BuildSonic: Detecting and Repairing Performance-Related Configuration Smells for Continuous Integration Builds

Chen Zhang*
School of Computer Science
Fudan University
Shanghai, China

Bihuan Chen*†
School of Computer Science
Fudan University
Shanghai, China

Junhao Hu*
School of Computer Science
Fudan University
Shanghai, China

Xin Peng*
School of Computer Science
Fudan University
Shanghai, China

Wenyun Zhao*
School of Computer Science
Fudan University
Shanghai, China

ABSTRACT

Despite the benefits, continuous integration (CI) can incur high costs. One of the well-recognized costs is long build time, which greatly affects the speed of software development and increases the cost in computational resources. While there exist configuration options in the CI infrastructure to accelerate builds, the CI infrastructure is often not optimally configured, leading to CI configuration smells. Attempts have been made to detect or repair CI configuration smells. However, none of them is specifically designed to improve build performance in CI.

In this paper, we first create a catalog of 20 performance-related CI configuration smells (PCSs) in three tools (i.e., Travis CI, Maven and Gradle) of the CI infrastructure for Java projects. Then, we propose an automated approach, named BuildSonic, to detect and repair 15 types of PCSs by analyzing configuration files. We have conducted large-scale experiments to evaluate BuildSonic. We detected 20,318 PCSs in 99.0% of the 4,140 Java projects, with a precision of 0.998 and a recall of 0.991. We submitted 1,138 pull requests for sampled PCSs of each PCS type, 246 and 11 of which have been respectively merged and accepted by developers. We successfully triggered CI builds before and after merging 288 pull requests, and observed an average build performance improvement of 12.4% after repairing a PCS.

CCS CONCEPTS

• **Software and its engineering → Software performance**; **Software configuration management and version control systems**.

KEYWORDS

continuous integration, configuration smells, build performance

*Also with Shanghai Key Laboratory of Data Science, and Shanghai Collaborative Innovation Center of Intelligent Visual Computing.
†Bihuan Chen is the corresponding author.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556923

ACM Reference Format:
Chen Zhang, Bihuan Chen, Junhao Hu, Xin Peng, and Wenyun Zhao. 2022. BuildSonic: Detecting and Repairing Performance-Related Configuration Smells for Continuous Integration Builds. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556923

1 INTRODUCTION

Continuous integration (CI) is a software engineering practice of allowing developers to frequently merge their code changes to a shared repository [14, 17]. CI has gradually gained wide adoption due to its capability of automating the build process, including installing dependencies, compiling source code, running static analysis, and executing test cases. The adoption of CI can bring many benefits to software development. For example, it helps to detect integration errors earlier, achieve better code quality, improve developer productivity, deliver faster, and reduce development risk [14, 27, 28, 64].

However, the adoption of CI can also incur high costs [28]. Particularly, one of the well-known costs is *long build time* (i.e., the time duration of a CI build is long) [21, 28, 70]. As evidenced by recent studies, long build time is common in open-source projects [9, 21], and it has been recognized as a common barrier and pain point faced by developers adopting CI [27, 70]. On the one hand, long build time makes it hard to get quick CI feedback, and developers have to wait for the long build to finish. As a result, developers may switch contexts and activities. This fragmented work is known to affect the productivity of developers and the speed of software development [14, 47], and thus overshadows the benefits of adopting CI. On the other hand, long build time consumes tremendous computational resources. For example, the CI system costs millions of dollars for the computation at Google [28] and Microsoft [26]. Hence, it is an important task to reduce the time duration (or improve the performance) of CI builds.

Multiple techniques have been proposed to improve the performance of CI builds. One line of work tries to lazily retrieve part of dependencies to reduce retrieval time [8] and design incremental build techniques to accelerate builds (e.g., [18, 63]). One thread of work attempts to skip the execution of some specific builds for saving their whole build time through manual identification [10, 12], automated identification techniques (e.g., [2, 32]) and build outcome prediction techniques (e.g., [9, 24]). Another thread of work is focused on test case prioritization (e.g., [15, 73]) and test case selection (e.g., [40, 60])

techniques to minimize test execution time in CI builds. However, these techniques can be considered as *heavyweight* as they need to be integrated as new tools/plugins into the existing CI infrastructure.

Complementary to the heavyweight techniques, one *lightweight* technique is to properly configure the existing CI infrastructure (e.g., Travis CI for the CI tool, and Maven for the automated build tool) to improve the build performance. This technique is feasible because i) some configuration options of the CI infrastructure may negatively affect the build performance, and ii) designers of the CI infrastructure are also aware of the significance of build performance and thus provide some configuration options to accelerate CI builds. For example, it is expected to have a speed improvement of 20%–50% by enabling parallel builds in Maven 3 [53].

However, there is a learning curve in grasping the usage of all the configuration options, while the configuration can be complex even for a simple CI pipeline [27]. In addition, during software evolution, the configuration can become less effective or more difficult to maintain in a timely fashion [13, 29, 77]. As a result, the CI infrastructure is often not optimally configured, leading to misconfigurations (i.e., *CI configuration smells*) that potentially violate best CI practices [14, 29] and thus hinder the correctness, maintainability, performance or security of CI builds. Initial attempts [20, 66] have been made to detect or repair CI configuration smells; i.e., Gallaba and McIntosh [20] detect four types of configuration smells in Travis CI and repair three of them, while Vassallo et al. [66] detect four types of configuration smells in GitLab. However, neither of them is specifically designed from the perspective of improving build performance in CI.

In this paper, we first create a catalog of 20 performance-related CI configuration smells (PCSs) by analyzing official documentations of the three tools (i.e., one CI tool, Travis CI, and two automated build tools, Maven and Gradle) in the CI infrastructure for Java projects. These PCSs capture the potential build performance issues caused by developers' misconfigurations. Then, we propose BuildSonic to automatically detect and repair PCSs so as to automatically improving build performance. It is enabled to detect and repair 15 types of PCSs by analyzing configuration files. To the best of our knowledge, our work is the first to systematically investigate PCSs.

We conduct large-scale experiments to evaluate BuildSonic with 4,140 Java projects hosted on GitHub. First, we investigate the prevalence of the 15 types of PCSs in the wild. Surprisingly, we detect a total of 20,318 PCSs, covering all PCS types and affecting 99.0% of the projects; and each project contains an average of 5 PCSs. Second, we achieve a precision of 0.998 in a sample of 1,171 detected PCSs, and a recall of 0.991 in a sample of 200 projects. Third, we submit 1,138 pull requests. 246 pull requests have been merged by developers with our repairs adopted, and 11 pull requests have been accepted. Fourth, we fork the projects, trigger a CI build before and after merging the 1,138 pull requests, and measure the build performance change. As our CI environment might be different from the one used in each project, we successfully trigger CI builds for 288 pull requests; and on average, the build performance is improved by 12.4% after repairing a PCS. Finally, we also measure the efficiency of BuildSonic. On average, BuildSonic takes 158.7 and 5.6 milliseconds to detect and repair the PCSs in each project.

In summary, this paper makes the following contributions.

```
1. language: java
2. jdk: # two jobs running with different jdk versions
3.   - openjdk8
4.   - openjdk-ea
5. jobs:
6.   fast_finish: true
7.   allow_failures: # the job with openjdk-ea won't affect build
8.     - jdk: openjdk-ea
9. script: # run Maven and Gradle command
10.  - travis_retry ./gradlew build
11.  - travis_wait 60 ./mvnw install
12.  - ./build.sh
13.git:
14.  depth: 9999
15.cache:
16.  directories:
17.  - $HOME/.m2
```

(a) `.travis.yml`

```
1. gradle --no-parallel --no-daemon --no-build-cache
   --no-watch-fs --no-configure-on-demand build
2. mvn -T 1 install
```

(b) `build.sh`

Figure 1: Example Excerpt of Travis CI Configuration

- We created a catalog of 20 performance-related CI configuration smells in three tools of the CI infrastructure for Java projects.
- We proposed an automated approach, BuildSonic, to detect and repair 15 types of performance-related CI configuration smells.
- We conducted large-scale experiments with 4,140 Java projects to demonstrate the effectiveness and efficiency of BuildSonic.

2 PRELIMINARIES

As we focus on configuration smells, we briefly introduce the configuration files of the three tools (see Sec. 3.1) selected in this work.

Travis CI. In Travis CI, a build is a group of jobs, and a job is an automated process that clones a repository into a virtual environment and carries out a series of phases such as compiling code and executing tests. The configuration file of Travis CI is `.travis.yml`. An example excerpt is presented in Fig. 1a. It defines the language support, two jobs that run with different JDK versions, and the `script` phase that invokes the shell script of the Maven and Gradle command. The shell script can also be defined in a file (e.g., `build.sh` in Fig. 1b).

Maven. The configuration file of the automated build tool Maven is `pom.xml`. Fig. 2 presents an example excerpt of Maven configuration. It declares the project information, two modules of the project, repositories where dependencies can be retrieved, dependencies used in the project, and two plugins that carry out the task of compilation (i.e., the `compiler` plugin) and testing (i.e., the `surefire` plugin).

Gradle. The work that Gradle performs on a project is defined by one or more tasks. A task represents an atomic piece of work such as compiling some classes and executing some tests. Gradle has three configuration files, i.e., `settings.gradle`, `gradle.properties` and `build.gradle`. An example excerpt is shown in Fig. 3. It defines the project hierarchy (i.e., two modules) in `settings.gradle`. It specifies the execution behavior of Gradle in key-value pairs (e.g., whether to enable cache) in `gradle.properties`. It declares repositories where dependencies can be retrieved, dependencies used in the project, test configuration, and configuration of tasks of a specific type (i.e., the `JavaCompile` and `Test` type).

```xml
<groupId>com.example</groupId>
<artifactId>maven-example</artifactId>
<version>1.0.0</version>
<packaging>pom</packaging>
<modules> <!-- two modules-->
  <module>sub-a</module>
  <module>sub-b</module>
</modules>
<repositories> <!--repository declaration-->
  <repository>
    <id>codehausSnapshots</id>
    <name>Codehaus Snapshots</name>
    <url>http://snapshots.maven.codehaus.org/maven2</url>
  </repository>
  <repository>
    <id>central</id>
    <name>Maven Repository Switchboard</name>
    <url>https://repo1.maven.org/maven2</url>
  </repository>
</repositories>
<dependencies> <!--dependency declaration-->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.9.0</version>
      <configuration>
        <fork>false<fork>
        <useIncrementalCompilation>false</useIncrementalCompilation>
        <maxmem>512m</maxmem>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.surefire</groupId>
      <artifactId>surefire</artifactId>
      <version>2.18.1</version>
      <configuration>
        <parallel>false</parallel>
        <forkCount>1</forkCount>
        <disableXmlReport>false</disableXmlReport>
        <excludes>
          <exclude>**/Bar.class</exclude>
        </excludes>
        </configuration>
     </plugin>
  </plugins>
<build>
```

Figure 2: Example Excerpt of Maven Configuration (`pom.xml`)

3 METHODOLOGY

We first introduce our process of creating a catalog of performance-related CI configuration smells (PCSs), and then elaborate the details of our approach BuildSonic to detect and repair PCSs.

3.1 Creating PCS Catalog

Collection Scope. There are many CI tools (e.g., Travis CI and Jenkins) available to build the CI infrastructure so that developers are enabled to customize CI pipelines. According to a study of 34,544 open-source projects on GitHub, 40% of the projects adopt CI, and 90% of them adopt Travis CI as their CI tool [28]. To be beneficial to wide audiences, our work is focused on Travis CI. Apart from the CI tool, the tool chain of the CI infrastructure contains specialized tools to automate the build process (e.g., dependency installation, source code compilation, static analysis and test case execution). This tool chain depends on the programming language used in a project. We decide to focus on Java in this work because it is widely used, and choose two automated build tools, Maven and Gradle, because they are the two most dominant tools for Java [42]. In summary, we define our scope to one CI tool, Travis CI, and two automated build tools, Maven and Gradle, for Java projects. We will extend our scope to other CI tools, automated build tools and programming languages in future.

```
rootProject.name = 'gradle-example' // project name
include 'sub-a', 'sub-b' // two modules
```

(a) `settings.gradle`

```
org.gradle.parallel = false
org.gradle.daemon = false
org.gradle.caching = false
org.gradle.vfs.watch = false
org.gradle.configureondemand = false
org.gradle.jvmargs = -Xmx1024M
```

(b) `gradle.properties`

```
repositories { // repository declaration
    mavenCentral()
    maven {
        url "https://repo.spring.io/release"
    }
}
dependencies { // dependency declaration
  implementation 'junit:junit:4.8.2'
}
test {
  exclude '**/Bar.class'
}
tasks.withType(JavaCompile).configureEach {
    options.fork = false
    options.incremental = false
}
tasks.withType(Test).configureEach {
    maxParallelForks = 1
    forkEvery = 0
    reports.html.required = true
    reports.junitXml.required = true
}
```

(c) `build.gradle`

Figure 3: Example Excerpt of Gradle Configuration

Collection Process. We define PCSs as misconfigurations of the CI infrastructure, which potentially violate best CI practices [14, 29] and hinder the performance of CI builds. Similar to previous approaches [20, 66], we collect PCSs by analyzing tool documentations. Specifically, two of the authors separately read documentations of Travis CI [11], Maven [44] and Gradle [22], look for configuration options that can affect the build performance, and record how they affect the build performance, following an open coding procedure [57]. Then, they discuss the collected configuration options together to derive a catalog of PCSs that misconfigure these options, and a third author is involved to resolve disagreements and reach consensus. It is worth mentioning that our manual effort, involved in our PCS collection process, requires about three man-months.

Our Catalog. Following the above process, we finally create a catalog of 20 PCSs, as listed in the first column of Table 1. The rationale that we regard it as a PCS will be introduced in Sec. 3.2 with the approach to detect and repair the PCS. As shown by the second column of Table 1, Travis CI, Maven and Gradle respectively have 6, 9 and 14 of the 20 types of PCSs. Maven and Gradle share 9 types of PCSs. Moreover, we explore whether these PCSs are covered in literature [13, 20, 66, 77]. The result is reported in the third column of Table 1. Specifically, only three types of PCSs (marked by ✓) have been covered in literature and their detection or repairing techniques have been proposed. Only two types of PCSs (marked by ◯) have been covered in literature but no detection and repairing techniques have been proposed for them. The remaining 15 types of PCSs (marked by ×) are uncovered in literature. Therefore, our work fills this gap.

Table 1: Our Catalog of PCSs (Lit. = Literature, Exp. = Explicit, Imp. = Implicit, Det. = Detection, and Rep. = Repairing)

PCS	Tool	Lit.	Exp.	Imp.	Det.	Rep.
Deep Clone	Travis CI	×	✓	×	✓	◯
No Dependency Cache	Travis CI	×	✓	✓	✓	◯
Retry Failed Command	Travis CI	✓	✓	×	✓	✓
Wait Long Command	Travis CI	×	✓	×	✓	✓
Slow Finish	Travis CI	×	✓	×	✓	✓
Heavy Job	Travis CI	×	✓	×	×	×
Sequential Build	Maven, Gradle	◯	✓	✓	✓	◯
No Compiler Daemon	Maven, Gradle	×	✓	✓	✓	✓
Sequential Test	Maven, Gradle	◯	✓	✓	✓	◯
Non-Fork Test	Maven, Gradle	×	✓	✓	✓	◯
Generate Test Report	Maven, Gradle	×	✓	✓	✓	✓
Improper Repository	Maven, Gradle	×	✓	×	×	×
Unused Dependency	Maven, Gradle	✓	✓	×	×	×
Unnecessary Test	Maven, Gradle	✓	✓	×	×	×
Small Heap Size	Maven, Gradle	×	✓	✓	×	×
Non-Incremental Compile	Gradle	×	✓	✓	✓	✓
No Gradle Daemon	Gradle	×	✓	✓	✓	✓
No Gradle Cache	Gradle	×	✓	✓	✓	✓
No File System Watch	Gradle	×	✓	✓	✓	✓
No On-Demand Configure	Gradle	×	✓	✓	✓	✓

3.2 Detecting and Repairing PCSs

We propose and design BuildSonic as a *linter* that can statically and efficiently detect and repair PCSs by mainly analyzing configuration files. Five of the 20 types of PCSs (marked by × in the last two columns of Table 1) are not supported in BuildSonic because heavyweight analysis (e.g., program analysis) is needed to detect them and hence the principle (i.e., to be efficient) of a linter is violated. For the other 15 types of PCSs, BuildSonic can automatically detect all of them and automatically repair ten of them (marked by ✓ in the last two columns of Table 1), and repair five of them semi-automatically (marked by ◯ in the last column of Table 1) by allowing developers to set a specific value (e.g., the number of parallel threads) when there exist multiple feasible values.

Further, we distinguish whether a PCS can be explicitly and/or implicitly introduced according to default configurations. We consider a PCS as explicitly introduced if developers manually set configuration options (by using or changing the default value). It means that developers are aware of the configuration options but might still introduce PCSs. We regard a PCS as implicitly introduced if developers do not manually set configuration options but adopt their default configurations. It means that developers might not be aware of the configuration options, but the default configurations introduce PCSs. The results are reported in the fourth and fifth columns of Table 1.

Below, for each PCS, we first present our rationale of regarding it as a PCS, and then introduce the approach to detect and repair it.

PCS01: Deep Clone.

Rationale. Travis CI needs to clone repository of a project to the build server after a CI build is triggered. As a project may have a long commit history, the clone may take a long time. To address this issue, Travis CI provides the configuration option `depth`, as shown at Line 14 in Fig. 1a, to control the clone depth (i.e., to clone a limited depth of `depth` commits). A smaller value of `depth` indicates a faster clone. However, restarting historical CI builds on old commits will fail if the old commits are outside of the clone depth. Hence, Travis CI recommends to set `depth` to 50, which is its default value. However, developers may set `depth` to a large value, and introduce a PCS.

Detection. The configuration option `depth` is enabled by default. Hence, this PCS can only be explicitly introduced. We detect a *Deep Clone* if `.travis.yml` contains the configuration option `depth` and its value is set to `false` (i.e., all commits) or a larger value than 50.

Repairing. We repair this PCS by removing `depth` (i.e., using the default one) or asking developers to set `depth` to a smaller value.

PCS02: No Dependency Cache.

Rationale. A project often directly and transitively depends on a lot of library dependencies [69]. As a result, a CI build may take a long time to download and install the dependencies [8]. Moreover, the dependencies that a project depends on are not often changed, and thus it wastes time to download and install dependencies in every CI build. To solve this issue, Travis CI provides the configuration option `cache`, as illustrated at Lines 15–17 in Fig. 1a, to cache dependencies in specified directories for Maven and Gradle projects. When developers do not enable this configuration option, they introduce a PCS.

Detection. The configuration option `cache` is disabled by default for Maven and Gradle projects. Hence, this PCS can be explicitly or implicitly introduced. We detect an explicit *No Dependency Cache* if `.travis.yml` contains the configuration option `cache` and its value is set to `false`; and we detect an implicit one if `.travis.yml` does not contain the configuration option `cache`.

Repairing. We repair this PCS by enabling the configuration option `cache` and specifying `directories`. For Maven projects, we set `directories` to `$HOME/.m2`. For Gradle projects, we set `directories` to `$HOME/.gradle/caches/` and `$HOME/.gradle/wrapper/`. These above directories are the default ones. As developers can choose arbitrary directories as the cache directories, we allow developers to set `directories` to their own specified directories.

PCS03: Retry Failed Command.

Rationale. The build process should be deterministic. However, some non-deterministic behaviors, e.g., flaky tests [39] and unstable network, make a CI build sometimes fail but sometimes pass. To partially mitigate this problem, Travis CI allows developers to wrap a command in the `travis_retry` function such that the command, if failed, can be re-run up to two times (e.g., Line 10 in Fig. 1a), or to add the `----retry n` option to a command such that the command, if failed, can be re-run up to `n-1` times. In fact, this configuration slows down CI builds, while only reducing the build failure ratio by 3% according to a recent study [21]. Moreover, this configuration may not only hide the non-determinism and the potentially deeper underlying issues, but also make issues harder to debug [66]. In that sense, developers introduce a PCS when they configure this retry feature.

Detection. The retry feature is not enabled by default. Thus, *Retry Failed Command* can only be explicitly introduced. We detect it if commands in `.travis.yml` and shell script file contain the `travis_retry` function, or contain the `----retry n` option with `n` larger than one.

Repairing. We repair this PCS by removing the `travis_retry` function or the `----retry n` option in the detected commands.

PCS04: Wait Long Command.

Rationale. One of the best CI practices is that a CI build provides feedback as quickly as possible (often in under 10 minutes) [13]. To follow this practice, Travis CI will interrupt a CI build and mark it as failed when a command in this CI build takes longer than 10 minutes without producing any output. To enable a long running command, Travis CI allows develops to wrap the command in the `travis_wait n` function so that the waiting time for the command to finish is extended to `n` minutes, as illustrated at Line 11 in Fig. 1a. If `n` is not specified, the waiting time is extended to 20 minutes. Although this feature improves flexibility, it hinders quick feedback, and potentially hides the underlying real problems in long running commands. In that sense, a PCS is introduced when developers use this feature.

Detection. The wait feature is not enabled by default. Hence, this PCS can only be explicitly introduced. We detect a *Wait Long Command* if commands in `.travis.yml` contain the `travis_wait n` function, and `n` is not specified or `n` is larger than 10.

Repairing. We repair this PCS by removing the `travis_wait n` function in the detected commands.

PCS05: Slow Finish.

Rationale. A CI build is marked as passed if all jobs are passed. However, developers may want to add in some experimental and preparatory jobs (e.g., to test against runtime versions that developers are not ready to officially support) whose failure does not matter. Thus, Travis CI provides the configuration option `allow_failures` to define the jobs that are allowed to fail without causing the entire CI build to fail (e.g., Lines 7–8 in Fig. 1a indicate that the job running with a JDK version of `openjdk-ea` is allowed to fail). If some jobs are allowed to fail, the CI build will not be marked as finished until they have finished, causing the long build time and slow feedback. To address this issue, Travis CI provides the configuration option `fast_finish` (e.g., Line 6 in Fig. 1a) such that the CI build is marked as finished as soon as all the required jobs finish, while the other jobs that are allowed to fail continue to run. Therefore, when developers configure `allow_failures` but do not set `fast_finish` to `true`, they introduced a PCS.

Detection. As both configuration options `allow_failures` and `fast_finish` are not enabled by default, *Slow Finish* can only be explicitly introduced. We detect it if `.travis.yml` contains the configuration option `allow_failures`, but does not contain the configuration option `fast_finish` or `fast_finish` is set to `false`.

Repairing. We repair this PCS by adding the configuration option `fast_finish` and/or setting it to `true`.

PCS06: Heavy Job.

Rationale. In Travis CI, one virtual machine is created for each job in a CI build. Therefore, multiple jobs can run in parallel across virtual machines. To speed up a CI build, it can be reconfigured by splitting up independent tasks into different jobs instead of adding them into one job. For example, unit testing and integration testing can be split up into two jobs; and testing and static analysis can be split up into two jobs. Developers may introduce a PCS when they add many independent tasks into one job because they may miss the efficiency benefit of parallel jobs across virtual machines.

Detection & Repairing. This PCS can only be explicitly introduced because heavy jobs are defined by developers. It is difficult to decide whether a job contains independent tasks by only analyzing configuration files. For example, testing code analysis is also needed to split up unit testing and integration testing. Therefore, we cannot provide lightweight detection and repairing support for this PCS.

PCS07: Sequential Build.

Rationale. Modular development can reduce development complexity and risk and make code easier to write, test and read. Hence, multi-module projects are quite common. Modules in a multi-module project can be potentially built in parallel rather than sequentially to speed up builds. Maven and Gradle build all modules sequentially by default, but can be configured to enable parallel builds. Particularly, Maven allows developers to add the `--T n` option to the `mvn` build command, where `n` specifies the number of parallel threads, as shown at Line 2 in Fig. 1b. Gradle allows developers to set the configuration option `org.gradle.parallel` to `true`/`false` (e.g., Line 1 in Fig. 3b) or add the `----parallel`/`----no-parallel` option to the `gradle` build command (e.g., Line 1 in Fig. 1b) to enable/disable the build of modules in parallel in one virtual machine. Developers introduce a PCS when they do not enable parallel builds.

Detection. Parallel builds are not enabled by default, and thus this PCS can be both explicitly and implicitly introduced. We first decide whether a project is multi-module by analyzing the `modules` section (e.g., Lines 5–8 in Fig. 2) in `pom.xml` and `include` section (e.g., Line 2 in Fig. 3a) in `settings.gradle`. If yes, we then detect an explicit *Sequential Build* in a Maven project if the `mvn` command in `.travis.yml` and shell script file contains the `--T 1` option, and detect an implicit one if the `mvn` command does not contain the `--T n` option. We detect an explicit *Sequential Build* in a Gradle project if the configuration option `org.gradle.parallel` in `gradle.properties` is set to `false`, or the `gradle` command in `.travis.yml` and shell script file contains the `----no-parallel` option, and detect an implicit one if `gradle.properties` does not contain the configuration option `org.gradle.parallel`, or the `gradle` command does not contain the `----no-parallel` and `----parallel` option.

Repairing. We repair this PCS in a Maven project by adding the `--T n` option in the `mvn` command and/or asking developers to decide `n`. We repair it in a Gradle project by setting the configuration option `org.gradle.parallel` to `true`, or adding the `----parallel` option and removing the `----no-parallel` option in the `gradle` command.

PCS08: No Compiler Daemon.

Rationale. The compilation of a large number of source code files is memory-intensive. Maven and Gradle provide a compiler daemon feature to allow running the compiler in a separate process. It leads to much less garbage collection in the main build daemon, making the build run faster. To configure this feature, Maven provides the configuration option `fork` in the `compiler` plugin (e.g., Line 36 in Fig. 2), and Gradle provides the configuration option `options.fork` in the `JavaCompile` task (e.g., Line 14 in Fig. 3c). When developers do not enable compiler daemon for a large project, they introduce a PCS.

Detection. Compiler daemon is disabled by default, and thus this PCS can be both explicitly and implicitly introduced. We first decide whether a project is large by counting whether the number of source code files is larger than 1,000, as suggested in Gradle's documentation [22]. If yes, we then detect an explicit *No Compiler Daemon* in a Maven project if the configuration option `fork` in the `compiler` plugin in `pom.xml` is set to `false`, and detect an implicit one if the

compiler plugin does not set fork. We detect an explicit *No Compiler Daemon* in a Gradle project if options.fork in the JavaCompile task in build.gradle is set to false, and detect an implicit one if the JavaCompile task does not set options.fork.

Repairing. We repair this PCS in a Maven/Gradle project by setting the configuration option fork/options.fork to true.

PCS09: Sequential Test.

Rationale. A large proportion of the build time is taken by test execution. To speed up tests, Maven and Gradle provide a parallel test execution feature to run parallel tests across threads in a single virtual machine process. Specifically, this feature is supported by the configuration option parallel in the surefire plugin in Maven (e.g., Line 46 in Fig. 2) and the configuration option maxParallelForks in the Test task in Gradle (e.g., Line 18 in Fig. 3c). Developers introduce a PCS when they do not enable parallel test execution.

Detection. Parallel test execution is disabled by default. Thus, this PCS can be both explicitly and implicitly introduced. We detect an explicit *Sequential Test* in a Maven project if i) the configuration option parallel in the surefire plugin in pom.xml is set to false or none, or ii) parallel is not set to false or none, useUnlimitedThreads (indicating that one thread is created per CPU core) is not set or set to false, threadCount (indicating the number of created threads) is set to 1 and perCoreThreadCount (indicating that threadCount is defined per CPU core) is set to false, and detect an implicit one if the surefire plugin does not configure parallel. We detect an explicit *Sequential Test* in a Gradle project if the configuration option maxParallelForks in the Test task in build.gradle is set to 1, and detect an implicit one if the Test task does not set maxParallelForks.

Repairing. We repair this PCS in a Maven project by setting parallel to classes and useUnlimitedThreads to true, or asking developers to configure parallel to the granularity of methods, classes, suites or a combination of them and determine thread configuration. We repair it in a Gradle project by asking developers to set maxParallelForks to a larger value than 1.

PCS10: Non-Fork Test.

Rationale. Apart from parallel tests across threads, Maven and Gradle support parallel tests across processes via forking to avoid potentially heavy garbage collection that slows the tests down. Particularly, this feature is supported by the configuration option forkCount (i.e., the number of forked processes) in the surefire plugin in Maven (e.g., Line 47 in Fig. 2) and the configuration option forkEvery (i.e., the maximum number of test classes to execute in a forked process) in the Test task in Gradle (e.g., Line 19 in Fig. 3c). A PCS is introduced when developers do not enable process-level parallel tests.

Detection. Parallel test execution across processes is disabled by default, and hence this PCS can be both explicitly and implicitly introduced. We detect an explicit *Non-Fork Test* in a Maven project if the configuration option forkCount in the surefire plugin in pom.xml is set to 1, and detect an implicit one if the surefire plugin does not configure forkCount. We detect an explicit *Non-Fork Test* in a Gradle project if the configuration option forkEvery in the Test task in build.gradle is set to 0, and detect an implicit one if the Test task does not configure forkEvery.

Repairing. We repair this PCS in a Maven/Gradle project by asking developers to set the configuration option forkCount/forkEvery.

PCS11: Generate Test Report.

Rationale. Test reports are automatically generated by default in both Maven and Gradle, which takes time and slows the entire build down. However, developers may only focus on whether the tests succeed or fail but not want to look at the reports, or only need the reports for some specific builds but not every build. In that sense, developers may introduce a PCS if they enable test report generation.

Detection. Test report generation is enabled by default. Thus, this PCS can be both explicitly and implicitly introduced. We detect an explicit *Generate Test Report* in a Maven project if the configuration option disableXmlReport in the surefire plugin in pom.xml is set to false, and detect an implicit one if the surefire plugin does not configure disableXmlReport. We detect an explicit *Generate Test Report* in a Gradle project if the configuration option reports.html.required or reports.junitXml.required in the Test task in build.gradle is set to true, and detect an implicit one if the Test task does not configure reports.html.required and reports.junitXml.required.

Repairing. We repair it in a Maven and Gradle project by setting disableXmlReport to true and setting reports.html.required and reports.junitXml.required to false.

PCS12: Improper Repository.

Rationale. To resolve a dependency, Maven and Gradle search each repository in the order that they are declared in the repositories section in pom.xml and build.gradle (e.g., Lines 9–20 in Fig. 2 and Lines 1–6 in Fig. 3c) until they find the dependency. Hence, the repository that hosts the largest number of the declared dependencies should be declared first to reduce dependency resolution time; otherwise, a PCS might be introduced due to the improper order of repositories or the improper selection of repositories.

Detection & Repairing. There exist many repositories to host dependencies. As a result, it is difficult to determine whether the selection and order of declared repositories are reasonable by looking at the configuration files. Hence, we cannot provide lightweight detection and repairing support for this PCS.

PCS13: Unused Dependency.

Rationale. It is common that some dependencies are declared in the dependencies section in pom.xml and build.gradle (e.g., Lines 21–28 in Fig. 2 and Lines 7–9 in Fig. 3c), but are not used in a Maven and Gradle project [61, 62]. This leads to wasted dependency resolution time and slows the build down, and thus introduces a PCS.

Detection & Repairing. It is difficult to detect unused dependencies by static analysis due to dynamic features like reflection, not to mention by analyzing configuration files. Therefore, we cannot provide lightweight detection and repairing support for this PCS.

PCS14: Unnecessary Test.

Rationale. Code changes in a build can be small, and only affect a small number of tests. Therefore, only the tests that are affected by code changes but not all tests need to be executed in each build so as to speed up test execution. Maven and Gradle provide the configuration option exclude (e.g., Line 11 in Fig. 3c and Line 50 in Fig. 2) to exclude such unnecessary tests from execution. Developers may introduce a PCS when they do not exclude unnecessary tests.

Detection & Repairing. Test case selection [40, 45, 60] has been proposed to select test cases that are affected by code changes. However, such techniques often involve program analysis. Hence, we cannot provide lightweight detection and repairing support for this PCS.

PCS15: Small Heap Size.

Rationale. By default, Maven and Gradle reserve 512 MB and 1024 MB of heap space for a build. The heap size can be changed by the configuration option `maxmen` in Maven (e.g., Line 38 in Fig. 2) and the configuration option `org.gradle.jvmargs` in Gradle (e.g., Line 6 in Fig. 3b). However, large projects might need more memory than the default size to build. In such cases, the build might become slow if the heap size is not increased, and thus a PCS is introduced.

Detection & Repairing. Dynamic techniques like memory profiling are needed to decide a suitable heap size. Therefore, we cannot provide lightweight detection and repairing support for this PCS.

PCS16: Non-Incremental Compile.

Rationale. Besides compiler daemon, Gradle supports an incremental compilation feature to recompile only the classes that are affected by a changed class) so as to make compilation run faster. The configuration option `options.incremental` in the `JavaCompile` task, as illustrated at Line 15 in Fig. 3c, is used to configure this feature. A PCS is introduced when developers do not enable incremental compilation in a Gradle project.

Detection. Incremental compilation is enabled by default since Gradle 4.10. Thus, this PCS can be explicitly introduced in all Gradle versions. It can also be implicitly introduced in lower versions than Gradle 4.10. We detect an explicit *Non-Incremental Compile* if the configuration option `options.incremental` in the `JavaCompile` task in `build.gradle` is set to `false`, and detect an implicit one if the Gradle version is lower than 4.10 and the `JavaCompile` task does not configure `options.incremental`.

Repairing. We repair this PCS by setting the configuration option `options.incremental` to `true`.

PCS17: No Gradle Daemon.

Rationale. Gradle runs on the JVM whose startup is expensive. To avoid the cost of JVM startup for every build, Gradle provides the Gradle daemon, a long-lived process, to start the JVM for once across multiple builds. Besides, it is able to cache information about project structure, files and tasks in memory to speed up builds. It can be configured via the configuration option `org.gradle.daemon` (e.g., Line 2 in Fig. 3b) and the command option `----daemon`/`----no-daemon` (e.g., Line 1 in Fig. 1b). A PCS is introduced when developers do not enable it.

Detection. The Gradle daemon is enabled by default since Gradle 3.0. Thus, this PCS can be explicitly introduced in all Gradle versions. It can also be implicitly introduced in lower versions than Gradle 3.0. We detect an explicit *No Gradle Daemon* if the configuration option `org.gradle.daemon` in `gradle.properties` is set to `false`, or the `gradle` command in `.travis.yml` and shell script file adds the `----no-daemon` option. We detect an implicit one if the Gradle version is lower than 3.0, and `gradle.properties` does not contain the configuration option `org.gradle.daemon` or the `gradle` command does not contain the `----no-daemon` and `----daemon` option.

Repairing. We repair this PCS by setting the configuration option `org.gradle.daemon` to `true`, or adding the `----daemon` option and removing the `----no-daemon` option in the `gradle` command.

PCS18: No Gradle Cache.

Rationale. Different from the dependency cache mechanism in Travis CI, Gradle provides the build cache mechanism to save time by reusing outputs produced by previous builds. It works by fetching stored build outputs from the cache if builds inputs have not changed. The expensive output regeneration can thus be avoided. It can be set via the configuration option `org.gradle.caching` (e.g., Line 3 in Fig. 3b) and the command option `----build-cache`/`----no-build-cache` (e.g., Line 1 in Fig. 1b). A PCS is introduced when it is not enabled.

Detection. This mechanism is disabled by default. Thus, this PCS can be both explicitly and implicitly introduced. We detect an explicit *No Gradle Cache* if the configuration option `org.gradle.caching` in `gradle.properties` is set to `false`, or the `gradle` command in `.travis.yml` and shell script file adds the `----no-build-cache` option. We detect an implicit one if `gradle.properties` does not set the configuration option `org.gradle.caching` or the `gradle` command does not contain `----no-build-cache` and `----build-cache`.

Repairing. We repair this PCS by setting the configuration option `org.gradle.caching` to `true`, or adding `----build-cache` and removing `----no-build-cache` in the `gradle` command.

PCS19: No File System Watch.

Rationale. Gradle maintains a virtual file system (VFS) in-memory for each build. To save the time to rebuild the VFS from disk for the next build, Gradle watches the file system such that it can keep the VFS in sync with the file system between builds. This feature can be configured by the configuration option `org.gradle.vfs.watch` (e.g., Line 4 in Fig. 3b) and the command option `----watch-fs`/`----no-watch-fs` (e.g., Line 1 in Fig. 1b). A PCS is introduced when it is not enabled.

Detection. File system watch is enabled by default since Gradle 7.0. Thus, this PCS can be explicitly introduced in all Gradle versions. It can also be implicitly introduced in lower versions than Gradle 7.0. We detect an explicit *No File System Watch* if the configuration option `org.gradle.vfs.watch` in `gradle.properties` is set to `false`, or the `gradle` command in `.travis.yml` and shell script file adds the `----no-watch-fs` option. We detect an implicit one if the Gradle version is lower than 7.0, and `gradle.properties` does not contain `org.gradle.vfs.watch` or the `gradle` command does not contain the `----no-watch-fs` and `----watch-fs` option.

Repairing. We repair this PCS by setting the configuration option `org.gradle.vfs.watch` to `true`, or adding `----watch-fs` and removing `----no-watch-fs` in the `gradle` command.

PCS20: No On-Demand Configure.

Rationale. Every module in a multi-module projects is configured although only some of the modules participate in the build. To reduce this configuration time, Gradle provides an on-demand configuration feature to configure only the modules that are involved in the build. This feature can be configured by the configuration option `org.gradle.configureondemand` (e.g., Line 4 in Fig. 3b) and the command option `----configure-on-demand`/`----no-configure-on-demand` (e.g., Line 1 in Fig. 1b). A PCS is introduced when it is not enabled.

Detection. On-demand configuration is disabled by default, and thus this PCS can be both explicitly and explicitly introduced. We first decide whether a project is multi-module (by the same way in *Sequential Build*). If yes, we then detect an explicit *No On-Demand Configure* if the configuration option `org.gradle.configureondemand` in `gradle.properties` is set to `false`, or the `gradle` command in `.travis.yml` and shell script file adds `----no-configure-on-demand`. We detect an implicit one if `gradle.properties` does not contain `org.gradle.configureondemand` or the `gradle` command does not contain `----no-configure-on-demand` and `----configure-on-demand`.

Repairing. We repair it by setting `org.gradle.configureondemand` to `true`, or adding `----configure-on-demand` in the `gradle` command while removing `----configure-on-demand`.

4 IMPLEMENTATION AND EVALUATION

We have implemented BuildSonic in 11.2K lines of code, i.e., 7.6K lines of Groovy code, 2,1K lines of Java code, 0.9K lines of Python code and 0.6K lines of Ruby code. Overall, taking a project as the input, BuildSonic is implemented by first parsing configuration files in the project to extract related configuration options and their configured values, then detecting the PCSs in the project by the approaches in Sec. 3.2 based on the extracted information, and finally updating the configuration files by the approaches in Sec. 3.2 to repair the PCSs.

In detail, we use `SnakeYAML` to parse `.travis.yml`, use `VTD-XML` to parse `pom.xml`, use Groovy's syntax parser to parse `settings.gradle` and `build.gradle` (which are often written in Groovy), use `Apache Commons Configuration` to parse `gradle.properties`, and use `Tree-sitter` to parse shell script files to extract Maven and Gradle commands. Besides, as the version of Maven and Gradle is used in detecting and repairing some PCSs, we extract this version information from the `maven-wrapper.properties` file in a Maven project and the `gradle-wrapper.properties` file in a Gradle project.

We have released the source code of BuildSonic and all the evaluation data at https://buildsonic.github.io.

4.1 Evaluation Setup

Research Question. To evaluate the effectiveness and efficiency of BuildSonic, we designed experiments to answer five RQs.

- **RQ1**: How prevalent are PCSs in the CI infrastructure?
- **RQ2**: How accurate is BuildSonic in detecting/repairing PCSs?
- **RQ3**: Are developers willing to repair PCSs?
- **RQ4**: How much build time is reduced after repairing PCSs?
- **RQ5**: How efficient is BuildSonic in detecting/repairing PCSs?

Data Set. We followed the method proposed by Zhang et al. [78] to construct the data set of open-source Java projects. We first crawled the list of non-forked open-source Java projects as of June 5, 2020 via the GitHub API, which resulted in 7,328,086 projects. To ensure the quality and frequent Travis CI usage, we chose the projects that had more than 25 stars and 50 Travis CI builds, which restricted our selection to 5,606 projects. We further filtered projects that did not contain `.travis.yml`, which led to 4,329 projects. Of these projects, we only kept the projects that used Maven or Gradle as the automated build tool. This resulted in 2,547 projects using Maven and 1,622 projects using Gradle. Configuration files of Gradle can be written in Groovy or Kotlin, and we found that 1,593 of the 1,622 projects used Groovy and only 29 of the 1,622 projects used Kotlin. Due to the small usage of Kotlin, our current implementation of BuildSonic only supports Groovy, and we only kept the 1,593 projects that used Groovy. Finally, we had a set of 4,140 projects, i.e., 2,547 (61.5%) Maven projects and 1,593 (38.5%) Gradle projects, and crawled their repositories.

4.2 Prevalence Analysis (RQ1)

We analyzed the prevalence of the 15 types of PCSs in the wild by running BuildSonic against the 4,140 projects. The number of explicitly and implicitly introduced PCSs are respectively reported under the column *Exp.* and *Imp.* in Table 2. A project can contain multiple instances of a PCS type, and we counted them as one in our analysis.

Overall, BuildSonic detected 484 explicitly introduced PCSs across all the 15 PCS types except for *No Compiler Daemon*, affecting 426 (10.3%) of the projects. BuildSonic also detected 19,834 implicitly introduced PCSs across all the 11 PCS types, affecting 4,096 (98.9%) of the projects. These results indicate that all PCS types are quite prevalent in the wild, and PCSs are often implicitly introduced by default configurations, which motivates the need of BuildSonic to automatically detect them. Moreover, for the five, five and ten PCS types in Travis CI, Maven and Gradle, BuildSonic detected 2,864, 7,920 and 9,534 PCSs respectively, affecting 2,769 (66.9%) of the projects, 2,332 (91.6%) of the Maven projects and 1,592 (99.9%) of the Gradle projects. These results indicate that PCSs are prevalent in all three tools. Besides, 209 (5.0%), 112 (2.7%), 448 (10.8%), 1,203 (29.1%) and 2,126 (51.4%) of the projects respectively contained one, two, three, four and more than four PCSs. On average, each project contained 5 PCSs.

Summary. We found a total of 20,318 PCSs, covering all the 15 types of PCSs and affecting 99.0% of the projects. Each project contained an average of 5 PCSs. Therefore, PCSs are prevalent in the wild.

4.3 Accuracy Evaluation (RQ2)

To evaluate the detection precision of BuildSonic, we randomly sampled 50 detected PCSs for each PCS type while distinguishing explicitly and implicitly introduced PCSs. If the number of detected PCSs was smaller than 50, we sampled all of them. We finally sampled 371 explicitly introduced PCSs and 800 implicitly introduced PCSs, as reported under the column *Sam.* in Table 2. These sample sizes achieved an error margin of ±2.5% and ±3.4% with a confidence level of 95%. Then, two of the authors individually validated these PCSs, and a third author was involved to resolve disagreements. Finally, we found two false positives, as listed under the column *F.P.* in Table 2, and achieved a precision of 0.998. The reason of the false positive for *Sequential Build* is that the white space in the `--T n` option can be skipped but our detection is not aware of that. The reason of the false positive for *Sequential Test* is that the configuration option `maxParallelForks` is dynamically configured in an `echo` command.

Moreover, to evaluate the detection recall of BuildSonic, we randomly sampled 200 projects. Two of the authors individually analyzed the configuration files to locate PCSs, and a third author was involved to resolve disagreements. We eventually identified 983 PCSs, covering 15 PCS types. BuildSonic detected 974 of them, achieving a recall of 0.991. One project uses both Maven and Gradle, but BuildSonic detects it as a Maven project without further checking whether it is a Gradle project, causing six false negatives. The other three false negatives are caused by the dynamic configuration of Gradle projects.

Table 2: Results of Detection, Accuracy and Pull Requests (Exp. = Explicit, Imp. = Implicit, Sam. = Sampled, F.P. = False Positive, P.R. = Pull Request, Mer. = Merged, Acc. = Accepted, Rej. = Rejected, Ign. = Ignored, Pen. = Pending, and P.F. = Positive Feedback)

PCS	Tool	Exp.	Sam.	F.P.	P.R.	Mer.	Acc.	Rej.	Ign.	Pen.	P.F.	Imp.	Sam.	F.P.	P.R.	Mer.	Acc.	Rej.	Ign.	Pen.	P.F
Deep Clone	Travis CI	63	50	0	50	9	0	7(12)	1	21	52.9	–	–	–	–	–	–	–	–	–	–
No Dependency Cache	Travis CI	20	20	0	15	1	0	0(1)	0	13	100	2,604	50	0	50	18	2	1(16)	2	11	87.0
Retry Failed Command	Travis CI	41	41	0	34	6	0	2(6)	2	18	60.0	–	–	–	–	–	–	–	–	–	–
Wait Long Command	Travis CI	80	50	0	50	10	0	8(9)	2	21	50.0	–	–	–	–	–	–	–	–	–	–
Slow Finish	Travis CI	56	50	0	48	16	0	2(15)	3	12	76.2	–	–	–	–	–	–	–	–	–	–
Sequential Build	Maven	0	–	–	–	–	–	–	–	–	–	1,088	50	1	49	11	0	6(8)	1	23	61.1
	Gradle	8	8	0	4	0	1	0(1)	0	2	100	808	50	0	50	14	0	1(3)	1	31	87.5
No Compiler Daemon	Maven	0	–	–	–	–	–	–	–	–	–	228	50	0	50	6	1	6(9)	2	26	46.7
	Gradle	0	–	–	–	–	–	–	–	–	–	75	50	0	50	8	1	6(5)	2	28	52.9
Sequential Test	Maven	9	9	0	8	0	1	0(0)	0	7	100	2,165	50	0	50	12	0	10(1)	1	26	52.2
	Gradle	15	15	0	8	1	0	0(0)	0	7	100	1,155	50	1	49	14	0	10(5)	4	16	50.0
Non-Fork Test	Maven	82	50		50	7	0	3(0)	0	40	70.0	2,122	50	0	50	4	0	5(7)	3	31	33.3
	Gradle	0	–	–	–	–	–	–	–	–	–	1,169	50	0	50	9	1	8(4)	4	24	45.5
Generate Test Report	Maven	6	6	0	4	1	0	0(0)	0	3	100	2,220	50	0	50	5	0	4(7)	1	33	50.0
	Gradle	0	–	–	–	–	–	–	–	–	–	1,192	50	0	50	14	0	11(5)	4	16	48.3
Non-Incremental Compile	Gradle	1	1	0	1	0	0	0(1)	0	0	–	757	50	0	50	7	1	1(3)	4	34	61.5
No Gradle Daemon	Gradle	82	50	0	50	1	2	5(1)	2	39	30.0	294	50	0	50	1	0	1(2)	2	44	25.0
No Gradle Cache	Gradle	2	2	0	1	0	0	0(0)	0	1	–	1,559	50	0	50	31	0	3(5)	2	9	86.1
No File System Watch	Gradle	1	1	0	1	0	0	1(0)	0	0	0	1,550	50	0	50	23	0	2(1)	1	23	88.5
No On-Demand Configure	Gradle	18	18	0	16	2	1	0(2)	0	11	100	848	50	0	50	15	0	1(3)	1	30	88.2
Overall		484	371	0	340	54	5	28(48)	10	195	60.8	19,834	800	2	798	192	6	76(84)	35	405	64.1

Except for the false positives and false negatives that are caused by dynamic configurations in Gradle projects, other false positives and false negatives can be removed by future iterations of BuildSonic.

Further, to evaluate the repairing accuracy of BuildSonic, we removed the two false positives from the 371 + 800 PCSs sampled in our detection precision analysis, and applied BuildSonic to repair the remaining 1,169 PCSs. Two of the authors individually validated the repaired configuration files, and a third author was involved to resolve disagreements. Finally, of the 669 PCSs that should be automatically repaired, BuildSonic correctly repaired all of them; and of the 500 PCSs that should be semi-automatically repaired, BuildSonic correctly located the configuration options where developer intervention was needed for all of them.

Summary. BuildSonic detected PCSs with a precision of 0.998 and a recall of 0.991 and repaired PCSs with a perfect accuracy.

4.4 Developer Feedback (RQ3)

To evaluate the practical usefulness of BuildSonic, we removed the false positives from the 371 + 800 PCSs sampled in our detection precision analysis (Sec. 4.3), applied BuildSonic to repair the other 1,169 PCSs (human intervention was involved in repairing the PCS types that were semi-automatically repairing by BuildSonic), and submitted repaired configuration files as pull requests. Finally, we submitted 1,138 pull requests, as listed under the column *P.R.* in Table 2. We did not submit pull requests for 31 PCSs because the projects were read-only on GitHub. At the time of writing, we have received developer feedback from 538 pull requests. We classified feedback into five categories, i.e., *merged* (a pull request has been merged by adopting our repair), *accepted* (a pull request has been accepted by confirming our detected PCSs), *rejected* (a pull request has been rejected with negative feedback, or due to irresistible factors, e.g., developers no longer adopt Travis CI as it has been no longer free since November 2, 2020, and developers regard us as a spam bot because we have submitted too many pull requests), *ignored* (a pull request has been closed without any feedback), and *pending* (a pull request is still open and under discussion). The result of each category is shown under the column *Mer.*, *Acc.*, *Rej.*, *Ign.* and *Pen.* in Table 2. The column *Rej.* is reported in the form of $a(b)$, where a denotes the ones rejected with negative feedback and b denotes the ones rejected due to irresistible factors. We measured a positive feedback rate (under the column *P.F.*) as the ratio of merged and accepted pull requests among all feedbacked pull requests except for the rejected ones due to irresistible factors.

Overall, 246 and 11 pull requests have been merged and accepted by developers, gaining a positive feedback rate of 63.3%. Developers commented that *"I'm not very familiar with travis and I'm not aware of this feature. Thanks for this PR"*, "*Thank you a lot for your contribution, this improved build time from 3 to 1 minute on my computer*", and "*This PR attend[s] to improve Build Performance. We can see the improvement here: ... So total improvement is 13.26%*".

Moreover, 104 pull requests have been rejected with negative feedback. One common reason is that developers think the build is fast and hence there is no need to further improve it. Inspired by this, we plan to improve BuildSonic by scanning projects only when their historical builds take a long time to run. Besides, for *Deep Clone*, the main reason of rejection is that the full commit history is required by static analysis tools like SonarCloud and SonarQube. Therefore, we plan to skip the detection of this PCS for projects that use such tools. For *Retry Failed Command* and *Wait Long Command*, the main reason of rejection is that developers have to use the retry and wait feature to deal with unstable behaviors and long running commands as it is difficult and expensive to fundamentally fix them. For *Sequential Build*, *Sequential Test* and *Non-Fork Test*, the main reason of rejection is that there are dependencies among modules or tests, and hence parallelism might cause failures, and it is also expensive to refactor modules or tests for enjoying the benefit of parallelism. For *No Compiler Daemon*, the main reason of rejection is the small performance improvement. We plan to increase the threshold of the number of

Table 3: Build Performance Change after Repairing PCSs (Suc. = Succeed, Pos. = Positive, and Neg. = Negative)

PCS	Tool	Suc. (#)	Fail (#)	Δ (%)	Pos. (#)	Pos. (%)	Neg. (#)	Neg. (%)
Deep Clone	Travis CI	43	0	7.6	43	7.6	0	0
No Dependency Cache	Travis CI	14	0	23.2	14	23.2	0	0
Retry Failed Command	Travis CI	11	0	33.9	11	33.9	0	0
Wait Long Command	Travis CI	15	0	58.4	15	58.4	0	0
Slow Finish	Travis CI	15	0	4.9	15	4.9	0	0
Sequential Build	Maven	11	0	9.6	11	9.6	0	0
	Gradle	10	0	4.3	10	4.3	0	0
No Compiler Daemon	Maven	13	0	10.2	8	22.4	5	-9.3
	Gradle	10	0	3.6	9	4.7	1	-6.2
Sequential Test	Maven	11	6	18.7	11	18.7	0	0
	Gradle	11	0	7.5	11	7.5	0	0
Non-Fork Test	Maven	15	3	11.3	15	11.3	0	0
	Gradle	20	0	6.4	20	6.4	0	0
Generate Test Report	Maven	15	0	15.0	15	15.0	0	0
	Gradle	11	0	7.8	11	7.8	0	0
Non-Incremental Compile	Gradle	9	0	11.8	9	11.8	0	0
No Gradle Daemon	Gradle	12	0	5.6	12	5.6	0	0
No Gradle Cache	Gradle	16	0	4.3	16	4.3	0	0
No File System Watch	Gradle	14	0	6.2	14	6.2	0	0
No On-Demand Configure	Gradle	12	0	4.2	12	4.2	0	0
Overall		288	9	12.4	282	12.8	6	-8.8

source code files used in our detection. For *Generate Test Report*, the main reason of rejection is that developers need test report, and they suggest to conditionally skip the test report. For *No Gradle Daemon*, the main reason of rejection is that their builds are not very frequent, and thus the long-lived process may overshadow the benefit.

Summary. 257 pull requests have been merged or accepted by developers, achieving a positive feedback rate of 63.3%. The positive feedback has demonstrated the usefulness and practical value of BuildSonic for CI developers. We also received useful suggestions to enhance BuildSonic from the 104 rejected ones.

4.5 Benefit Analysis (RQ4)

To evaluate the practical benefit of repairing PCSs, we measured the build performance change after repairing PCSs. Specifically, for each of our 1,138 pull requests, we forked the project, triggered a CI build before and after merging our pull request, and measured the build performance change. As our Travis CI environment might be different from the one used in a project, the CI build before merging our pull request failed for 840 pull requests. For the remaining 297 pull requests, the CI build after merging our pull request succeeded for 288 pull requests but failed for 9 pull requests, as reported under the column *Suc.* and *Fail* in Table 3. The reason of failed build after repairing *Sequential Test* and *Non-Fork Test* is that there exist dependencies among tests, and thus parallel testing causes failure. In fact, this problem is also reflected by our developer study (see Sec. 4.4).

On average, the build performance was improved by 12.4% after repairing a PCS (as listed under the column Δ). The highest performance improvement was achieved by repairing *No Dependency Cache*, *Retry Failed Command*, *Wait Long Command*, *Sequential Test* and *Generate Test Report*. While we achieved performance improvement in 282 pull requests with an average improvement of 12.8% (as shown under the columns *Pos.*), we suffered performance degradation in 6 pull requests with an average degradation of 8.8% (as shown under the columns *Neg.*). The reason of performance degradation after repairing *No Compiler Daemon* is that the projects are relatively small-scale, and thus cannot take the full advantage of compiler daemon. In fact, this is consistent with our developer study (see Sec. 4.4).

Summary. The build performance was averagely improved by 12.4% after a PCS was repaired by BuildSonic.

4.6 Efficiency Evaluation (RQ5)

We measured the time overhead of detecting and repairing PCSs in the 4,140 projects. On average, BuildSonic respectively took 158.7 and 5.6 milliseconds to detect and repair all PCSs in one project. We believe this time overhead is practically acceptable for a linter.

Summary. It averagely took 158.7 and 5.6 milliseconds for BuildSonic to detect and repair all PCSs in one project.

4.7 Threats

First, our PCS catalog is derived from three tools in the CI infrastructure for Java projects. However, to the best of our knowledge, this is the first work to study PCSs. Our catalog is served as a starting point for future studies to include more tools and support more languages. Second, a detected PCS is not always a *bad* smell, as reflected by our developer study (see Sec. 4.4). Specific contexts have to be leveraged to determine a bad smell, which often goes beyond the scope of a lightweight linter. In fact, this threat is also shared with others [20, 66]. Third, manual analysis is involved in our evaluation, which might introduce biases. To mitigate it, three authors make great effort to carefully carry out our analysis by following an open coding procedure. Fourth, we currently look at individual configuration options in isolation (but not interactions of configuration options). However, it could be that tweaking one option negatively affects another option. We plan to consider combinations of configuration options when optimizing performance in future.

5 RELATED WORK

Smells in CI/CD. Duvall et al. [14] and Humble and Farley [29] discuss best and bad practices in CI and CD in their classic books about CI and CD. The benefits of CI/CD can be achieved by following best practices and avoiding bad practices. Duvall [13] create a catalog of 50 patterns and their corresponding anti-patterns in the CI/CD lifecycle by surveying related work in literature. Differently, Zampetti et al. [77] compile a catalog of 79 CI bad smells by interviewing practitioners and analyzing StackOverflow posts, where 35 CI bad smells cover 39 patterns/anti-patterns created by Duvall [13]. Their work aims at a broader scope of smells than ours, and inspires our work.

Inspired by Duvall's catalog, Vassallo et al. [65] first propose CI-Odor to detect four CI anti-patterns (i.e., late merging, slow build, broken master branch, and skipping failed tests) by analyzing build logs and repository information. Vassallo et al. [66] then propose CI-Linter to detect four CD configuration smells in GitLab (i.e., fake success, retry failure, manual execution and fuzzy version) by analyzing configuration files alone. Gallaba and McIntosh [20] investigate how Travis CI configurations are used, and design Hansel/Gretel to detect/repair four/three anti-patterns (i.e., redirecting scripts into interpreters, bypassing security checks, using irrelevant properties and commands unrelated to the phase) in Travis CI configuration files. While some of these approaches [20, 66] explore CI configuration

smells, to the best of our knowledge, our work is the first to systematically investigate performance-related CI configuration smells.

It is worth mentioning that smells in IaC (Infrastructure as Code) scripts are also investigated. Sharma et al. [59] collect a catalog of 24 implementation and design configuration smells for IaC scripts developed in Puppet, and develop Puppeteer to detect design configuration smells. They focus on the maintainability of Puppet code. Differently, Rahman et al. [51] concern with the security of Puppet code, and develop SLIC to detect seven security smells in IaC scripts. Our work differs from their work by concentrating on configuration smells that affect the performance of builds in CI.

Time Reduction in CI. Various techniques have been proposed to reduce the consumed time of CI builds. To reduce dependency retrieval time, Celik et al. [8] lazily retrieve parts of libraries that are needed during build execution. To accelerate CI build, build target decomposition techniques [31, 63] are proposed to better suit incremental build, where build targets that are not affected by code changes are skipped [16, 18]. To reduce build time, some plugins [10, 12] are designed into CI tools to skip some builds by manual configuration, whereas some rule-based [2, 34], learning-based [1, 32] and search-based [55] techniques are proposed to automatically identify builds that can be CI skipped. In addition, build outcome prediction techniques [9, 24, 49, 50, 54, 71, 72] are developed to save the time of the builds that are predicted to pass. To reduce testing time, test case prioritization [6, 7, 15, 43, 73] and test case selection [40, 45, 60] are developed into CI to minimize test execution time. Recently, Jin and Servant [33] systematically evaluate and compare the effectiveness of the previous build-level and test-level time reduction techniques. Zhang et al. [79] propose a change-aware approach to triage test failures to reduce diagnosis time. BuildSonic is actually orthogonal to these techniques; i.e., we focus on CI configuration which is different from their focuses.

Build Repairing in CI. Several techniques have been proposed to automatically repair broken builds. Gupta et al. [23], Santos et al. [56] and Mesbah et al. [46] leverage deep learning techniques to automatically fix compilation errors in builds. Hassan and Wang [25] repair build scripts with predefined fix-pattern templates. Instead of relying on historical fixes, Lou et al. [37] design a search-based method to generate build script fixes from the present project and external resources. Vassallo et al. [67] summarize the reasons of build failures and suggest potential fixes by linking related StackOverflow discussions. Macho et al. [41] target dependency-related build failures and develop three rules to repair them. Besides, Lou et al. [38] empirically investigate fix patterns of build failures in three build systems (i.e., Maven, Ant and Gradle). Differently, BuildSonic is not designed to repair build failures but to fix configuration smells.

Empirical Studies about CI. With the increasing adoption of CI, empirical studies have been widely conducted to understand and characterize CI from different aspects, e.g., costs, benefits, barriers, pain points and needs of using CI [27, 28, 64, 70], types of build failures [30, 35, 52, 68], build failures caused by compilation errors [58, 78], test failures [3, 36] and violations in static analysis [76], interplay between pull requests and CI [5, 74], test code evolution in CI [48], pipeline evolution and reconstruction in CI/CD [75], characteristics of long builds in CI [21], and noise and heterogeneity [19] in dataset of CI builds [4]. Several studies [21, 27, 28, 70, 75] have reported that long build time is common; waiting for long builds to finish is a common barrier and pain point faced by developers; and changing configurations to improve performance is common. These studies motivate the need to detect and repair performance-related CI configuration smells to reduce build time and accelerate CI.

6 CONCLUSIONS

In this paper, we have created a catalog of 20 PCSs in three tools (i.e., Travis CI, Maven and Gradle) of the CI infrastructure for Java projects. We have proposed and implemented BuildSonic to detect and repair 15 types of PCSs by analyzing configuration files. We have conducted extensive experiments to demonstrate the effectiveness and efficiency of BuildSonic. In future, we plan to include more CI tools and support more programming languages.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China (2021ZD0112903).

REFERENCES

[1] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. 2021. A Machine Learning Approach to Improve the Detection of CI Skip Commits. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2740–2754.
[2] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. 2019. Which commits can be CI skipped? *IEEE Transactions on Software Engineering* 47, 3 (2019), 448–463.
[3] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 356–367.
[4] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 447–450.
[5] João Helis Bernardo, Daniel Alencar da Costa, and Uirá Kulesza. 2018. Studying the impact of adopting continuous integration on the delivery time of pull requests. In *Proceedings of the IEEE/ACM 15th International Conference on Mining Software Repositories*. 131–141.
[6] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1–12.
[7] Benjamin Busjaeger and Tao Xie. 2016. Learning for Test Prioritization: An Industrial Case Study. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 975–980.
[8] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. 2016. Build System with Lazy Retrieval for Java Projects. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 643–654.
[9] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. 2020. BuildFast: history-aware build outcome prediction for fast feedback and reduced cost in continuous integration. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 42–53.
[10] Travis CI. 2022. *Customizing the Build - Skipping a Build.* https://docs.travis-ci.com/user/customizing-the-build/#skipping-a-build
[11] Travis CI. 2022. *Travis CI User Documentation.* https://docs.travis-ci.com
[12] Cloudbee. 2022. *Jenkins Enterprise by CloudBees 14.5 User Guide - Using the Skip Next Build plugin.* https://docs.huihoo.com/jenkins/enterprise/14/user-guide-14.5/skip-sect-using.html
[13] Paul Duvall. 2011. *Continuous delivery: patterns and antipatterns in the software lifecycle.* https://dzone.com/refcardz/continuous-delivery-patterns
[14] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk.* Pearson Education.
[15] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 235–245.
[16] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's distributed and caching build service. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 11–20.
[17] Martin Fowler. 2000. *Continuous Integration.* http://martinfowler.com/articles/originalContinuousIntegration.html

[18] Keheliya Gallaba, Yves Junqueira, John Ewart, and Shane Mcintosh. 2020. Accelerating Continuous Integration by Caching Environments and Inferring Dependencies. *IEEE Transactions on Software Engineering* (2020).

[19] Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. 2018. Noise and heterogeneity in historical build data: an empirical study of Travis CI. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 87–97.

[20] Keheliya Gallaba and Shane McIntosh. 2018. Use and misuse of continuous integration features: An empirical study of projects that (mis) use travis ci. *IEEE Transactions on Software Engineering* 46, 1 (2018), 33–50.

[21] Taher Ahmed Ghaleb, Daniel Alencar Da Costa, and Ying Zou. 2019. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering* 24, 4 (2019), 2102–2139.

[22] Gradle. 2022. *Gradle User Manual.* https://docs.gradle.org/current/userguide/userguide.html

[23] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. 1345–1351.

[24] Foyzul Hassan and Xiaoyin Wang. 2017. Change-Aware Build Prediction Model for Stall Avoidance in Continuous Integration. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 157–162.

[25] Foyzul Hassan and Xiaoyin Wang. 2018. Hirebuild: An automatic approach to history-driven repair of build scripts. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering*. 1078–1089.

[26] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The art of testing less without sacrificing quality. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 483–493.

[27] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 197–207.

[28] 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*. 426–437.

[29] Jez Humble and David Farley. 2010. *Continuous delivery: reliable software releases through build, test, and deployment automation.* Pearson Education.

[30] Md Rakibul Islam and Minhaz F Zibran. 2017. Insights into continuous integration build failures. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 467–470.

[31] Lukas Jendele, Markus Schwenk, Diana Cremarenco, Ivan Janicijevic, and Mikhail Rybalkin. 2019. Efficient automated decomposition of build targets at large-scale. In *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification*. 457–464.

[32] Xianhao Jin and Francisco Servant. 2020. A Cost-efficient Approach to Building in Continuous Integration. In *Proceedings of the 42nd International Conference on Software Engineering*. 13–25.

[33] Xianhao Jin and Francisco Servant. 2021. What helped, and what did not? An Evaluation of the Strategies to Improve Continuous Integration. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*. 213–225.

[34] Xianhao Jin and Francisco Servant. 2022. Which builds are really safe to skip? Maximizing failure observation for build selection in continuous integration. *Journal of Systems and Software* 188 (2022), 111292.

[35] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. 2014. Why do automated builds break? an empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. 41–50.

[36] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the cost of regression testing in practice: a study of Java projects using continuous integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 821–830.

[37] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-driven build failure fixing: how far are we?. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 43–54.

[38] Yiling Lou, Zhenpeng Chen, Yanbin Cao, Dan Hao, and Lu Zhang. 2020. Understanding build issue resolution in practice: symptoms and fix patterns. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 617–628.

[39] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 643–653.

[40] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice*. 91–100.

[41] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*. 106–117.

[42] Simon Maple and Andrew Binstock. 2018. *JVM Ecosystem report 2018 – About your Tools.* https://snyk.io/blog/jvm-ecosystem-report-2018-tools/

[43] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. In *Proceedings of the IEEE International Conference on Software Maintenance*. 540–543.

[44] Maven. 2022. *Maven Documentation.* https://maven.apache.org/plugins/index.html

[45] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. 233–242.

[46] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. Deepdelta: learning to repair compilation errors. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 925–936.

[47] André N Meyer, Laura E Barton, Gail C Murphy, Thomas Zimmermann, and Thomas Fritz. 2017. The work life of developers: Activities, switches and perceived productivity. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1178–1193.

[48] Gustavo Sizilio Nery, Daniel Alencar da Costa, and Uirá Kulesza. 2019. An Empirical Study of the Relationship between Continuous Integration and Test Code Evolution. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. 426–436.

[49] Ansong Ni and Ming Li. 2017. Cost-effective build outcome prediction using cascaded classifiers. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 455–458.

[50] Ansong Ni and Ming Li. 2018. ACONA: Active Online Model Adaptation for Predicting Continuous Integration Build Failures. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering: Companion*. 366–367.

[51] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The seven sins: Security smells in infrastructure as code scripts. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering*. 164–175.

[52] 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 *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 345–355.

[53] Kristian Rosenvold. 2018. *Parallel builds in Maven 3.* https://cwiki.apache.org/confluence/display/MAVEN/Parallel+builds+in+Maven+3

[54] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. 2020. Predicting continuous integration build failures using evolutionary search. *Information and Software Technology* 128 (2020), 106392.

[55] Islem Saidani, Ali Ouni, and Wiem Mkaouer. 2021. Detecting skipped commits in continuous integration using multi-objective evolutionary search. *IEEE Transactions on Software Engineering* (2021).

[56] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. 2018. Syntax and sensibility: Using language models to detect and correct syntax errors. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*. 311–322.

[57] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering* 25, 4 (1999), 557–572.

[58] 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*. 724–734.

[59] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories*. 189–200.

[60] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and improving regression test selection in continuous integration. In *Proceedings of the IEEE 30th International Symposium on Software Reliability Engineering*. 228–238.

[61] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A longitudinal analysis of bloated Java dependencies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1021–1031.

[62] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering* 26, 3 (2021), 1–44.

[63] Mohsen Vakilian, Raluca Sauciuc, J David Morgenthaler, and Vahab Mirrokni. 2015. Automated decomposition of build targets. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 123–133.

[64] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. 805–816.

[65] Carmine Vassallo, Sebastian Proksch, Harald C Gall, and Massimiliano Di Penta. 2019. Automated reporting of anti-patterns and decay in continuous integration. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering*. 105–115.

[66] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C Gall, and Massimiliano Di Penta. 2020. Configuration smells in continuous delivery pipelines:

a linter and a six-month study on gitlab. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 327–337.

[67] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. 2018. Un-break My Build: Assisting Developers with Build Repair Hints. In *Proceedings of the 26th International Conference on Program Comprehension*. 41–51.

[68] 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 *Proceedings of the IEEE international conference on software maintenance and evolution*. 183–193.

[69] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An empirical study of usages, updates and risks of third-party libraries in java projects. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. 35–45.

[70] 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. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 647–658.

[71] Jing Xia and Yanhui Li. 2017. Could we predict the result of a continuous integration build? An empirical study. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security Companion*. 311–315.

[72] Zheng Xie and Ming Li. 2018. Cutting the Software Building Efforts in Continuous Integration by Semi-Supervised Online AUC Optimization. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. 2875–2881.

[73] Shin Yoo, Robert Nilsson, and Mark Harman. 2011. Faster fault finding at Google using multi objective regression test optimisation. In *Proceedings of the 8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

[74] Fiorella Zampetti, Gabriele Bavota, Gerardo Canfora, and Massimiliano Di Penta. 2019. A study on the interplay between pull request review and continuous integration builds. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*. 38–48.

[75] Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. 2021. CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. 471–482.

[76] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 334–344.

[77] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. 2020. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering* 25, 2 (2020), 1095–1135.

[78] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. 2019. A large-scale empirical study of compiler errors in continuous integration. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 176–187.

[79] Chen Zhang, Bihuan Chen, Xin Peng, and Wenyun Zhao. 2022. Buildsheriff: Change-Aware Test Failure Triage for Continuous Integration Builds. In *Proceedings of the IEEE/ACM 44th International Conference on Software Engineering*. 312–324.