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.

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 [28, 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.

- 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).
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.

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.
We propose and design BuildSonic as a linter that can statically and efficiently detect and repair PCSs by analyzing configuration files. Five of the 20 types of PCSs (marked by $\times$ in the last two columns of Table 1) are not supported in BuildSonic because of heavy weight analysis (e.g., program analysis) is needed to detect them. We detect 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 $\circ$ in the last two columns of Table 1), and repair five of them semi-automatically (marked by $\Box$ in the last two columns 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 introduced 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. 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 $\text{HOME}/.m2$ For Gradle projects, we set directories to $\text{HOME}/.gradle/caches$ and $\text{HOME}/.gradle/wrapper/$. These 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

---

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Deep Clone</td>
<td>Travis CI</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>No Dependency Cache</td>
<td>Travis CI</td>
<td></td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Retry Failed Command</td>
<td>Travis CI</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Wait Long Command</td>
<td>Travis CI</td>
<td></td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Slow Finish</td>
<td>Travis CI</td>
<td></td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Heavy Job</td>
<td>Travis CI</td>
<td></td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Sequential Build</td>
<td>Maven, Gradle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No Compiler Daemon</td>
<td>Maven, Gradle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequential Test</td>
<td>Maven, Gradle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Fork Test</td>
<td>Maven, Gradle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generate Test Report</td>
<td>Maven, Gradle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Improper Repository</td>
<td>Maven, Gradle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unused Dependency</td>
<td>Maven, Gradle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unnecessary Test</td>
<td>Maven, Gradle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Small Heap Size</td>
<td>Maven, Gradle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Incremental Compile</td>
<td>Gradle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No Gradle Daemon</td>
<td>Gradle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No Gradle Cache</td>
<td>Gradle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No File System Watch</td>
<td>Gradle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No On-Demand Compile</td>
<td>Gradle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**3.2 Detecting and Repairing PCSs**

We propose and design BuildSonic as a linter that can statically and efficiently detect and repair PCSs by analyzing configuration files. Five of the 20 types of PCSs (marked by $\times$ in the last two columns of Table 1) are not supported in BuildSonic because of heavy weight 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 $\circ$ in the last two columns of Table 1), and repair five of them semi-automatically (marked by $\Box$ in the last two columns 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 introduced 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.
**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 developers 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 introduce 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.to true/false` 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.to true/false`, 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 to 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/false`, 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). APCS 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.
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 maxmex 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: No-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 No-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 also 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 option 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 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 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 --no-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 5.3 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 5.3 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 Exp. 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>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Deep Clone</td>
<td>Travis CI</td>
<td>63</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>50</td>
<td>9</td>
<td>0</td>
<td>7(12)</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>No Dependency Cache</td>
<td>Travis CI</td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>15</td>
<td>1</td>
<td>0</td>
<td>0(1)</td>
<td>0</td>
<td>13</td>
<td>100</td>
</tr>
<tr>
<td>Retry Failed Command</td>
<td>Travis CI</td>
<td>41</td>
<td>41</td>
<td>0</td>
<td>34</td>
<td>6</td>
<td>2(6)</td>
<td>2</td>
<td>18</td>
<td>60.0</td>
<td></td>
</tr>
<tr>
<td>Wait Long Command</td>
<td>Travis CI</td>
<td>80</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>10</td>
<td>0</td>
<td>8(9)</td>
<td>2</td>
<td>21</td>
<td>50.0</td>
</tr>
<tr>
<td>Slow Finish</td>
<td>Travis CI</td>
<td>56</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>48</td>
<td>16</td>
<td>0</td>
<td>2(15)</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Sequential Build</td>
<td>Maven</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>No Compiler Daemon</td>
<td>Grapple</td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>(0)</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Sequential Test</td>
<td>Maven</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0(0)</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Non-Fork Test</td>
<td>Grapple</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0(0)</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Generate Test Report</td>
<td>Maven</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Non-Incremental Compile</td>
<td>Grapple</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0(1)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>No Gradle Daemon</td>
<td>Grapple</td>
<td>82</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5(1)</td>
<td>2</td>
<td>39</td>
</tr>
<tr>
<td>No Gradle Cache</td>
<td>Grapple</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0(0)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>No File System Watch</td>
<td>Grapple</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>No On-Demand Config</td>
<td>Grapple</td>
<td>18</td>
<td>18</td>
<td>0</td>
<td>16</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2(0)</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>Overall</td>
<td></td>
<td>484</td>
<td>371</td>
<td>54</td>
<td>340</td>
<td>54</td>
<td>28(48)</td>
<td>10</td>
<td>195</td>
<td>60.8</td>
<td>19834</td>
</tr>
</tbody>
</table>

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 338 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 F.P.) as the ratio of merged and accepted pull requests among all feedback 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...
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 Pos. in Table 3. 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 bias. 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

<table>
<thead>
<tr>
<th>Table 3: Build Performance Change after Repairing PCSs (Suc. = Succeed, Pos. = Positive, and Neg. = Negative)</th>
</tr>
</thead>
<tbody>
<tr>
<td>--------------------------</td>
</tr>
<tr>
<td>Deep Clone</td>
</tr>
<tr>
<td>No Dependency Cache</td>
</tr>
<tr>
<td>Retry Failed Command</td>
</tr>
<tr>
<td>Wait Long Command</td>
</tr>
<tr>
<td>Slow Finish</td>
</tr>
<tr>
<td>Sequential Build</td>
</tr>
<tr>
<td>No Compiler Daemon</td>
</tr>
<tr>
<td>Sequential Test</td>
</tr>
<tr>
<td>Non-Fork Test</td>
</tr>
<tr>
<td>Generate Test Report</td>
</tr>
<tr>
<td>Non-Incremental Compile</td>
</tr>
<tr>
<td>No Gradle Daemon</td>
</tr>
<tr>
<td>No Gradle Cache</td>
</tr>
<tr>
<td>No File System Watch</td>
</tr>
<tr>
<td>No On-Demand Configure</td>
</tr>
<tr>
<td>Overall</td>
</tr>
</tbody>
</table>

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).
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 [58] techniques are proposed to automatically identify builds that can be CI skipped. In addition, build outcome prediction techniques [9, 24, 49, 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**


