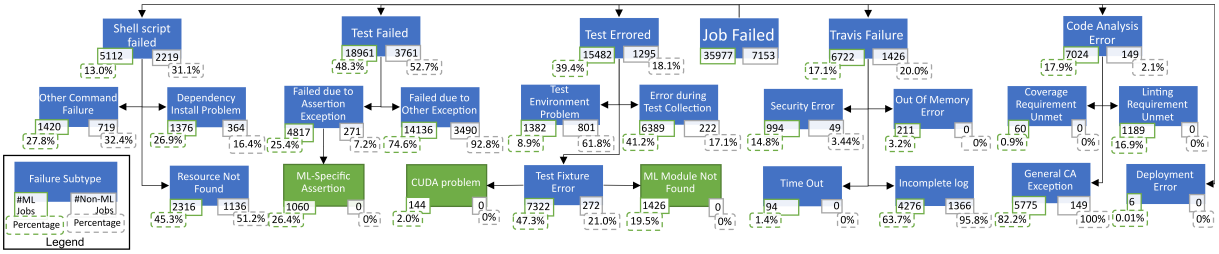

Figure 7: Job Fail Taxonomy. We show the count of Failed ML and Non-ML jobs of each sub-type in each block, along with the relative percentage of failed jobs of each sub-type in relation to its direct super-type

```
Example:
$tox
ERROR: unknown environment 'vulture'
```

Code Analysis Error: A failure during the process of Code Analysis or caused by its result. For example, if an unexpected exception occurs during the code coverage phase, or if severe code formatting issues are reported, this type of error occurs. Its sub-types are:

⇨*Code Coverage*: This type of failure occurs when the testing coverage does not meet the minimum criteria set by the developer.

```
Example:
FAIL Required test coverage of 100% not reached
```

⇨*Linting*: This type of failure occurs when the coding practices of the software do not conform with the conventions set by the developer.

⇨*Other Code Analysis Fail*: this occurs when an exception is thrown during a code analysis process. In general, it's due to an unexpected exception during the process of Code Coverage or Linting.

Deployment Error: This error occurs when a problem is encountered when the CI process attempts to deploy artifacts to a certain destination, for example, due to connection or authentication issues. The reason behind the low number of deployment-related job failures is the fact that the failure of the deployment phase built into Travis does not affect the build outcome status. Hence even if a deployment fails, if the job has not encountered any problems beforehand, it will still be considered passed.

Travis Failure: This error occurs due to problems related to the CI tool being used within a project. For instance, in the context of Travis, this could be a security error related to certificates within the Travis container, the container running out of memory, etc. Its sub-types are:

⇨*Security Error*: This error occurs when the Travis instance faces a problem related to the verification of the signatures of certain resources, such as a package manager.

⇨*Out of Memory Error*: This error occurs when the Travis CI instance runs out of memory and can no longer load needed resources into its memory.

⇨*Incomplete Log*: This type of error occurs when the Travis Log is unexpectedly incomplete (for example, stopping in the middle of an output line). This is due to a known issue with Travis CI [2].

Interpretation of Failure Taxonomy results: While a job may fail for multiple reasons, it's clear that most of the job failures of ML projects are linked to testing. Focusing on test failures, 4817 jobs of ML projects, 25.4% of the reported failures, were a result of an assertion exception, thus 74.6% of the reported test failures did not follow the best practices of testing for the Python language [18]. Furthermore, only 26.4% of the test failures of ML projects which followed these practices were ML-specific, such as failing to meet accuracy criteria, indicating that the majority of the tests performed were not necessarily ML-specific. Regarding test errors, the second most common failure reason, 88.5% of them were due to programmatic problems related to test collection or test fixtures, which are problems linked to the coding of the tests themselves and to their frameworks' configuration. Comparing these results with those of Non-ML projects, it's clear that ML projects have different frequencies of job failure types. One surprising results is that ML projects are better at following testing practices than Non-ML projects, since only 7.2% of the latter's failures followed the recommended practices, especially considering their relative novelty [37]. Focusing on build failures in OSS in general, Pinto et al. [46] found similar build failure reasons as we did, ranging from inadequate testing to missing edge cases, when analyzing build breakages within OSS via interviews. Vassallo's work [53] illustrates a similar picture for both open source and Industrial Java software in the context of build failures. Beller et al.'s work [27] illustrated that most build failures are due to failed tests, where 70.89% of Java build failures and 67.13% of Ruby build failures were due to failed Tests. In comparison, 48.3% ML build failures were due to Test failures. The similarities between OSS job failures, our comparison set of Non-ML job failures, and ML build failures confirm that the same problems that affect CI in OSS also affect CI in ML projects, with some variance in frequency. For example, static analysis failures were present in 17.9% of ML projects, but only affected 4.2% of builds of OSS in Vasallo's results [53], and 2.1% of the job failures of our Non-ML set.

None of the test failures or test errors we detected through our semi-automatic log analysis or the manual methods we used in constructing it revealed the usage of ML-specific testing frameworks or tools, even though a few of these tools have been introduced, such as that by Karlaš et al. [40], or Amazon [5], or the usage of recommended practices [40], such as changing the test set with different builds to avoid the overfitting of models. This indicates that even though ML-specific tools and practices were introduced to the software engineering process, their adoption is still lagging.

6 IMPLICATIONS

For Researchers. First, while CI in ML projects has not received much attention from the research community, it's adopted by up to 37.22% of ML projects, showcasing its importance as a subject of study. We provide researchers with a set of CI-using ML projects [26], in order to guide their work in developing and adjusting CI servers and tools for ML projects. Second, Travis CI is the best source of information regarding CI practices of ML projects, and based on our study, there is no evidence of the widespread adoption of CI platforms specifically designed for ML projects. Thus,

TraVanalyzer [26] is a great tool to further investigate CI practices in ML projects and can be easily extended to support other types of Travis-using projects, especially since we were able to successfully use it in our set of Non-ML projects. Third, we found some of the most frequent CI problems of ML projects were related to test failures, test errors, and code analysis failures. Our Failure taxonomy and corresponding log analyzer can help with the automatic detection and debugging of these problems and guide the development of Travis CI configuration repair tools. **For ML Developers**. While adopting a CI tool is a step in the right direction, simply building the software with it is insufficient. Testing is a basic tenant of Continuous integration [1], and code analysis procedures, especially code coverage, are highly recommended. ML projects' developers should invest in implementing these processes in their CI workflows and adapting them to the context of ML projects and that of their project. For example, during our manual log-analysis step for the construction of our CI Log analyzer, we found that some projects are executing their tests on the same test set in every build, increasing their risk of ML model over-fitting. To avoid this problem, ML projects are recommended to vary their test sets frequently [40]. Another example is that some ML projects had job failures due to a restrictive 100% code coverage requirement, which can be harder to achieve as the code-base grows. Relaxing this requirement or monitoring CI status frequently can help minimize unnecessary disruptions.

7 RELATED WORK

Hilton et al.'s work [39] contains one of the most exhaustive CI adoption estimates for open-source projects on GitHub, as it considered a larger amount of open-source CI tools and projects than many other works examining this issue [27, 38, 52]. When it comes to CI goals, Durieux et al.'s [30] work is a good indicator of the tasks CI in OSS performs, It classified the tasks being performed by Travis CI into categories ranging from testing to communication. Moving on to CI problems, Gallaba & McIntosh [36] did an excellent job at analyzing a set of .travis.yml files, extracting configuration anti-patterns from them Focusing more on Travis in action, Beller et al.'s [27] work analyzes Travis CI jobs with problems resultant from testing and presents important information about their frequency per language. Focusing also on the testing aspect of CI, Karlaš et al. [40] outline a few problems concerning CI tool-support for ML projects and discuss some of the problems that using the CI tools' traditional testing practices may engender in regards to an ML model's accuracy.

What distinguishes our work from these is scope: it is one of the few that specifically focus on CI within ML projects in practice, as well as analyzing CI in Non-ML in Python-based projects as a baseline for contextualization. Furthermore, to estimate CI adoption within ML projects, we apply a triangulation-based approach on more recent data, rather than the API-only approach that Hilton et al. [39] employ, which may give false positives in case a project is no longer using a CI service. Concerning the goals of CI, Durieux et al's work's [30] granularity with its focus on jobs may be skewing the results in the direction of the CI tasks being performed by more active projects that generate more builds, and thus more jobs, or projects which are configured to run multiple jobs in different environments, thus artificially increasing the count of the tasks they are trying to perform. Our work attempts to investigate the multitude of CI problems ML projects experience in practice, ranging from dependency installation problems to deployment errors.

8 THREATS TO VALIDITY

The major threat to this study's internal validity is the correctness of the classification of the ML projects dataset. To reduce this threat, we have manually inspected the breadth corpus to ensure that the studied projects are actual software projects, not study guides or toy projects, and that they are in fact using ML. Moreover, the Travis CI AST-based task analyzer and CI log analyzer developed for automatic analysis can have internal threats to the correct analysis of the tasks, failure types, and error types. To minimize such threats, we also evaluated the correctness of these tools with manually labeled data as described in Section 4. The major threat to the study's external validity is that we analyzed open-source ML projects available on GitHub and mainly focused on Python-based projects. So, the CI adoption by ML projects findings can be different for closed-source projects and projects developed in other programming languages. However, our findings are still significant, since our dataset included large-scale ML projects such as `tensorflow/tensor2tensor`, which was developed and open-sourced by organizations like Google Brain. We also focused on Python, the most popular programming language of ML projects.

9 CONCLUSION

In this work, we have shown that CI has been less widely adopted among ML projects in comparison to Non-ML projects. We also analyzed their different CI tasks, and extracted knowledge about common problems of CI in ML projects. To the best of our knowledge, this is the first work that has analyzed ML projects' CI usage, practices, and issues. Furthermore, we have also contextualized these results by comparing them with similar Non-ML projects, and summarized useful findings for researchers and ML developers to identify possible issues and improvement scopes for CI. For future works, we plan to utilize the tools we developed for this study, to develop a tool-chain for automatic fault localization and repair of CI build breakages in ML projects, and to contribute this tool-chain to the open-source community. The replication package is accessible at [26].

ACKNOWLEDGMENTS

The UofM-Dearborn authors are supported in part by UofM-Dearborn Research Support and NSF Award NSF-2152819.

REFERENCES

[1] 2000. Continuous Integration. https://martinfowler.com/articles/continuousIntegration.html Accessed: 2021-08-25.
[2] 2017. After failure output gets truncated unless sleep is used - Issue #6018 - travis-ci/travis-ci. https://github.com/travis-ci/travis-ci/issues/6018
[3] 2018. DigitalOcean Blog. https://www.digitalocean.com/blog/currents-march-2018/
[4] 2021. https://neptune.ai/blog/machine-learning-model-management
[5] 2021. Amazon SageMaker – Machine Learning – Amazon Web Services. https://aws.amazon.com/sagemaker/ accessed 08-28-2021.
[6] 2021. API Developer Documentation - Travis CI. https://developer.travis-ci.com/ Accessed: 2021-08-20.
[7] 2021. Buildbot - The Continuous Integration Framework. https://www.buildbot.net/ accessed 08-31-2021.
[8] 2021. CircleCI-Continuous Integration and Delivery. https://circleci.com/ accessed 08-31-2021.
[9] 2021. Cloud Build Serverless CI/CD Platform. https://cloud.google.com/build accessed 08-31-2021.
[10] 2021. Continuous Integration and Deployment service for Windows, Linux and macOS. https://www.appveyor.com/ accessed 08-31-2021.

[11] 2021. Continuous Integration with Visual Studio Team Services. https://microsoft.github.io/PartsUnlimitedMRP/cicd/200.3x-CICD-CI.html accessed 08-31-2021.

[12] 2021. DocConverter: A Java library to parse yaml file. https://github.com/assimbly/docconverter Accessed: 2021-08-20.

[13] 2021. Docker: open source containerization platform. https://www.docker.com/ accessed 08-31-2021.

[14] 2021. ease.ml\ci. http://ease.ml/ci accessed 08-31-2021.

[15] 2021. GitHub Actions Documentation. https://docs.github.com/en/actions accessed 08-31-2021.

[16] 2021. GitHub REST API. https://docs.github.com/en/rest Accessed: 2021-08-27.

[17] 2021. GitLab CI/CD. https://docs.gitlab.com/ee/ci/ accessed 08-31-2021.

[18] 2021. How to use unittest-based tests with pytest — pytest documentation. https://doc.pytest.org/en/latest/how-to/unittest.html Accessed: 2021-08-27.

[19] 2021. Introduction to Kubeflow. https://www.kubeflow.org/docs/about/kubeflow/ accessed 08-28-2021.

[20] 2021. Jenkins – an open source automation server which enables developers around the world to reliably build, test, and deploy their software. https://www.jenkins.io/https:/www.jenkins.io/ accessed 08-31-2021.

[21] 2021. PyTravisCI documentation. https://pytravisci.readthedocs.io/en/latest/usage/ Accessed: 2021-08-23.

[22] 2021. Travis CI | Test and Deploy With Confidence. https://www.travis-ci.com/ accessed 08-31-2021.

[23] 2021. Travis CI Documentation. https://docs.travis-ci.com/user/deployment/ accessed 08-31-2021.

[24] 2021. Travis CI Documentation - Using YAML as a build configuration language. https://docs.travis-ci.com/user/build-config-yaml/ accessed 08-31-2021.

[25] 2021. What is AWS CodeBuild? - AWS CodeBuild. https://docs.aws.amazon.com/codebuild/latest/userguide/welcome.html accessed 08-31-2021.

[26] 2022. Replication package for this work. https://figshare.com/s/13f6afeb3fbc668cfbe4

[27] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 356–367. https://doi.org/10.1109/MSR.2017.62

[28] G. Booch. 1991. *Object Oriented Design with Applications*. Benjamin/Cummings Publishing Company. https://books.google.com/books?id=w5VQAAAAMAAJ

[29] Nancy Carter, Denise Bryant-Lukosius, Alba DiCenso, Jennifer Blythe, and Alan J. Neville. 2014. The Use of Triangulation in Qualitative Research. *Oncology Nursing Forum* 41, 5 (Sep 2014), 545–547. https://doi.org/10.1188/14.ONF.545-547

[30] Thomas Durieux, Rui Abreu, Martin Monperrus, Tegawendé F. Bissyandé, and Luís Cruz. 2019. An Analysis of 35+ Million Jobs of Travis CI. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (Sep 2019), 291–295. https://doi.org/10.1109/icsme.2019.00044 arXiv: 1904.09416.

[31] Thomas Durieux, Claire Le Goues, Michael Hilton, and Rui Abreu. 2020. Empirical Study of Restarted and Flaky Builds on Travis CI. In *Proceedings of the 17th International Conference on Mining Software Repositories*. ACM, 254–264. https://doi.org/10.1145/3379597.3387460

[32] Sally Fincher and Josh Tenenberg. 2005. Making sense of card sorting data. *Expert Systems* 22, 3 (Jul 2005), 89–93. https://doi.org/10.1111/j.1468-0394.2005.00299.x

[33] Joseph L. Fleiss and Jacob Cohen. 1973. The Equivalence of Weighted Kappa and the Intraclass Correlation Coefficient as Measures of Reliability. *Educational and Psychological Measurement* 33, 3 (1973), 613–619. https://doi.org/10.1177/001316447303300309 arXiv:https://doi.org/10.1177/001316447303300309

[34] Lloyd D Fosdick and Leon J Osterweil. 1976. Data flow analysis in software reliability. *ACM Computing Surveys (CSUR)* 8, 3 (1976), 305–330.

[35] Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. 2018. Noise and heterogeneity in historical build data: an empirical study of Travis CI. ACM, 87–97. https://doi.org/10.1145/3238147.3238171

[36] Keheliya Gallaba and Shane McIntosh. 2020. Use and Misuse of Continuous Integration Features: An Empirical Study of Projects That (Mis)Use Travis CI. *IEEE Transactions on Software Engineering* 46, 1 (Jan 2020), 33–50. https://doi.org/10.1109/TSE.2018.2838131

[37] Danielle Gonzalez, Tom Zimmermann, and Nachi Nagappan. 2020. The State of the ML-universe: 10 Years of Artificial Intelligence &; Machine Learning Software Development on GitHub. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*.

[38] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. 2015. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 358–368. https://doi.org/10.1109/ICSE.2015.55

[39] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 426–437. https://doi.org/10.1145/2970276.2970358

[40] Bojan Karlaš, Matteo Interlandi, Cedric Renggli, Wentao Wu, Ce Zhang, Deepak Mukunthu Iyappan Babu, Jordan Edwards, Chris Lauren, Andy Xu, and Markus Weimer. 2020. Building Continuous Integration Services for Machine Learning. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2407–2415. https://doi.org/10.1145/3394486.3403290

[41] Shahedul Huq Khandkar. 2009. Open coding. *University of Calgary* 23 (2009), 2009.

[42] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. 2019. A Survey of DevOps Concepts and Challenges. *ACM Comput. Surv.* 52, 6, Article 127 (Nov. 2019), 35 pages. https://doi.org/10.1145/3359981

[43] Lucy Ellen Lwakatare, Ivica Crnkovic, and Jan Bosch. 2020. DevOps for AI – Challenges in Development of AI-enabled Applications. In *2020 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. IEEE, 1–6. https://doi.org/10.23919/SoftCOM50211.2020.9238323

[44] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. 2005. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*. 1–5.

[45] Michael Olan. 2003. Unit testing: Test early, test often. *Journal of Computing Sciences in Colleges - JCSC* 19 (Jan 2003).

[46] Gustavo Pinto, Fernando Castor, Rodrigo Bonifacio, and Marcel Rebouças. 2018. Work practices and challenges in continuous integration: A survey with Travis CI users: Work practices and challenges: A survey with Travis CI users. *Software: Practice and Experience* 48, 12 (Dec 2018), 2223–2236. https://doi.org/10.1002/spe.2637

[47] Quantilus. 2020. Why is Machine Learning Important and How will it Impact Business? https://quantilus.com/why-is-machine-learning-important-and-how-will-it-impact-business/ Accessed: 2021-08-24.

[48] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 345–355. https://doi.org/10.1109/MSR.2017.54

[49] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 724–734. https://doi.org/10.1145/2568225.2568255

[50] Matúš Sulír and Jaroslav Porubän. 2016. A quantitative study of Java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, 17–25. https://doi.org/10.1145/3001878.3001882

[51] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and back again: Can you compile that snapshot?: There and Back Again: Can you Compile that Snapshot? *Journal of Software: Evolution and Process* 29, 4 (Apr 2017), e1838. https://doi.org/10.1002/smr.1838

[52] Bogdan Vasilescu, Stef van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark G.J. van den Brand. 2014. Continuous Integration in a Social-Coding World: Empirical Evidence from GitHub. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 401–405. https://doi.org/10.1109/ICSME.2014.62

[53] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. 2017. A Tale of CI Build Failures: An Open Source and a Financial Organization Perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 183–193. https://doi.org/10.1109/ICSME.2017.67

[54] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. 2019. A conceptual replication of continuous integration pain points in the context of Travis CI. ACM, 647–658. https://doi.org/10.1145/3338906.3338922