Optimizing Continuous Development By Detecting and Preventing Unnecessary Content Generation

Talank Baral[1], Shanto Rahman[2], Bala Naren Chanumolu[1], Başak Balcı[3], Tuna Tuncer[3], August Shi[2], Wing Lam[1]
[1] George Mason University, USA
{tbaral,bchanumo,winglam}@gmu.edu
[2] The University of Texas at Austin, USA
{shanto.rahman,august}@utexas.edu
[3] Technical University of Munich, Germany
{basak.balci,tuna.tuncer}@tum.de

***Abstract*—Continuous development (CD) helps developers quickly release and update their software. To enact CD, developers customize their CD builds to perform several tasks, including compiling, testing, static analysis checks, etc. However, as developers add more tasks to their builds, the builds take longer to run, therefore slowing down the entire CD process. Furthermore, developers may unknowingly include tasks into their builds whose results are not used (e.g., generating coverage files that are never read or uploaded anywhere), therefore wasting build runtime doing unnecessary tasks.**

We propose OptCD, a technique to dynamically detect unnecessary work within CD builds. Our intuition is that unnecessary work can be identified by the generation of files that are not used by any other task within the build. OptCD runs alongside a CD build, tracking the generated files during the build and which files are read/written. Files that are written to but are never read from are unnecessary content from a build. Based on the names of the unnecessary files, OptCD then maps the files to the specific build tasks responsible for generating or writing to those files. Finally, OptCD leverages ChatGPT to suggest changing the build configuration to disable generating these unnecessary files. Our evaluation of OptCD on 22 open-source projects finds that 95.6% of projects generate at least one unused directory, a directory whose contents are all unnecessarily generated. OptCD identifies the correct task that generates 92.0% of the unused directories. Further, OptCD can produce a patch for the CD configuration file to prevent generating 72.0% of the unused directories. Using the patches, we reduce the runtime by 7.0% on average for the projects we studied. We submitted 26 pull requests for the unused directories that we could disable. Developers have accepted 12 of them, with five rejected, and nine still pending.

I. Introduction

Developers practice continuous development (CD) to release their software quickly and make continuous, ongoing improvements after the software is released. The process of CD typically involves a build that is triggered when developers make changes to their code and commit/push their changes. Within this build, various other processes are performed, such as (1) continuous integration [1]–[5], which checks whether the changes can be integrated into the code base without breaking existing functionality, and (2) continuous deployment, which releases the newly changed software to end users. Developer can customize CD builds to release the changes and perform various tasks, commonly including compiling, testing, and analyzing the code. If any of these tasks fail, then developers need to debug and fix their code changes before they can make any additional changes. There are an abundance of CD-related services available to developers, such as GitHub Actions [6], Jenkins [7], or Travis CI [8]. These services help developers by providing temporary virtual machines to compile, test, and analyze code.

CD can be time-consuming, given that CD builds occur on every change, and developers are frequently making changes to their code base [9]. Furthermore, a developer may configure the CD build to perform many other tasks beyond the basics of checking whether the code compiles or that tests pass, such as collecting code coverage [10], building documentation [11], or static analysis [12]. Unfortunately, developers may end up not using any of the results of these additional tasks they configured for the CD build. For example, if developers configured the CD build to collect code coverage but then do not upload those results elsewhere before the virtual machines are destroyed, then the work that went into collecting the results is unnecessary and a waste of CD build time.

We propose *Optimizing Continuous Development (OptCD)* to automatically identify the generation of unnecessary content during CD builds. Tasks in CD builds will often create or modify files. The intuition of OptCD is that if these files are never read or used after they were created or modified, then the work that went into creating or modifying the files is unnecessary. As such, if we can identify these unused files and map them back to the CD-build tasks that resulted in those files, then developers can disable or reconfigure these tasks, therefore saving overall CD build time.

For a given CD-configured project, OptCD first modifies the CD configuration to log the file reads/writes that occur during a CD build. After the CD build finishes, OptCD then analyzes the generated log, identifying unused files as those where there are no read operations to the file after the final write operation. OptCD then clusters the unused files together into unused directories, namely directories that contain only unused files. The intuition is that build tasks often do not generate individual files but rather a set of files all gathered within some common directory (e.g., code coverage reports). OptCD systematically searches through the available tasks and checks whether disabling the task also disables the generation of the unused directories. To speed up this search process, OptCD leverages three strategies to prioritize the order in which to search through the tasks based on the unused directory name: (1) Information Retrieval, (2) ChatGPT, and (3) Log Search. The Information Retrieval strategy uses TF-IDF [13] to prioritize the build tasks based on the task names that are most related to the unused directory name. The ChatGPT strategy relies on asking ChatGPT [14] to prioritize the list of build tasks based on which ones it thinks are most likely responsible for generating a given unused directory. The Log Search strategy parses the generated log of file reads/writes as well as the default log of build activity, matching the timestamps of the creation of unused files to the timestamps of build tasks. Whichever task has a timestamp that overlaps with the final creation/modification timestamp of the unused files is ranked the highest. In all three strategies, if none of the ranked list of tasks are correct (i.e., disabling the task does not disable the generation of the unused files), OptCD proceeds to search on the remaining unranked tasks. Once the exact task is identified, a developer may disable this task on their CD and local builds by removing this task from the project configuration file. If a developer wishes to disable the task only on their CD builds, OptCD can also leverage ChatGPT to help generate a patch for the CD build configuration file to stop the generation of files within the unused directories.

We evaluate OptCD on 22 popular, open-source, Maven projects from GitHub that use GitHub Actions for their CD. We use OptCD to reconfigure their CD builds to track the unused directories and later to identify the CD build tasks, specifically the Maven plugins that generated unused directories during the build. We find that 95.6% of projects in our evaluation generate at least one unused directory, with an average of 4.0 unused directories per project. OptCD's search strategies are able to identify the correct plugin responsible for generating 92.0% of the unused directories. OptCD can also use ChatGPT to suggest a patch to the CD build configuration to disable generating 72.0% of unused directories. Unfortunately, not all unused directories can be easily disabled, e.g., some directories are automatically generated by build systems, such as Maven, to contain diagnostic information that is often unused. When we use OptCD's patches for CD build files, we reduce the average time to run the corresponding build command by 7.0%. We submitted 26 pull requests for disabling unused directories. Developers have accepted 12 pull requests, with five rejected, and nine pending.

This paper makes the following main contributions:

- We propose dynamically tracking file reads/writes during CD builds to identify the generation of unnecessary content that can be disabled to speed up CD build runtime.
- We present OptCD [15] to identify the Maven build tasks doing unnecessary work during a Maven build when a GitHub Actions build is triggered.
- We evaluate OptCD on 22 Maven projects that use GitHub Actions for their CD. We find that 95.6% of projects generate at least one unused directory. OptCD can identify a change to the CD configuration file to disable the generation of 72.0% of identified unused directories. We sent 26 pull requests to developers with these changes – 12 are accepted, nine are pending, and five are rejected.

II. Background

Continuous development (CD) is the process of continuously developing and releasing code, which involves checks and other processes to ensure the changes are safe to integrate and to deploy [1], [2], [4], [5]. Developers can configure the CD process for their projects, such that changes they make to the code can trigger a CD build that performs a number of tasks to check for the correctness of those changes. Commonly what happens is that these changes are automatically pulled onto a remote server that tries to compile, test, and perform any additional tasks on the code. Depending on the tasks, the CD build may fail, which in turn prevents developers from integrating their changes until they inspect and fix the problems in those changes. There are an abundance of CD-related systems and services available to developers, such as GitHub Actions [6], Jenkins [7], or Travis CI [8].

In this work, we specifically focus on GitHub Actions, given its easy integration with any open-source project on GitHub. To configure a GitHub Actions build, a developer can create any number of `.yml` files that each define a *workflow* that triggers upon an event, such as a developer pushing changes to the GitHub repository or receiving a pull request [16]. A developer can define any number of *jobs* within the workflow, and each job runs inside its own virtual machine or container within the GitHub Actions servers, isolated from any other jobs. All jobs operate on the same changed code in the repository, but they may run different tasks. A job is defined by a number of *steps*, where a step is a set of script commands. After all steps in a job finish execution, the outcome of the job is presented to developers and the virtual machine where the job was run is deleted, along with any files generated during the job (unless they are explicitly uploaded elsewhere).

Figure 1 shows an example workflow `.yml` file for the project JSQLParser [17]. We can see that the workflow defines jobs (identified as `build` under the `jobs` label), where individual steps (listed under the `steps` label) are run on an Ubuntu system, and each job runs with a different version of Java. These individual steps can use predefined actions that GitHub Actions provides (e.g., Line 15) or bash commands (e.g., Line 17).

Aside from setting up CD, developers often also set up a build system for their project, such as Maven [18] for Java projects. Maven can automatically download necessary dependencies, compile Java code, run tests, or perform any number of other configurable tasks. In Maven, a developer defines Maven *plugins* that perform these build tasks within a Maven build. There are an abundance of existing plugins that a developer can include, such as JaCoCo [10] for collecting code coverage. Even compiling and testing are defined and configurable as plugins, namely the Compiler [19] and Surefire [20] plugins, respectively. A developer may configure their

```
name: Java CI with Maven
 on: [push, ...]
 jobs:
   build:
     runs-on: ubuntu-latest
     strategy:
       matrix:
         java: [8, 11]
     name: Java ${{ matrix.java }} building ...
     steps:
+    - uses: actions/setup-python@v2
+      with: python-version: '3.10'
+    - name: Install dependencies
+    - run: ... # install pandas, numpy, inotify
     - uses: actions/checkout@v3
+    - run: ... # run script to start inotifywait
+    - run: touch starting_build_BuildwithMaven_29
+    - run: rm starting_build_BuildwithMaven_29
     ... # steps to setup java
     - name: Build with Maven
       run: mvn -B package -file pom.xml
+    - name: Pushes results to another repository
+      run: ... # push inotify logs for analysis
```

Fig. 1. Example of a GitHub Actions workflow file. Highlighted lines are modifications OptCD includes to enable logging of file operations.

`.yml` workflow file to include the Maven build commands they want to run in each step of a job. Figure 1 shows how the "Build with Maven" step runs the command `mvn -B package -file pom.xml`, which compiles the code, runs tests, and finally packages the code. We explain the added highlighted lines of code in Section III-A.

When that Maven command runs, the Surefire plugin runs tests and creates test logs in a directory called `surefire-reports/`. However, this Maven command step is the final step in this job, and this step does not use the generated files, e.g., by uploading the logs elsewhere. Once the job finishes, these files are deleted so there is no point for the Surefire plugin to generate them.

One way to prevent generating unused files is to configure the Surefire plugin to not generate them, which can be done by adding the `-DdisableXmlReport` argument to the Maven command. When we modify the workflow file accordingly, we can prevent generating the entire `surefire-reports/` directory while also reducing the time for running the Maven command by 14.2%. We submitted a pull request to the developers with this workflow file change, and they accepted our pull request.

III. OptCD

We present OptCD, a technique for identifying unnecessary work done during a CD build by tracking unused files. OptCD identifies the build tasks, namely the Maven build plugins, that generate unused files. If these plugins do not provide useful work, then the developer may choose to remove the plugins from the project configuration file and consequently, from their local and CD build. Further, given the unused files and corresponding plugins, OptCD also proposes changes to the CD build configuration file to avoid generating unused files in only CD builds. OptCD is divided into five main components: Logger, Classifier, Clusterer, Mapper, and Fixer.

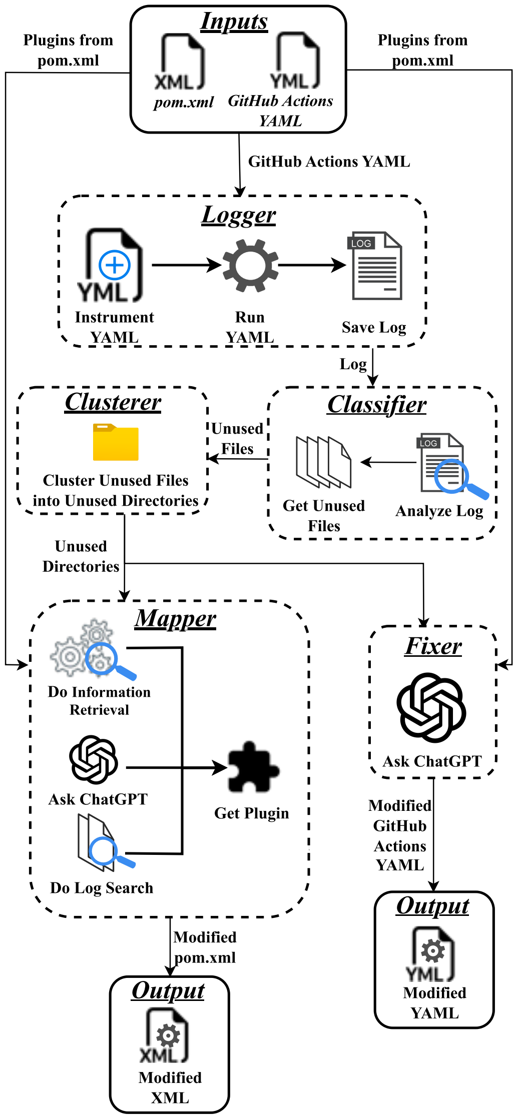

Fig. 2. Overview of OptCD technique.

Figure 2 illustrates OptCD's process and how its components interact with each other.

A. Logger

The Logger component tracks the file operations that occur during a CD build. Specifically, for a given project, OptCD modifies its GitHub Actions workflow `.yml` configuration file to install and start `inotifywait` [21], an existing Linux-based tool that can track changes to files recursively contained within a directory. When `inotifywait` is running, it records the operations that occur for all files under the specified directory. The file operations that `inotifywait` records are Created, Modified (written to), Accessed (read from), Closed, and Deleted. `inotifywait` also records the timestamp of when each such operation occurs for a file.

We configure the workflow file such that each job starts up `inotifywait` before the first step, tracking changes to files contained within the main work directory where the job pulls in the code and performs the overall build. We configure inotifywait to continue until the last step in the workflow before uploading the generated log for later use.

The added highlighted lines in Figure 1 shows an example of the modifications OptCD makes. The modifications involve adding extra steps to install inotifywait and other dependencies for the analysis we want to perform (Lines 11 to 14) and to start inotifywait (Line 16). We also include additional steps that touch and remove dummy files based on the original steps included in the existing workflow file (Lines 17 and 18). We use this modification to discern which file operations belong to which original steps by mapping file operations to a step based on the creation and removal of the dummy files. We include a final step that pushes the results for analysis later (Lines 22 and 23).

B. *Classifier*

Given the log file generated by the Logger, the Classifier component determines which files are unused files. We define an unused file as one that is generated and written to during the build, but that file is never read from later. The intuition is that the build has to do some work to generate the output that is ultimately written into this file, but since the contents of the file are never used later, the work that went into modifying the file is unnecessary.

For each file in the inotifywait log, we track the files that are Created during the build. For each such file, we look for the last time the file is Modified, representing something being written into the file. We then check whether there are any Accessed operations on the same file later. If the file is not Accessed after it is Created or after the final Modified, then we classify this file as an unused file. Note that when we track files, we use both the relative path of the file and when the file is Created in the log. If a file with the same file path is Created again later in the log, we treat that file as a brand new file to see whether it is an unused file.

C. *Clusterer*

We ultimately want to identify the Maven build plugins that generate unused files, so that a developer can reconfigure or disable the plugins to not do the unnecessary work that results in the unused files. Our intuition is that a plugin often does not modify specifically a single, individual file; they often do work across many files that are contained within the same directory. As such, we should try to map plugins to the directories that contain the individual unused files. We define an unused directory as a directory that contains only unused files. If the directory contains any non-unused files (i.e., necessary files that are used in the CD build), then the directory is not an unused directory. While we can map individual files to Maven build plugins without clustering them into directories, doing so for every single file can be time-consuming and likely we would still find that all unused files within the same unused directory would map to the same plugin anyway.

Given the unused files from the Classifier as input, the Clusterer clusters these unused files into their corresponding unused directories and outputs these directories. Figure 3

```
1  def Clusterer(unused_files, used_files):
2    paths = []
3    unused_dirs = []
4    paths.append(longest_path_prefix(unused_files))
5    while len(paths) > 0:
6      path = paths.pop()
7      # Some used file matches path
8      if path_match(path, used_files):
9        # Explore matching subdirs among unused files
10       for f in unused_files:
11         if path_match(path, f):
12           new_path = one_deeper(path, f)
13           if is_file(new_path):
14             continue
15           if not new_path in paths
16               and not new_path in unused_dirs:
17             paths.append(new_path)
18     else:
19       unused_dirs.append(path)
20   return unused_dirs
```

Fig. 3. Pseudo-code for Clusterer.

shows how the Clusterer identifies the unused directories. Given the set of unused files and used files (namely the remaining files that are not unused files), the Clusterer first finds the longest common prefix path among all unused files. If this path also matches for any used files (Line 8), then the path does not represent an unused directory. Clusterer needs to then explore deeper down this path, trying out the different subdirectories underneath to see whether any of those can be unused directories. For each unused file that matches the current path prefix, Clusterer constructs a new path that is the current path prefix plus one more directory deeper (Line 12), which still matches that unused file. If the path cannot be extended by one more directory, namely it results in the file itself, then the path is discarded. This scenario occurs when there is a directory that directly contains both unused files and used files, so the directory is not an unused directory and cannot be extended further. We do not report individual unused files and we report only the top most level of unused directory (unused subdirectories are also not reported).

If the new path is not already in the set of paths to be explored and not already identified as an unused directory (Line 16), then Clusterer checks whether this path is an unused directory in future iterations (Line 17). When there are no more paths to explore, Clusterer returns all the unused directories it identified. For example, the target/site/ directory may contain various other directories, such as jacoco/ and pmd/. If all the (recursive) files inside jacoco/ are unused files, then jacoco/ will be considered an unused directory. If at least one file is a used file in pmd/, then the pmd/ directory will not be considered an unused directory. However, if all files in pmd/ are also unused files, then pmd/ will also be considered an unused directory like jacoco/. If all the directories under target/site/ are all unused directories, then target/site/ will be considered an unused directory.

D. *Mapper*

The Mapper component takes the set of unused directories from the Clusterer and identifies the plugin that modifies the files in each unused directory. To identify the correct plugin for each unused directory, we use a systematic search process that iterates through the plugins defined in a project's `pom.xml` build configuration file to see which one generates the unused directory (more precisely, we iterate through the effective `pom.xml` that also contains plugins inherited from parent `pom.xml` files). For each plugin, we remove the plugin from the `pom.xml` file and then run the steps. If all steps complete successfully, the unused directory no longer exists, and all the used files are still there, then we consider the disabled plugin to be the one responsible for generating the unused directory. The developer may then use this information to debug and determine how to best prevent the plugins from generating unused files, or simply disable or remove the plugin from their local and CD builds. As special cases, we exclude the directories `maven-status/` and `surefire-reports/` if they are the unused directories. The reason is that we know that the two directories are generated by the Compiler and Surefire plugins, respectively. These two plugins are always included in any Maven build and cannot be removed. Therefore, we do not bother searching in such cases.

Overall, this process of trying each plugin can be time-consuming due to the rerunning of all steps for each plugin tried during the search. To speed up this process, we propose three strategies to rank the list of plugins to try.

Information Retrieval. We use information retrieval, namely TF-IDF [13], to match the name of an unused directory to the likely plugin that modifies the unused directory. The intuition is that the unused directory name is likely related to the plugin's name (as listed in the `pom.xml` file), e.g., the JaCoCo plugin by default writes to a directory called `jacoco`. To rank the plugins, we tokenize the name (group ID and artifact ID) of each plugin listed in the `pom.xml`. We treat the tokenization of the unused directory name as the query and the tokenization of each plugin as the documents to match the query against. TF-IDF provides a cosine similarity score between 0 and 1 for each plugin based on its relevance to the unused directory name, and we rank the different plugins based on this score. We then perform the search by iterating through the plugins in decreasing score order. Plugins with the same score are ordered based on their appearance in the `pom.xml` file. A plugin with score 0 is considered unranked. If none of the ranked plugins are correct, we proceed to search the unranked plugins (in order of appearance in the `pom.xml` file, because they all tie with the same score of 0). We stop the search after identifying a correct plugin and consider an unused directory to have no correct plugins if we do not stop the search before trying all plugins. As we show in Section V-B, our search process can identify the correct plugin for 92.0% of the unused directories in our evaluation.

ChatGPT. ChatGPT [14] is a large-language model-based chatbot. ChatGPT can help with a wide range of tasks, including coding-related questions. We utilize ChatGPT (version 4.0) as a "Google search" equivalent to represent what a developer would do if they were trying to identify and disable the plugin responsible for generating unused files.

To prepare the ChatGPT queries, we considered two different prompting strategies: Instruction Following Prompting Strategy [22] and TruthfulQA Prompting Strategy [23]. We selected all the strategies from Wei et al. [24] that did not require follow-up prompts. Out of these two strategies, we chose the Instruction Following Prompting Strategy based on a small experiment to find the correct plugin for a sample of five unused directories from our evaluation dataset. Based on this experiment, we found that the Instruction Following Prompting Strategy gives more accurate results.

Given an unused directory for a step in the GitHub Actions `.yml` file along with all the plugins defined in the `pom.xml`, we construct a prompt to ChatGPT based on the Instruction Following Prompting Strategy, asking it for a ranked list of plugins along with a score for how likely each plugin is responsible for generating that unused directory. The prompt template for ChatGPT is:

Given the list of dependency plugins in a `pom.xml` as "<list of plugins>", a Maven command that has been executed as "<Maven Command>", and an unused directory that was created in the target directory named "<unused directory>", please rank and score the plugins in the list based on how likely each plugin is responsible for the generation of this unused directory and the score should be greater than or equal to zero, where zero means the plugin may not likely generate the unused directory.

We perform the search by iterating through each plugin in descending order of the score given by ChatGPT. Similar to the Information Retrieval strategy, we first search the ranked plugins before the unranked ones and we break ties in scores the same way (based on their appearance in the `pom.xml`).

Log Search. Our last strategy for identifying the plugin responsible for generating an unused directory is to parse the log files directly. The log files generated by the Logger component have timestamps for each file event. A GitHub Actions build also produces timestamped logs detailing all the Maven build activities, including when each plugin starts running. As such, we can correlate the timestamps for the creation and modification of files in the Logger logs with the timestamps of when plugins were started and ended in the GitHub Actions build logs to figure out which plugin was running when unused files were generated.

Similar to the other two strategies, Log Search also produces a ranked list of plugins to try, except its "list" consists of just the one plugin found to be running at the same time as when the unused files were created or modified. If we find this plugin is not the correct one for generating the unused directory, we once again search through the remaining unranked plugins.

E. Fixer

In the Fixer component, we use ChatGPT to obtain a new Maven command that disables the generation of unused directories. The goal is to update the Maven command in the GitHub Actions workflow `.yml` build configuration file. While a developer may use the plugin information from the Mapper to simply disable or remove the plugin completely from their Maven build, e.g., removing the plugin from their `pom.xml` configuration file, doing so would disable the plugin from running both on the CD servers and on the developers' local machine. The generated files that OptCD identifies on CD servers are unused files during a CD build, but they may be used by developers when they build locally, e.g., they may use detailed test reports for debugging local test failures, while such reports are not saved anywhere and are deleted after a CD build. As such, it is desirable to also find a way to disable the generation of unused files only in CD builds by adding arguments to the Maven command used in the GitHub Actions `.yml` build configuration file. By modifying only the `.yml` file with these arguments, only the CD builds would be affected, i.e., local builds would not be affected.

Similar to how we use ChatGPT for Mapper, we also use the Instruction Following Prompting Strategy to develop the prompts for how to fix the Maven commands. We provide to ChatGPT the entire list of plugins from the `pom.xml` file, the unused directory from Clusterer, and the Maven command from the step in the GitHub Actions `.yml` file. While we could also provide to ChatGPT the name of the responsible plugin from the Mapper (if identified successfully), our experiments find that providing this extra information does not help generate more fixes than just providing the entire list of plugins. By relying on the entire plugin list, the Fixer is also usable even when the Mapper did not successfully identify a plugin. Ultimately, our prompt to ChatGPT is:

Given the list of dependency plugins in a `pom.xml` represented as "<list of plugins>", and considering the Maven command "<Maven command>" which results in the creation of an unused directory named "<unused directory>" in the target directory, please provide an argument that can be added to the given Maven command. The objective is that when this updated command is executed, the "<unused directory>" should not be created. Suggest the most relevant argument based on the context and the nature of the plugins listed.

We test the resulting commands by updating the step in the GitHub Actions `.yml` file and running the build. If the unused directory is still generated, then we say ChatGPT was unsuccessful at fixing the build. If the unused directory is no longer generated and all used files are still generated, then we consider the new Maven command to be valid and report it as a patch to developers.

IV. Experimental Setup

We answer the following research questions:

- **RQ1**: How often are unused directories generated?
- **RQ2**: How effective is OptCD at identifying the plugin that generated unused directories and suggesting ways to prevent the generation of such directories?
- **RQ3**: How much runtime does OptCD save?
- **RQ4**: What are developers' reaction to OptCD-inspired fixes?

We answer RQ1 to see how prevalent are unused directories being generated within CD builds. A large percentage of unused directories can indicate much unnecessary work that should be removed. We answer RQ2 to see how effective OptCD is at identifying what part of a build is responsible for generating unused directories as well as how it can help fix that problem. We answer RQ3 to understand how OptCD can help in reducing CD build time by preventing the generation of unused directories. Finally, we answer RQ4 to understand how interested developers are in the the fixes inspired by OptCD.

A. Subjects

We focus our evaluation on open-source Maven projects that build using GitHub Actions. First, we queried the GitHub REST API to search for popular projects marked with the Java programming language. We configured the search to sort the projects based on the number of forks as a measure of popularity. As the GitHub REST API cannot provide more than 1000 results at once, we stagger the search across two days, resulting in a total of 1563 unique Java projects. We then filtered this list of projects to include only the ones that are already configured to use GitHub Actions and use Maven as their build system, resulting in 215 projects.

Given that we will be running the workflows for these projects multiple times, we apply an additional filtering criteria to only include the projects with one to three `.yml workflow` files, reducing the number of workflows and jobs that get triggered each build. For ease of our analysis, we also filter out projects with more than one `pom.xml` file, restricting ourselves to only single module Maven projects. After these two filters, we are left with a total of 54 projects.

We forked these 54 projects and attempted to trigger a GitHub Actions build. We are unable to include projects where we cannot trigger a successful GitHub Actions build for any of the project's jobs, for reasons such as the job requiring some form of authentication from a secret key that we do not have or the job must run on self-hosted runners that we do not have access to. We also filter out projects with workflows on which we cannot successfully run OptCD due to reasons such as the job being configured to run on a non-Linux operating system, which `inotifywait` does not support, or there are strict timeouts that prevents our analysis from finishing. Ultimately, we are left with 22 projects with GitHub Actions workflows that we use for our evaluation.

B. RQ1: How often are unused directories generated?

We rely on `inotifywait` [21] to track file accesses during a CD build (Section III-A). Before settling on using `inotifywait`, we tried `lsof` and `strace` as other options for logging changes to files. We find that these other two tools produced output that are more difficult to parse than `inotifywait`, e.g., `strace` produced very detailed output but most of it is unnecessary for the purpose of tracking file reads and writes.

We conducted all experiments to collect `inotifywait` logs on default Ubuntu GitHub Actions machines that have two cores, 7GB RAM, and 14GB of disk storage. Note that we are running the same build as the developers would every time there are updates to their codebase, new pull requests being opened, etc. The only change we make to a build is to setup our logging needs. We also measure build-related runtimes on these machines. Timing measurements with and without fixes are done on original builds with no changes from us except for our commands to disable the generation of unused files. While developers can configure their builds to run different build profiles, including those that may be less expensive by disabling certain work during CD, the results we observe concerning generated unused files are what actually happens using their current build profiles.

C. RQ2: How effective is OptCD at disabling the generation of unused directories?

We obtain the ground truth for the correct plugin by disabling each plugin that is listed in the `pom.xml` file one-by-one and then running the Maven command to observe if the unused directory is still being generated. If the unused directory is no longer generated and the used files are still being generated, then we consider that plugin as the correct plugin to disable the unused directory.

D. RQ3: How much runtime does OptCD save?

After OptCD suggests some changes to the Maven build commands in one or more steps in a job, we run those jobs through GitHub Actions 14 times, both with and without the suggested changes. For each run, we record the runtime of each step, but we exclude the time taken to download dependencies, because the download time varies from run to run. The fluctuations in download time can mask and mischaracterize the actual runtime improvement from OptCD's proposed changes. Further, to remove the effect of outliers, we dropped the two highest and the two lowest runtimes for both with and without our fix. We then compute the runtime improvement as subtracting the average runtime with the change from the average runtime without the change, divided by the average runtime without the change. We only compute the runtime improvement for steps where there is a change to the Maven commands, because we want developers to accept our changes for runtime improvements, which developers are more likely to accept if our changes applied only to their CD build. To assess the statistical significance of the runtime improvements, we also conduct a two-sample independent t-test [25] on the obtained runtimes, comparing the runtimes with and without the changes suggested by OptCD.

E. RQ4: What are developer's reaction to OptCD?

We submit pull requests to the developers of the open-source projects to gauge their interest in the suggested fixes. Although, we attempt to open a pull request for all unused directories that OptCD can generate a valid Maven command for, our total number of pull requests opened is less, because many unused directories come from the same step, which we only opened a single pull request for in such cases. In fact, we find that one workflow may often have multiple jobs, and each job will have the same steps but the jobs slightly differ in their Java version, OS version, and so on. One example of a workflow with multiple jobs that have different Java versions is shown in Line 8 of Figure 1. The steps are the same in the different jobs, so we can create one pull request that helps with all of the jobs in this workflow.

V. Evaluation

A. RQ1: How often are unused directories generated?

Table I shows the number of unused directories that OptCD finds to be generated during the CD build across the 22 projects and the 43 steps (each given an ID for use in a later table) that run a Maven build command within the jobs of those projects, shown under column "U. Dir.". For each step, we also show the total number of steps within the same job as that step ("Steps"), including those that do not run a Maven command, and the number of plugins run within that step ("Plugins"). We see that almost every step with a Maven command generates an unused directory, with only three steps that do not generate any unused directories. Overall, all the steps together generate a total of 89 unused directories. Further, when we analyze the directories with at least one file generated, we find that 64.9% of them contain more unused files than used files.

When we look into the unused directories, we find that almost all of the steps generate a directory with the name `maven-status/` (only step S1-1a does not generate a `maven-status/` directory). The `maven-status/` directory contains diagnostics information generated during a Maven build. As such, it is reasonable to find that projects tend to not do anything with the contents in this directory during CD. Further, we do not find any means to prevent a Maven build from generating such a directory. To focus our efforts on unused directories that we can disable generating, we ignore the `maven-status/` directory. Table I reports the number of unused directories without any `maven-status/` directories under the column "U-MS Dir.". There are a total of 50 unused directories generated across all steps. We see that there are only six steps that do not generate any unused directories, i.e., 86.0% (37 / 43) of steps generate at least one unused directory. Each step generates on average 2.1 (89 / 43) unused directories. Further, 95.6% (21 / 22) of all projects generate at least one unused directory (only one project does not), with an average of 4.0 (89 / 22) unused directories per project. In summary, the generation of unused directories is rather prevalent in CD builds, showing the large amount of unnecessary work conducted within these builds.

RQ1 findings: 95.6% of projects generate at least one unused directory in its Maven-command steps, with an average of 4.0 unused directories per project. When we consider just the Maven-command steps, we find that an average of 2.1 unused directories are generated per step.

B. RQ2: How effective is OptCD at disabling the generation of unused directories?

Table II shows the results of using OptCD to identify the correct Maven plugins that generate the 50 unused directories, i.e., the effectiveness of the Mapper component. Each row in the table shows the name of the unused directory associated

TABLE I
STATISTICS OF GITHUB ACTIONS STEPS IN OUR EVALUATION. THE "-" IN CELLS UNDER "IMPROVEMENT" AND "IMPROV. P-VALUE" ARE CASES WHERE WE CANNOT MEASURE RUNTIME IMPROVEMENT. HIGHLIGHTED P-VALUE CELLS INDICATE STATISTICALLY SIGNIFICANT IMPROVEMENTS ($p < 0.05$).

ProjID		# of				% Runtime		Improv.
-StepID	GitHub User/Repository Name	Steps	Plugins	U. Dir.	U.-MS Dir.	Step/Total	Improvement	p-value
S1-1a	pedrovgs/Algorithms	4	8	1	1	45.9	7.8	0.320
S1-1b	pedrovgs/Algorithms	-	8	2	1	34.7	2.2	0.488
S2-1	cucumber/cucumber-java-skeleton	8	8	1	0	15.2	-	-
S3-1	baomidou/dynamic-datasource-spring-boot-starter	3	12	1	0	98.0	-	-
S4-1	hyperledger/fabric-sdk-java	3	15	2	1	99.2	-2.8	0.151
S4-2	hyperledger/fabric-sdk-java	3	15	2	1	99.2	3.2	0.052
S4-3	hyperledger/fabric-sdk-java	3	15	2	1	99.3	-3.0	0.197
S5-1	hub4j/github-api	3	16	4	3	85.3	9.2	0.076
S5-2	hub4j/github-api	3	16	2	1	82.2	-	-
S5-3	hub4j/github-api	4	16	3	2	80.6	-	-
S5-4	hub4j/github-api	4	16	3	2	94.4	-0.7	0.708
S5-5	hub4j/github-api	4	16	2	1	73.9	-	-
S6-1	TooTallNate/Java-WebSocket	3	10	0	0	58.5	-	-
S7-1	JSQLParser/JSqlParser	3	20	3	2	90.3	14.2	0.000
S7-2	JSQLParser/JSqlParser	3	20	3	2	89.3	8.7	0.002
S8-1	junit-team/junit4	5	14	2	1	89.5	12.8	0.000
S8-2	junit-team/junit4	5	14	2	1	82.8	3.9	0.161
S8-3	junit-team/junit4	5	14	2	1	88.0	7.0	0.019
S8-4	junit-team/junit4	5	14	2	1	88.8	10.9	0.006
S9-1	mate-academy/jv-fruit-shop	3	9	1	0	42.6	-	-
S10-1	obsidiandynamics/kafdrop	3	12	3	2	81.1	5.5	0.058
S11-1	matsim-org/matsim-example-project	3	10	2	1	92.0	15.0	0.000
S12-1	mcMMO-Dev/mcMMO	4	12	2	1	86.9	7.0	0.001
S13-1	SERG-Delft/mooc-software-testing	3	9	4	3	55.1	24.1	0.000
S14-1	pagehelper/Mybatis-PageHelper	3	9	2	1	64.1	-0.2	0.921
S15-1	google/open-location-code	3	9	2	1	93.6	0.1	0.970
S15-2	google/open-location-code	3	9	2	1	80.7	0.6	0.807
S15-3	google/open-location-code	3	9	2	1	92.4	4.8	0.143
S15-4	google/open-location-code	3	9	2	1	91.9	8.1	0.005
S16-1	MicrosoftDocs/pipelines-java	4	9	2	1	33.4	-	-
S17-1	socketio/socket.io-client-java	5	16	2	1	88.5	-3.5	0.000
S17-2	socketio/socket.io-client-java	5	16	2	1	92.6	-0.3	0.674
S17-3	socketio/socket.io-client-java	5	16	2	1	90.0	-0.5	0.330
S18-1	soot-oss/soot	3	15	2	1	95.7	7.5	0.022
S18-2	soot-oss/soot	3	15	2	1	96.0	10.6	0.001
S18-3	soot-oss/soot	3	15	2	1	95.8	0.6	0.874
S18-4a	soot-oss/soot	4	15	0	0	47.5	-	-
S18-4b	soot-oss/soot	4	15	0	0	21.3	-	-
S19-1	spring-petclinic/spring-framework-petclinic	3	14	4	3	71.9	16.2	0.000
S20-1	spring-petclinic/spring-petclinic-rest	3	13	3	2	70.8	22.0	0.000
S21-1	88250/symphony	3	10	3	2	66.2	46.9	0.000
S22-1	UniversalMediaServer/UniversalMediaServer	4	14	2	1	76.3	-4.2	0.228
S22-2	UniversalMediaServer/UniversalMediaServer	4	14	2	1	71.9	-1.2	0.296
Total/Mean		**3.7**	**13.0**	**89**	**50**	**76.6**	**7.0**	**-**

with a step ID. The table also shows the ground truth plugin responsible for generating each unused directory based on our inspection (Section IV). Note that two unused directories (S7-1 and S7-2 `site`) are generated using more than one plugin.

The column "Baseline" shows the number of plugins tried when using a simple baseline of iteratively trying to disable each plugin in order of appearance in the `pom.xml` file (i.e., without any of the ranking heuristics that Mapper employs). This baseline would try on average 6.5 plugins per directory (we do not count cases where we cannot identify the correct plugin by this iterative search, marked as "-" in the table).

For the three strategies for ranking plugins during Mapper's search (Information Retrieval, ChatGPT, and Log Search), we show the number of plugins each would try before identifying the correct plugin under column "#Tried"; we use "-" to denote when a strategy cannot identify the plugin. We also show the number of plugins that are actually ranked (column "#Rnkd") by Information Retrieval and ChatGPT; Log Search would only ever rank one plugin, so we do not show it in the table.

Comparing the number of plugins tried by each strategy, we see that all of them try a smaller number of plugins compared to the baseline, trying on average 2.0, 1.6, and 1.0 plugins for Information Retrieval, ChatGPT, and Log Search, respectively. ChatGPT on average creates larger ranked list of plugins than Information Retrieval (2.1 versus 1.1), and both of them occasionally need to try plugins beyond the ranked list, but ChatGPT ends up trying fewer plugins than Information Retrieval. Log Search is the most effective as it tries the fewest number of plugins among the three strategies.

Overall, we see that the search process of trying each plugin can identify the correct plugin for 92.0% (46 / 50) of the unused directories. Naturally, this process cannot handle cases

TABLE II
PLUGIN ANALYSIS RESULTS FOR EACH UNUSED DIRECTORY. "-" DENOTES THE SEARCH COULD NOT IDENTIFY THE CORRECT PLUGIN. "#RNKD" DENOTES RANKED. "N/A" DENOTES NO PLUGINS WERE RANKED. "*" DENOTES A PULL REQUEST IS SUBMITTED FOR THE DIRECTORY.

			Baseline	Information Retrieval		ChatGPT		Log Search	
ID	Unused Dir	Ground Truth	#Tried	#Tried	#Rnkd	#Tried	#Rnkd	#Tried	Fixer
S1-1a	site	-	-	-	1	-	1	-	N*
S1-1b	surefire-reports	maven-surefire-plugin	5	1	1	1	1	1	Y*
S4-1	surefire-reports	maven-surefire-plugin	1	1	1	1	2	1	Y*
S4-2	surefire-reports	maven-surefire-plugin	1	1	1	1	2	1	Y*
S4-3	surefire-reports	maven-surefire-plugin	1	1	1	1	2	1	Y*
S5-1	cache	maven-surefire-plugin	8	8	N/A	8	6	1	N
S5-1	japicmp	japicmp-maven-plugin	11	1	1	1	1	1	Y*
S5-1	surefire-reports	maven-surefire-plugin	6	1	2	1	2	1	Y*
S5-2	site	-	-	-	1	-	3	-	N
S5-3	cache	maven-surefire-plugin	8	8	N/A	8	6	1	N
S5-3	surefire-reports	maven-surefire-plugin	6	1	2	1	2	1	Y*
S5-4	cache	maven-surefire-plugin	8	8	N/A	8	6	1	N
S5-4	surefire-reports	maven-surefire-plugin	6	1	2	1	2	1	Y*
S5-5	test-classes	maven-compiler-plugin	5	5	N/A	1	2	1	N
S7-1	site	maven-pmd-plugin & jacoco-maven-plugin	-	-	1	-	8	-	N*
S7-1	surefire-reports	maven-surefire-plugin	15	1	1	1	1	1	Y*
S7-2	site	maven-pmd-plugin & jacoco-maven-plugin	-	-	1	-	8	-	N
S7-2	surefire-reports	maven-surefire-plugin	15	1	1	1	1	1	Y*
S8-1	surefire-reports	maven-surefire-plugin	5	1	1	1	1	1	Y*
S8-2	surefire-reports	maven-surefire-plugin	5	1	1	1	1	1	Y*
S8-3	surefire-reports	maven-surefire-plugin	5	1	1	1	1	1	Y*
S8-4	surefire-reports	maven-surefire-plugin	5	1	1	1	1	1	Y*
S10-1	docker-ready	maven-resources-plugin	5	5	1	3	2	1	Y*
S10-1	surefire-reports	maven-surefire-plugin	9	1	1	1	2	1	Y*
S11-1	surefire-reports	maven-surefire-plugin	7	1	1	1	1	1	Y*
S12-1	surefire-reports	maven-surefire-plugin	1	1	1	1	2	1	Y*
S13-1	jacoco-ut	jacoco-maven-plugin	1	1	1	1	1	1	Y*
S13-1	site	jacoco-maven-plugin	9	2	1	2	1	1	N
S13-1	surefire-reports	maven-surefire-plugin	3	1	1	1	1	1	Y*
S14-1	surefire-reports	maven-surefire-plugin	6	1	1	1	1	1	Y*
S15-1	surefire-reports	maven-surefire-plugin	12	1	1	1	1	1	Y*
S15-2	surefire-reports	maven-surefire-plugin	12	1	1	1	1	1	Y*
S15-3	surefire-reports	maven-surefire-plugin	12	1	1	1	1	1	Y*
S15-4	surefire-reports	maven-surefire-plugin	12	1	1	1	1	1	Y*
S16-1	test-classes	maven-compiler-plugin	5	5	N/A	1	3	1	N
S17-1	surefire-reports	maven-surefire-plugin	3	1	1	1	2	1	Y*
S17-2	surefire-reports	maven-surefire-plugin	3	1	1	1	2	1	Y*
S17-3	surefire-reports	maven-surefire-plugin	3	1	1	1	2	1	Y*
S18-1	surefire-reports	maven-surefire-plugin	5	1	1	1	2	1	Y*
S18-2	surefire-reports	maven-surefire-plugin	5	1	1	1	2	1	Y*
S18-3	surefire-reports	maven-surefire-plugin	5	1	1	1	2	1	Y*
S19-1	.wro4j	wro4j-maven-plugin	8	1	1	1	1	1	N
S19-1	site	jacoco-maven-plugin	6	7	1	6	1	1	N*
S19-1	surefire-reports	maven-surefire-plugin	2	1	1	1	1	1	Y*
S20-1	site	jacoco-maven-plugin	2	2	1	2	1	1	N*
S20-1	surefire-reports	maven-surefire-plugin	10	1	1	1	1	1	Y*
S21-1	symphony	maven-assembly-plugin	4	4	N/A	1	3	1	Y*
S21-1	test-classes	maven-compiler-plugin	7	7	N/A	1	3	1	N*
S22-1	surefire-reports	maven-surefire-plugin	12	1	1	1	1	1	Y*
S22-2	surefire-reports	maven-surefire-plugin	12	1	1	1	1	1	Y*
Total/Mean			**6.5**	**2.0**	**1.1**	**1.6**	**2.1**	**1.0**	**Y=36 / N=14**

where multiple plugins are involved with the generation of the unused directory (S7-1 and S7-2 `site`). In the future, we can explore strategies that can try combinations of plugins. We also do not handle two cases (S1-1a and S5-2 `site`) where the responsible plugins are directly invoked by the Maven commands (e.g., the `checkstyle` plugin being invoked with `mvn checkstyle:checkstyle`). We do not consider such cases to be possible for our process to handle as disabling the plugin would fail the build in such cases.

We also evaluate the effectiveness of the Fixer component, which may suggest a change to the Maven command in the `.yml` CD configuration file to disable generating unused directories. The Fixer could successfully give a fix for 36 unused directories (marked with "Y" under column "Fixer"), i.e., 72.0% (36 / 50) of the unused directories. We find that for five unused directories, ChatGPT was unable to provide a fix at all for how to disable their generation. For four directories, using the provided fix could not disable generating the unused directories. Finally, for the remaining five, while the fix does disable generating the unused directories, they also disabled generating some used files, so the solution is invalid.

RQ2 findings: The Mapper can identify the correct plugin for 92.0% of the unused directories, and all three strategies end up trying fewer plugins in their search than the baseline. Log Search is the most efficient, because it tries the fewest plugins. The Fixer can successfully propose a new Maven command to disable generating 72.0% of the unused directories.

C. RQ3: How much runtime does OptCD save?

Table I shows for each step the percentage of runtime improvement we can obtain from disabling the generation of the unused directories (column "% Runtime Improvement"). We show the improvements only for the 33 steps in which we could make a change to the Maven command to disable generating the directories. We also show for each step, the percentage of runtime that step by itself takes out of the full job runtime (column "% Runtime Step/Total"). We find that the steps that perform some Maven command take, on average, 76.6% of the overall job runtime.

We also see that the proposed changes to the Maven command can reduce the runtime of the corresponding step by 7.0%, on average, across all steps. There is a wide range of improvements between steps, where the highest runtime improvement is 46.9%, which comes from disabling the generation of all unused directories in that step, except for the `maven-status/` directory. We use a t-test [25] to compare the runtimes with and without the fixes and find that 15 out of 33 of the observed runtime differences are statistically significant, $p < 0.05$ (highlighted under column "Improv. p-value" in Table I). If we consider the runtime improvement only for these statistically significant steps, the average improvement is 13.8%.

There are nine steps with negative improvement, i.e., runtime for the step went up after our change. Our manual inspection finds that they are likely due to noise and the absolute difference is rather small, with only one of them (S17-1) being statistically significant.

RQ3 findings: The suggested changes to Maven commands by OptCD can reduce step runtimes by 7.0%, on average.

D. RQ4: What are developer's reaction to OptCD?

We submitted pull requests for all 33 steps in which we can disable the generation of unused directories using the changed Maven commands. We include the fixes for multiple steps in one pull request if the steps are from the same project and the fixes are similar. We submitted pull requests corresponding to all 36 unused directories marked with "Y" under column "Fixer" in Table II. In addition, from our manual inspection, we developed a way to fix an additional five more cases, leading to fixes handling a total of 41 unused directories (marked with "*" under "Fixer" in Table II).

In total, we submitted 26 pull requests. So far, 12 have been accepted (46.2%), five have been rejected, and nine are still open. Of the five rejected pull requests, two were rejected with no comments from developers and three were rejected because the developers claim that the improvement is minor and is worried about complications from disabling the plugin. Details of all pull requests are on our website [15].

RQ4 findings: We submitted 26 pull requests to disable the generation of unused directories. Developers accepted 12 of the pull requests, with five rejected, and nine still pending.

E. Comparison against BuildSonic

BuildSonic [26] and OptCD have very similar goals. At its core, BuildSonic fixes performance issues, statically detects configuration smells like deep clone and cache dependencies, and fixes configuration smells in build configurations. Given BuildSonic's focus on fixing performance configuration smells, it can serve as a baseline for comparison with OptCD. BuildSonic has strategies for a variety of build systems, while OptCD is focused on Maven [18]. BuildSonic's strategies for Maven include enabling parallel execution, enabling forks, setting fork count, and disabling test report generation.

We conduct a high-level analysis of the 1,263 pull requests submitted by BuildSonic in April 2023, revealing the following distribution: 745 closed pull requests, 280 open pull requests, and 238 deleted pull requests. Among the 745 closed pull requests, 521 were rejected and 224 were accepted. We observe that 314 out of the 521 rejected pull requests were closed without any specific reason (i.e., no comments in the pull request). When we inspected the remaining 207 rejected pull requests that included comments, we find that the majority of them get rejected due to reasons beyond any limitations related to the tool, e.g., 81 pull requests are rejected because the developers stopped using the CD service or 13 pull requests are rejected simply due to unclear writing in the pull request text. The pull requests rejected due to limitations of the tool are those rejected due to no major performance gains (28) or marked as having invalid changes (52).

We similarly find that developers may reject pull requests if they feel the performance gains are not high enough, despite the changes reducing the generation of unused files. A developer using either tool can ultimately choose to use the suggested changes if they feel the performance gains are sufficient. We then look into the pull requests with invalid changes (52), specifically focusing on the pull requests that were built using the BuildSonic strategy of disabling test report generation, which is related to the detection of a common type of unused directories that OptCD can find. We notice that BuildSonic disables test report generation for any project that runs tests. However, these reports could be helpful when a build fails. In fact, we find that one BuildSonic pull request was rejected because BuildSonic suggests to disable the generation of test reports even though these test reports are later uploaded when the build fails [27]. At the same time, OptCD marks these files as used files and does not suggest the developers to disable generating them. The reason that BuildSonic proposes such a change on any such build is because it only performs a static analysis on the build configuration, acting as a sort of linter, without looking deeper into whether the suggested changes are disabling the generation of files that are actually used by the build. Meanwhile, OptCD would find these files being used when it tracks file accesses during an actual run of the build, and it therefore would mark them as used files and not suggest to disable their generation. This example showcases a main benefit of dynamically detecting the generation of unused files compared to statically doing so.

VI. Threats to Validity

Our results may not generalize to all projects. We create our evaluation dataset in a systematic manner starting with a large number of popular Java projects and filtering to include all projects that match our criteria (Section IV-A). The projects we evaluate on use GitHub Actions as a CD service, which is easily integratable with projects hosted on GitHub. Given the large number of open-source projects on GitHub, we believe such projects can be representative of open-source projects. The core idea behind OptCD is to identify unnecessary work via unused files. While we rely specifically on `inotifywait` to track file accesses, which is available only on Linux systems, the ideas are still applicable on other operating systems with similar tooling. Further, while we specifically develop OptCD for Maven projects, the idea is applicable to projects that use other build systems (e.g., Gradle builds may also run plugins that generate unused files). We can use the same approach to identify unused files for different build systems, but we would need to adapt our current strategies, e.g., adapting to get the list of Maven plugins to another build system for the Mapper.

The goal of our work is to identify unnecessary work that occurs during CD builds, which the contents of the build are typically destroyed shortly after it finishes. To accomplish this goal, we leverage the insight that unused directories represent unnecessary work. It may be the case that some of the unused directories we identify are actually useful for developers, but not visible in the constraints of the CD build that we observe. Further, it may be the case that for some plugins, OptCD can suggest disabling the plugin from running (consequently from generating unused files), while for other plugins, it can only suggest to disable generating the unused files. The correct solution depends on the plugin, e.g., the testing plugin should not be disabled, as we still want to run tests, but we should disable it from generating unused test reports. We mitigate these threats by manually checking the suggested fixes. We also send our fixes as pull requests to developers, checking whether such fixes are acceptable to developers.

We may also miss detecting some unused files, because, while these files are eventually read from, those read operations themselves are not so useful, e.g., copying the file elsewhere and then not reading any further. As such, the number of unused files and unused directories we identify may be fewer than the actual number.

There may also be noise in our runtime measurements, which are conducted on the GitHub Actions CD machines. We mitigate this threat by rerunning the jobs multiple times and then dropping the best two and worst two runtimes to discard large outliers. We ultimately end up with 10 runtimes to use for comparison. While we develop a systematic way to prompt ChatGPT for Mapper and Fixer, the same query may result in different responses given its nondeterministic nature. We experimented with querying ChatGPT multiple times, and we find that its answers do not change substantially (e.g., ChatGPT always fixed the same unused directories).

VII. Related Work

Hilton et al. performed extensive empirical studies on CD [1], [2], showing its importance along with the developers' perspective on the process. There has been extensive work in improving the efficiency of builds in general [28]–[30]. Telea and Voinea proposed decomposing C/C++ header files to remove performance bottlenecks from compilation [31]. Vakilian et al. proposed refactoring targets in a distributed build system to improve build times [32]. Work in build prediction aims to predict build outcomes as to skip running builds entirely [33]–[36]. In contrast, our work focuses on dynamically tracking unused directories with the goal to disable their generation as to reduce build runtime.

Regression testing is a large part of CD, with the cost of CD largely coming from the need to run tests on every build. Researchers have proposed a number of ways to reduce the cost of regression testing, such as through regression test selection, which selects to run only the tests affected by the changes [9], [37]–[45], test-suite reduction, which reduces the set of tests to run [46]–[53], or test parallelization [54]–[56]. Other work in improving regression testing include changing how the underlying build system runs tests [57]. OptCD does not focus on just testing but rather on when the build generates unused directories, indicative of unnecessary work. Interestingly, we find cases where compiling tests is unnecessary as the build runs no tests.

VIII. Conclusions

We propose OptCD, a technique that can dynamically detect unnecessary work within CD builds by tracking whether the build generates unused files. The intuition is that generating these unused files wastes time, so disabling the build from generating them can reduce the build runtime. OptCD logs file operations that occur during a build and analyzes those operations to determine the unused files. It then clusters unused files together into unused directories and then systematically searches through the tasks to identify the likely one responsible for generating those unused directories. We also use ChatGPT to propose changes to the CD build command to disable generating unused directories during CD, successfully providing the necessary changes for 72.0% of the unused directories. We sent out 26 pull requests with these changes, with 12 accepted, five rejected, and nine still pending.

Acknowledgements

We would like to acknowledge NSF grant no. CCF-2145774 and Dragon Testing for their support on software testing research. We thank Nate Levin for his insightful comments.

References

[1] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *International Conference on Automated Software Engineering*, 2016, pp. 426–437.

[2] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: Assurance, security, and flexibility," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2017, pp. 197–207.

[3] P. Duvall, S. M. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007.

[4] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in GitHub," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2015, pp. 805–816.

[5] M. Fowler, "Continuous Integration," https://martinfowler.com/articles/continuousIntegration.html, 2006.

[6] "GitHub Actions," https://github.com/features/actions, 2023.

[7] "Jenkins," https://www.jenkins.io, 2023.

[8] "Travis-CI," https://travis-ci.org, 2018.

[9] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *International Symposium on Software Testing and Analysis*, 2015, pp. 211–222.

[10] "JaCoCo Java Code Coverage Library," https://github.com/jacoco/jacoco, 2023.

[11] "Apache Maven Javadoc Plugin," https://maven.apache.org/plugins/maven-javadoc-plugin, 2023.

[12] "Apache Maven PMD Plugin," https://maven.apache.org/plugins/maven-pmd-plugin/index.html, 2023.

[13] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information Processing & Management*, vol. 24, no. 5, pp. 513–523, 1988.

[14] "Introducing ChatGPT," https://openai.com/blog/chatgpt, 2022.

[15] "Optimizing continuous development by detecting and preventing unnecessary content generation," https://sites.google.com/view/optimizing-cd, 2023.

[16] "Understanding GitHub Actions," https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions, 2023.

[17] "JSqlParser," https://github.com/JSQLParser/JSqlParser, 2023.

[18] "Apache Maven Project," https://maven.apache.org, 2020.

[19] "Apache Maven Compiler Plugin," https://maven.apache.org/plugins/maven-compiler-plugin, 2023.

[20] "Maven Surefire Plugin," https://maven.apache.org/surefire/maven-surefire-plugin, 2023.

[21] "inotify," https://pypi.org/project/inotify, 2023.

[22] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," in *Advances in Neural Information Processing Systems*, vol. 35, 2022, pp. 27 730–27 744.

[23] S. Lin, J. Hilton, and O. Evans, "TruthfulQA: Measuring how models mimic human falsehoods," in *Association for Computational Linguistics*, 2022, pp. 3214–3252.

[24] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, "Emergent abilities of large language models," *Transactions on Machine Learning Research*, 2022.

[25] "SciPy ttest_ind_from_stats," https://docs.scipy.org/doc/scipy-1.3.2/reference/generated/scipy.stats.ttest_ind_from_stats.html, 2023.

[26] C. Zhang, B. Chen, J. Hu, X. Peng, and W. Zhao, "BuildSonic: Detecting and repairing performance-related configuration smells for continuous integration builds," in *International Conference on Automated Software Engineering*, 2023, pp. 1–13.

[27] "apache/maven-surefire pull request #526," https://github.com/apache/maven-surefire/pull/526, 2022.

[28] B. Adams, R. Suvorov, M. Nagappan, A. E. Hassan, and Y. Zou, "An empirical study of build system migrations in practice: Case studies on KDE and the Linux kernel," in *International Conference on Software Maintenance*, 2012, pp. 160–169.

[29] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of Java build systems," *Empirical Software Engineering Journal*, vol. 17, pp. 578–608, 2012.

[30] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. Hassan, "A large-scale empirical study of the relationship between build technology and build maintenance," *Empirical Software Engineering Journal*, vol. 20, pp. 1587–1633, 2014.

[31] A. Telea and L. Voinea, "A tool for optimizing the build performance of large software code bases," in *European Conference on Software Maintenance and Reengineering*, 2008, pp. 323–325.

[32] M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni, "Automated decomposition of build targets," in *International Conference on Software Engineering*, 2014, pp. 123–133.

[33] F. Hassan and X. Wang, "Change-aware build prediction model for stall avoidance in continuous integration," in *International Symposium on Empirical Software Engineering and Measurement*, 2017, pp. 157–162.

[34] X. Jin and F. Servant, "A cost-efficient approach to building in continuous integration," in *International Conference on Software Engineering*, 2020, pp. 13–25.

[35] A. Ni and M. Li, "Cost-effective build outcome prediction using cascaded classifiers," in *Mining Software Repositories*, 2017, pp. 455–458.

[36] Z. Xie and M. Li, "Cutting the software building efforts in continuous integration by semi-supervised online AUC optimization," in *International Joint Conference on Artificial Intelligence*, 2018, pp. 2875–2881.

[37] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering Methodology*, vol. 6, no. 2, pp. 173–210, 1997.

[38] L. Zhang, "Hybrid regression test selection," in *International Conference on Software Engineering*, 2018, pp. 199–209.

[39] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *International Symposium on Foundations of Software Engineering*, 2004, pp. 241–251.

[40] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *International Symposium on Foundations of Software Engineering*, 2016, pp. 583–594.

[41] A. Shi, P. Zhao, and D. Marinov, "Understanding and improving regression test selection in continuous integration," in *International Symposium on Software Reliability Engineering*, 2019, pp. 228–238.

[42] A. Gyori, O. Legunsen, F. Hariri, and D. Marinov, "Evaluating regression test selection opportunities in a very large open-source ecosystem," in *International Symposium on Software Reliability Engineering*, 2018, pp. 112–122.

[43] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *International Conference on Software Engineering, Software Engineering in Practice*, 2019, pp. 91–100.

[44] A. Shi, M. Hadzi-Tanovic, L. Zhang, D. Marinov, and O. Legunsen, "Reflection-aware static regression test selection," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 187:1–187:29, 2019.

[45] A. Shi, S. Thummalapenta, S. K. Lahiri, N. Bjørner, and J. Czerwonka, "Optimizing test placement for module-level regression testing," in *International Conference on Software Engineering*, 2017, pp. 689–699.

[46] T. Y. Chen and M. F. Lau, "A new heuristic for test suite reduction," *Journal of Information and Software Technology*, vol. 40, no. 5-6, pp. 347–354, 1998.

[47] ——, "A simulation study on some heuristics for test suite reduction," *Journal of Information and Software Technology*, vol. 40, no. 13, pp. 777–787, 1998.

[48] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Journal of Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 219–249, 2002.

[49] H. Zhong, L. Zhang, and H. Mei, "An experimental study of four typical test suite reduction techniques," *Journal of Information and Software Technology*, vol. 50, no. 6, pp. 534–546, 2008.

[50] J. Black, E. Melachrinoudis, and D. Kaeli, "Bi-criteria models for all-uses test suite reduction," in *International Conference on Software Engineering*, 2004, pp. 106–115.

[51] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel, "On-demand test suite reduction," in *International Conference on Software Engineering*, 2012, pp. 738–748.

[52] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," in *International Conference on Software Maintenance*, 2001, pp. 92–102.

[53] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "An empirical study of JUnit test-suite reduction," in *International Symposium on Software Reliability Engineering*, 2011, pp. 170–179.

[54] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula, "CloudBuild: Microsoft's distributed and caching build service," in *International Conference on Software Engineering Companion*, 2016, pp. 11–20.

[55] O. Schwahn, N. Coppik, S. Winter, and A. Møller, "Assessing the state and improving the art of parallel testing for C," in *International Symposium on Software Testing and Analysis*, 2019, pp. 123–133.

[56] J. Candido, L. Melo, and M. d'Amorim, "Test suite parallelization in open-source projects: A study on its usage and impact," in *International Conference on Automated Software Engineering*, 2017, pp. 838–848.

[57] P. Nie, A. Celik, M. Coley, A. Milicevic, J. Bell, and M. Gligoric, "Debugging the performance of Maven's test isolation: Experience report," in *International Symposium on Software Testing and Analysis*, 2020, pp. 249–259.