Every Build You Break: Developer-Oriented Assistance for Build Failure Resolution

Vassallo, Carmine ; Proksch, Sebastian ; Zemp, Timothy ; Gall, Harald C

DOI: https://doi.org/10.1007/s10664-019-09765-y

Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Every Build You Break: Developer-Oriented Assistance for Build Failure Resolution

Carmine Vassallo · Sebastian Proksch · Timothy Zemp · Harald C. Gall

This is a pre-print of an article published in Empirical Software Engineering. The final authenticated version is available online at: https://doi.org/10.1007/s10664-019-09765-y.

Abstract Continuous integration is an agile software development practice. Instead of integrating features right before a release, they are constantly being integrated into an automated build process. This shortens the release cycle, improves software quality, and reduces time to market. However, the whole process will come to a halt when a commit breaks the build, which can happen for several reasons, e.g., compilation errors or test failures, and fixing the build suddenly becomes a top priority. Developers not only have to find the cause of the build break and fix it, but they have to be quick in all of it to avoid a delay for others. Unfortunately, these steps require deep knowledge and are often time-consuming. To support developers in fixing a build break, we propose Bart, a tool that summarizes the reasons for Maven build failures and suggests possible solutions found on the internet. We will show in a case study with 17 participants that developers find Bart useful to understand build breaks and that using Bart substantially reduces the time to fix a build break, on average by 37%. We have also conducted a qualitative study to better understand the workflows and information needs when fixing builds. We found that typical workflows differ substantially between various error categories, and that several uncommon build errors are both very hard to investigate and to fix. These findings will be useful to inform future research in this area.

Keywords Software Engineering · Agile Software Development · Software Development Tools · Build Break · Summarization · Error Recovery

Carmine Vassallo, Sebastian Proksch, Timothy Zemp, Harald C. Gall
University of Zurich - Switzerland
E-mail: vassallo@ifi.uzh.ch, proksch@ifi.uzh.ch, timothy.zemp@uzh.ch, gall@ifi.uzh.ch

1 Introduction

Continuous integration (CI) is an agile software development practice that advocates frequently integrating code changes introduced by different developers into a shared repository branch [18]. An automated system builds every commit, runs all tests, and verifies the quality of the software, e.g., through automated static analysis tools [8]. This helps to detect issues earlier and locate them more easily [19]. CI is widely adopted in industry and open source environments [53] and has already proven its positive effects on release frequency, software reliability, and overall team productivity [17].

Despite its undisputed advantages, the introduction of CI in established development contexts is anything but trivial. Hilton et al. [16] found that *build breaks* are a major barrier that hinders CI adoption and various reasons exist for a build to break [51], e.g., compilation errors, testing failures, poor code quality, or missing dependencies. Developers need to learn how to efficiently identify the reasons for a build break and, unfortunately, the required skill set is still different to traditional debugging. Established techniques that are widely used in the development environment [30], like setting breakpoints to investigate a program right before a crash, are not applicable, which makes it difficult and time consuming to remove a build break [16]. As a result, developers spend a significant amount of their working time comprehending and solving build breaks. It takes on average one hour to fix build breaks [19].

Those results motivate the need for new ways to support developers in *understanding* build breaks and in *deriving a fix*. Existing works have already proposed automatic build-fixing techniques, e.g., [23]. However, such approaches are typically limited to a specific type of build break (i.e., fixing unresolved dependencies). In this paper, we propose a *developer-oriented* assistance system that supports build break fixes by summarizing available information and by linking to external information. We do not focus on a specific build problem, but empower the developer by providing relevant information in a wide range of build failures. To the best of our knowledge, we are the first to propose such an *information-centric* developer support during build breaks.

In the first part of this paper, we will investigate whether generated summaries can help developers with comprehending build logs. We will also empirically analyze the effect of a build summarization tool on the time needed for understanding and fixing a build break. More specifically, we will answer two research questions:

RQ_1*:* Are summarized build logs more understandable?
RQ_2*:* Does a semi-automated support system influence the time that is required to fix a broken build?

We have implemented the *Build Abstraction and Recovery Tool* (BART) to study these questions. Bart is a Jenkins plugin that summarizes logs of failed Maven builds and that links related StackOverflow discussions to help solve the build failure. To answer both research questions, we deployed Bart in an empirical study with 17 developers. Our results show that developers

consider the generated summaries helpful for fixing build breaks; as a further result, the resolution time for fixing the build can be significantly reduced.

However, the results also show that developers only find some solution hints valuable, other cases are perceived as less helpful. To understand this observation, we asked the following research questions and discuss the corresponding results in the second part of the paper.

RQ_3: How do developers approach different types of build failures?
RQ_4: What types of build failures are hard to fix?

To answer these questions, we have conducted a large-scale qualitative study that involved 101 developers. Our results show that fundamental differences exist in how developers approach the different failure categories that have been investigated. For example, while code-analysis build failures are easy to fix with information contained in the build log, other scenarios like testing failures require a much deeper investigation of information that needs to be collected elsewhere. Our survey has also revealed that problems related to the build infrastructure are not only hard to investigate, but also hard to address, once the problem is understood. While these categories are known, they have not yet been the focus of an investigation, likely because they occur too infrequently. Our results suggest that they are still worth investigating, because they represent a major pain-point for developers when they occur.

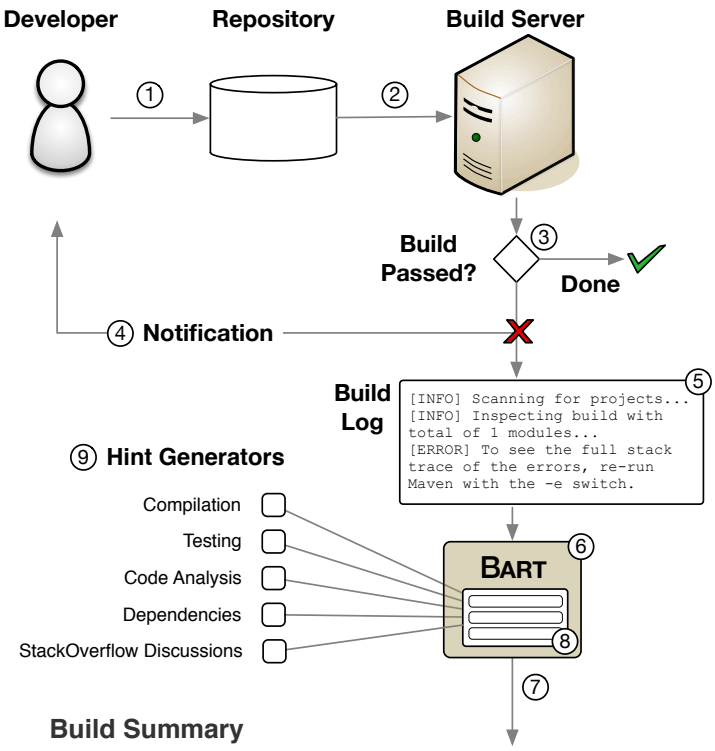
In summary, this paper makes the following contributions:

- Presentation of a novel idea to support build fixing through build log summarization and linking to StackOverflow resources.
- Proof-of-concept implementation for Bart.
- Investigation of the effect of Bart on the understandability of build failures and the time for fixing a build failure
- A qualitative analysis of the workflows and information needs of developers that resolve build failures.

This paper is an extension of previous work [49]. Compared to the original paper, it contains the following new contents.

- We have added two research questions that investigate qualitative aspects.
- We have doubled the number of the participants in the experiment to improve the evaluation of Bart and to make our findings more convincing.
- We have conducted 9 semi-structured interviews with several study participants to better understand information needs and the limitations of Bart.
- We have sent out a survey and analyzed 101 answers from a diverse set of developers to validate our interview results with a more general crowd.
- We have derived a methodology to open-code build fixing workflows.
- We have published Bart and open-sourced the implementation [3].

Scripts, data, and additional material are available in the online appendix [50].



Build Summary

Build Status	Failed
Failed Goal	org.apache.maven.plugins:maven-compiler-plugin:2.1:compile
Error Cause	Compilation Failure
Reactor	ActiveJpa SUCCESS test SUCCESS core FAILURE utils SKIPPED

Reason for Build Failure: *Compilation Failure*

Hint: Compilation

Your build contains a compilation error. Please check the following file:

File: Model.java
Line: 224
Reason: Model is abstract; cannot be instantiated

Fig. 1: Overview over the Build Summarization Approach

2 Creating a Build Abstraction and Recovery Tool (Bart)

To understand our vision of *developer-oriented* assistance, it is important to reflect on the typical CI pipeline that is illustrated in Figure 1. Developers working in such a pipeline synchronize their *working copy* frequently with the central repository that is shared by all team members (1). They pull changes from others and push their own contributions. The repository is being monitored by the build server. Every time a new commit is pushed to the repository, the build server will update its working copy and build the project (2). This typically includes multiple build stages, for example, compiling the sources, running the tests, generating documentation, or validating the software quality through static analysis. If all these stages have passed (3), the build is considered to be successful, which typically results in the release of the software. If the build fails, on the other hand, developers are being notified by the build server about the error. This typically happens through sending an email or by visiting the web frontend of the build server (4). The developers have to consult the build log (5) to understand the problem and provide a fix, a difficult and time-consuming task that typically consists of three steps.

Log Inspection: The developer investigates the build log to get further information about the build failure. While it is often simple to spot the part in which the build failed, it is very often difficult to read the log and to understand the failure reason.

Hypothesis: Once the developer has an intuition about the root cause for the break, the problem should be replicated, if possible on the developer machine and ideally by providing a test. This makes it possible to use a debugger to inspect the failure.

Fix: If the root cause of the build failure is identified, fixing it is usually the easy part. The developer implements the fix, pushes it to the repository, and waits for the result of the new build. All the steps are re-executed if the build fails again.

Executing these three steps is difficult and deriving a hypothesis about the root cause of the failure requires experience. If the developer gets stuck, a common strategy is to ask more experienced team members [20] or to search on the Internet for solutions [44].

In this paper, we present Bart (6), the *Build Abstraction and Recovery Tool* that supports developers by enriching the build log through summarization and linking of external resources. We have designed Bart as a support tool for Maven builds and we have created a proof of concept implementation for the build server Jenkins. Our solution is complementary to the existing pipeline that we have discussed before. Bart does not replace the inspection of the build log, instead, the build log is embellished with further information to facilitate a faster and better decision making of the developer. For example, our screenshot (7) shows the *Build Summary* (a general summary of the build result) as well as a list of *hints*, in this case details about a compilation error. These hints are included in the Jenkins build overview page.

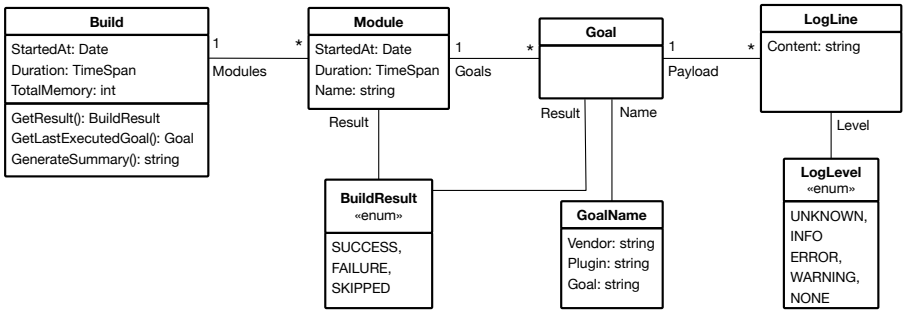


Fig. 2: Meta-Model that Is Available to Hint Generators In Bart

Bart facilitates the generation of these hints with two reusable parts. First, it parses the build log, extracts all relevant sections (e.g., keywords, commands, built modules), and stores this preprocessed information in a meta-model (8). Second, Bart is extensible through additional *Hint Generators* (9). We have built five different hint generators that summarize the information found in the build log, as well as hint generators that use information from the build log to search for solutions in the Internet. For example, our proof-of-concept implementation can link to related discussions on StackOverflow.

We will introduce these individual parts in the remainder of this section. Section 2.1 introduces our build-log meta model and describes our parsing. Section 2.2 contains the extension point mechanism for hint generators and a description of the four different hint generators that summarize build log information. Section 2.3 discusses the hint generator that links build failures to external information, such as discussions on StackOverflow.

2.1 Detecting Failure Information in the Log

Build tools typically log all their actions in a detailed log that allows developers to reconstruct their actions after the fact. Such a build log is typically stored as plain text. All details about the build are contained, but such logs are very large, e.g., even the build log of the relatively small Maven build tool itself ($\sim$ 130K LoC) results in a build log of more than 1,500 lines. To make the creation of new hint generators in Bart straight-forward, we preprocess these logs. We provide an abstraction over a Maven build log that makes it easy to find exactly the information that the hint generators need. While we are going to focus on the Maven build system [25], the most used build tool among Java developers [24], the underlying idea is general and can also applied to other build systems. In this section, we will first briefly describe Maven's building concept, the structure of its build log, and our parsing. We will then introduce our meta model that we use to store the relevant build information.

Maven follows the concept of convention over configuration. It provides a standard build configuration that defines several *phases* that are run one by

one in the default build lifecycle (e.g., `compile`, `test`, `verify`). The set of phases is fixed and most of them are empty by default. A concrete build job can now add specific *goals* to the different phases, if needed. So, for example, a project could add the invocation of a *static analysis tool* to the `verify` phase. In practice, build files contain the configuration for many of such *goals* that range from dependency resolution in the very beginning of builds to packaging or deployment that typically take place at the end.

Maven builds are organized hierarchically. In addition to the goals that are configured in the build file for the current *module*, parent configurations can be referenced to inherit configuration options. In addition, it is possible to refer to *submodules* that are then build together with the current *module*.

At each build and starting from the module for which the build was triggered, Maven creates the dependency tree between all (sub-) modules and schedules the individual builds in an order that does not violate their dependencies. In Maven terminology, this *build plan* that contains the names of all *modules* is called the *reactor*. During the build, one *section* is dedicated to each *module*. This section contains entries for each executed *goal*, which might also prints additional output to the log. At the end of the build, Maven generates a *reactor summary*, which lists the individual build results. In addition, the reactor summary will also show consumed memory, and -for build failures- further information about the *module* and *goal*, in which the build broke.

As a first step in the failure resolution, a developer that has to read such a build file, has to navigate through a big log to find the relevant information. This is also a hard task for an automated processor, because the individual parts need to be parsed or otherwise processed with string utilities.

To simplify the access to the contained information, we implemented a parser that, taking a build log as input, fills the meta-model that we have created, as depicted in Figure 2. The model follows the structure that we have introduced before. The root entity of a build log is a `Build`, which has basic properties like the required memory for the build. A build refers to several `Module` definitions that are part of the build. In addition to timing information, each module has a unique name and a result. It also contains information about the different `Goal`s that were executed while building it. Each `Goal` combines the `GoalName` (i.e., a reference to the tool that was run), the `BuildResult` or the invocation, and a `Payload`, which contains all output that was generated for this `Goal`. Each line is annotated with a `LogLevel` and contains content as `string`.

The original build log contains a *reactor summary* at the very end that can be requested by calling `Build.getSummary()`. We do not explicitly store the contained information in the meta-model, because it can be fully inferred from the stored data. In general, this meta-model does not lose any information of the original log. It splits information into individual sections and provides easy access, but it could be transformed back into the original log file.

Please note that the parser used for this paper does not support the complete meta-model yet. Our implementation supports all parts that were used for the summary generation, leaving out details like the build duration or the

amount of memory that were irrelevant in this context. This does not represent a conceptual limitation of the meta model though and can be solved by spending more implementation effort on the parser.

2.2 Summarizing Build Log Information

Understanding the extensive build logs generated through a MAVEN build is tedious. The *reactor summary* that is automatically generated at the end of a build contains basic information about a build failure and represents a first step in the right direction. However, the information is presented as plain text without any highlighting and it is hard to read by developers. In addition, only the lines of the failing goal that are marked as *error* are included in the summary, surrounding information, which could further explain the error, is omitted. We think that this current build summarization is not sufficient. Thus, we propose an improved summarization and a highlighting of the important pieces to ease the life of developers and to make them more efficient in understanding the build log.

Inspecting the failing section of the build log should provide all required information to understand the cause of the failure, so it represents the starting point of any investigation of a build failure. However, the actual information that is included in the corresponding part of the build log can be very extensive (e.g., long lists of executed tests) and it also heavily depends on the failing goal. Fortunately, it has already been shown that build failures can be assigned to different categories, based on the goal or build step that failed [51]. We propose to provide a better guidance in the fixing process by tailoring the summary to the failure category.

We have created a conceptual framework and use it to present build summaries. The implementation is modular and general, but we provide an integration into the build server JENKINS, which adds the polished output to a build report that is shown in Figure 1. To create this output, we first parse the build log into our meta-model. Build status and individual build goals are directly available in the model. We do not add hints to successful builds, so we can directly skip them in the processing. For a failing build, we determine the failure category based on the last executed goal. Using this category, it is possible to find applicable hint generators to save execution time. These specialized hint generators can provide hints that might point the developer towards a build fix. A `Hint` is just a key-value dictionary that can contain arbitrary contents. Hint generators can return an empty list, a single hint, or also multiple hints. The presentation to the user is achieved by iterating over all hints and by putting all their key/value pairs into a table. We do not imply any restriction on the type of key that can be generated, because the actual hint generator should select the most meaningful information for the developer.

In this paper, we focus on a proof-of-concept implementation that supports the most frequent build break categories. Based on the dataset of industrial and open-source projects analyzed in previous work [51], the most common

build failure types are *Compilation* (9% of open-source and 6.5% of industrial projects), *Dependencies* (7.1% of open-source and 6.3% of industrial projects), *Testing* (41.3% of open-source and 39.5% of industrial projects), and *Code Analysis* (4.2% of open-source and 16.4% of industrial projects). Please note that we did not consider cross-cutting categories such as *Release Preparation* as suitable for our implementation, because (i) they are not associated with a specific step of the build process and (ii) the structure of the log is highly variable. In addition, we want to provide an additional *Build Summary* that provides an overview over the whole build. In the following, after providing an overview, we will briefly introduce these different hint generators, which information we want to show to the developer in each case, and how we can get access to the required build log data.

□ *Build Summary:* The Build Summary provides a high-level overview over the result of the build. It mimics MAVEN's reactor summary, but reduces the amount of information to a minimum. Rich formatting options are applied to highlight the different information. You will find an example of the build summary in Figure 1. Each summary is composed by the following sections.

Reactor Summary: The list of modules can be requested from the `Build` object, their individual results can be directly used.

Build Result: Can directly be requested from a `Build` object.

Failed Goal: The last executed goal of a failing `Build`.

Error Cause: The error cause consists of the error message that is printed by the failing goal. These can be extracted by selecting all lines of the goal that have the log level "error".

□ *Compilation Failures:* The hint generator should provide detailed information about the location of the compilation error. All this information can be found in the `Payload` of the failing goal.

Type: Name of the type (e.g., class) that could not be compiled.

Line: Line number, in which the error has occurred.

Reason: Textual description of the compilation error, e.g, instantiation of an abstract class, when provided by the build log.

□ *Dependency Failures:* Declared dependencies can lead to various build failures. Our summarizer helps understanding the dependency error by showing the following information.

Dependency: The name of the dependency that causes the failure. The MAVEN coordinates of the dependency are mentioned in the error message and we use a regular expression to parse them.

Reason: Textual description of the dependency error. Typical reasons are invalid versions numbers or missing internet access.

□ *Testing:* Testing failures are particularly tricky to fix, because they can occur after introducing a change in a completely different part of the system. For this reason, it is important that a hint does not only contain the location of the test, which is required to replicate the failure locally, it should also contain the reason that explain the failure. As a result, the hint generator reports the following:

Location: The location, in which the testing failure occurs. This contains both the *test class* and the failing *test case*.

Reason: A textual description of the test failure. This is taken from the *failed* assertion statement, so the quality of these descriptions depends on the concrete test case. In case of an *error*, also the stack trace of the failure will be included.

□ *Code Analysis:* Many builds use static analysis tools to validate properties (e.g., a consistent programming style) of the system. Like for all other types, also for code-analysis failures developers need to find the relevant information that is spread across different sections of the log. Let us consider a simple example, a CHECKSTYLE error. The MAVEN's reactor summary only reports "You have 1 Checkstyle violation", but to understand the actual problem, the developer first has to find the right section of the log, which then includes the detailed description (e.g., "Line is longer than 120 characters"). Each static analysis tool produces a different output and individual hint generators are needed to cover them. We selected CHECKSTYLE as a representative for such static analysis tools because is the most frequently used in open-source JAVA projects [5, 56]. We include the following information that help to understand related build failures:

Location: The path to the *file*, in which the style violation was detected. The location also includes the *line number*.

Reason: Name of the style rule that caused the failure. These names are typically very expressive, e.g., "method name too long", so the proposed hints are potentially very meaningful.

□ *Future Extensions:* Future hint generators might use other build-log information in their hints. They can either reuse our meta model or provide their own extraction strategy to find the interesting information in the build log. It is possible, for example, to use custom regular expressions to parse specific information from the `Payload`. As a fallback, it is always possible to recover a full build log from our meta-model, which ensures support for all hint generators that work on the build log. Extensions that require *external files* in addition to the build log, like test coverage reports, represent a special case. These files are not contained in the build log. Hint generators that require access can still parse the respective path from the build log and open these files separately.

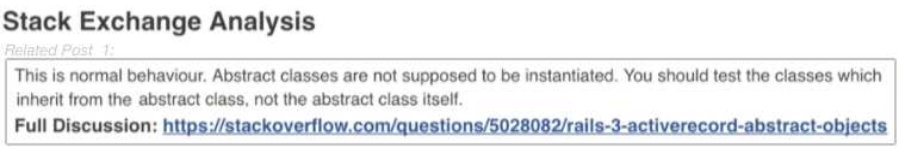


Fig. 3: Hint that Links Related StackOverflow Discussion

2.3 Hints from External Sources

Summarizing local information improves the ability to understand the contents of the build log. However, developers may encounter situations, in which the error message is easy to understand, but requires a complex fix. For example, if the *source level* is not configured in Maven, it will use Java version 5 by default. If the developer now writes Java code in a newer version, e.g., version 8, and uses one of the newly introduced constructs, e.g., lambda expressions, the Maven compiler plugin will fail with a syntax error, even though no problem will be reported in the development environment. While the summary will point out a syntax error very clearly, in this case, an inexperienced developer will struggle to solve this on their own and will either ask a more experience colleague for help or simply search for a solution on the internet. For this reason, we also need to provide an infrastructure in Bart that allows the creation of hint generators, which can go beyond a *local summarization*. These *external* hint generators should be able to add additional hints and link to external resource in their suggestions for possible solutions.

It is very likely that, in case of a build failure, a similar build break has already been discussed online. Previous work has already shown that question and answer sites, like StackOverflow, can provide a great source of information to support developers [32]. The site contains almost 60K discussions that are related to Maven development [43], which makes us very confident that it can also be a good source for tips on how to fix a broken build. An example of a hint that refers to a StackOverflow discussion is shown in Figure 3. The example hint explains a specific compilation failure and also links the full discussion to provide additional context.

To obtain relevant discussions from StackOverflow, we use a twofold approach. First, we query StackOverflow for discussions that are related to the build log. Second, we rank the returned posts and present the most relevant discussions to the developer. The hint engine starts with requesting the build log and the hints that have already been generated in the *local summarization* step. Given that the *local* hint generators have already identified the key parts of the build log, we make use of this information to create a query that is as specific as possible. The hints are being *cleaned* by removing local information (e.g., paths or file names) and common overhead that is added in every Maven build (e.g., formatting characters or *goal* names). The resulting query mainly contains the error message that describes the failing build and it is used to search on StackOverflow.

Table 1: Analyzed Projects

Project Name	Version	Size (LoC)	#GitHub Stars	#Builds
ActiveJPA	0.3.5	39,335	143	123 (master)
Sentry-java	5.0.0	113,332	312	509 (master)
Fongo	2.1.1	31,088	374	404 (master)

In a second step, we rank the returned posts to identify the ones that are most related to the actual build log. To achieve this, the build log is first *cleaned* in the same way as the query and then *tokenized* to create a set of keywords that describe the build. Common english stop words (e.g., "the", "or", "and") are removed to improve this set of keywords. For each post, we calculate a post score by counting how many different keywords are used in the body of the discussion. After ordering the posts by their score, the top four proposals are selected and shown to the developer. We decided to list the top four posts because because this number typically avoid that developers have to further scroll the page. We did not want to come up with a long list that adds an additional burden on the developers.

3 Investigating the Effect of Bart on Build Fixing Practice

To understand the effects that a tool like Bart has on the practice of fixing a build, we conduct an empirical study, in which we will quantitatively measure Bart's capability to improve the *understandability* of build failures and to improve the *performance* of developers when fixing broken builds. This first study consists of two parts, a *controlled experiment* and a follow-up *questionnaire*, that covers the different aspects that we want to investigate.

Understandability: In the first part, we assess whether or not the summaries generated by Bart make it easier to understand the cause of a build break and to formulate a solution strategy.

Performance: In the second part, we measure Bart's effect on the required time to fix certain types of build breaks.

In the following, we will first introduce the context and our methodology that we have applied to investigate both aspects and we will then answer the first two research questions.

3.1 Context

The *context* of our study includes (i) as *objects*, build breaks that we have generated from selected Java projects, and (ii) as *subjects*, developers that participated in our controlled experiment.

The three software systems that we considered in our study are illustrated in Table 1. We followed the methodology of Bavota et al. [4] to select systems that developers can easily get familiar with and that are, at the same time, representative for real software systems. ActiveJpa [2] is a Java library that implements the active record pattern on top of Java Persistence APIs (JPA). Sentry-java [39] is an error tracking system that helps developers to monitor and fix crashes in real time. Fongo [10] is an in-memory Java implementation of MongoDB. The considered systems are hosted on GitHub and built on the cloud-based build infrastructure of TravisCI. While our selected systems have less than 500K lines of code, they are very popular (more than 100 stars on GitHub) and frequently built (more than 100 builds on the master branch). For our study, we have injected bugs into these systems to generate broken builds. The introduced bugs belong to the four most recurrent categories of build failures [51], i.e., *compilation*, *dependencies*, *testing*, and *code analysis*. We created different mutations of the extracted systems for every category of broken builds and ended up with five broken ActiveJPA versions, two broken Sentry-java versions, and one broken Fongo version.

More details about these broken versions are depicted in Table 2. We always created two mutated versions to avoid learning effects in both tasks of the study. To generate the *testing* build breaks we changed an assertion in the test class `FongoAggregateProjectTest` from `assertNotNull` to `assertNull`. We have also altered the `count` method of the class `org.activejpa.entity.Model` by adding an incorrect offset to the returned value. We chose an obvious mistake to fail the build, i.e., we added 100, which is easy to spot. To provoke *dependency* build breaks, we have first inserted an obvious non-existing dependency and, second, we included a typo in another (existing) dependency. We have introduced two *code analysis* build breaks in Sentry-java, by adding a new method with a very long name to the class `SentryAppender` and by deleting the Javadoc comment of the method `doClose` in the class `AsyncConnection`. Both are picked up by CheckStyle, which will raise the errors *Very long function name* and *Javadoc has empty description section*. Finally, to create *compilation* errors we added a return statement in the void method `close()` of the class `org.activejpa.JPA` and inserted an illegal combination of `static` and `abstract` in the signature of the method `deleteAll` of class `org.activejpa.entity.Model`.

We contacted participants by sending out invitations to students from the University of Zurich (UZH) and Swiss Federal Institute of Technology in Zurich (ETHZ) and to professional developers that we reached through our personal contacts. In total, 17 participants completed our controlled experiment. We made sure that all participants have used Maven before. The majority of our participants (9, 53%) report three to five years of programming experience, while six participants (35%) declare that their experience exceeds five years. Only two participant reported less than three years of programming experience. 11 (65%) participants work as professional developers. We asked the participants to self-estimate their programming experience in a five-point *Likert scale* [21] from *very low* to *very high*. Out of all participants, 10 report an

Table 2: Mutated Components in the Analyzed Systems

Build Break Type	Project	Mutated Component
Task 1		
Test	Fongo	com.github.fakemongo.FongoAggregateProjectTest
Compilation	ActiveJPA	org.activejpa.jpa.JPA
Code Analysis	Sentry-java	net.kencochrane.raven.connection.AsyncConnection
Dependencies	ActiveJPA	pom.xml
Task 2		
Test	ActiveJPA	org.activejpa.entity.Model
Compilation	ActiveJPA	org.activejpa.entity.Model
Code Analysis	Sentry-java	net.kencochrane.raven.log4j.SentryAppender
Dependencies	ActiveJPA	pom.xml

experience level of *above average* or higher (*very high*: 2). Only 2 participants rate their experience as *below average* and no one rated their experience as *very low*. Our participants represent a diverse group with different backgrounds.

3.2 Experimental Procedure

The empirical study we conducted with our participants consists of two different tasks and was supervised by one of the authors. We provided summaries and solution hints generated by Bart to our participants to study the *understandability* of build breaks. In the second task, we investigate whether Bart can speed up the fix.

First Task: Understandability: In the first task, our participants answered a questionnaire about the *understandability* of the build break summaries provided by Bart. We used Bart to generate summaries and solution hints for the broken builds of the mutated software components in Table 2 (*Task 1*) and asked our participants to evaluate them. We provided our participants with the following three questions and we asked them to answer on a five-point Likert scale from *very high* to *very low*:

- How much did your understanding of the build failure improve through the summary of the build log?
- To what extent do the suggested solutions help you in conceiving a strategy to solve the build failure?
- To what extent are the suggested solutions applicable to the specific build failure?

Second Task: Resolution Performance: In the second task, we measured the time it takes developers to fix a broken build to analyze the effect of Bart. Every participant was asked to fix two of the four manually injected bugs for Task 2. Specifically, each participant had to fix one bug with treatment (i.e., support through Bart) and another one from a different category without.

We have avoided learning effects between the two different fix attempts of each participant by changing the type of build failure and by changing the software component, in which the bug was introduced. In total, we tested eight scenarios and each of the four different build failures was fixed twice, once with and once without treatment. The participants always started with fixing the build failure without treatment. All participants managed to fix both assigned builds without external help.

One of the authors supervised the task and started the experiment with an introduction to Bart. Participants were then asked to import the assigned projects into their development environment. We were using a Jenkins build server, which, in case of a build failure, produces a build overview that indicates the build result (i.e., Failed), the last Git commit that was pushed to the remote repository, and the name of the committer. In addition, Jenkins provides access to the generated build log. The supervisor gave our participants time to get familiar with the projects and with the Jenkins instance that was used to build the projects. To start the fix attempt of the build failure, our participants were asked to trigger a new build of the assigned project and to repair the resulting build failure. The task supervisor measured the *resolution time*, i.e., the time between the build break and the next build success. The same methodology was applied for the second build fix attempt.

General Feedback: After finishing the experiment, our participants filled a survey and participated in semi-structured interviews, in which we asked them for their opinions on the build-failure summarization. The survey was focused on three main questions. The first question was about Bart's ability to provide assistance for build failure resolution. We discussed with our participants what they like or dislike about our approach. In the second question, we asked for the perceived benefits of using Bart. The last question asked whether our participants would integrate a tool like Bart in their regular CI pipeline.

All 17 participants filled out the survey, but only nine participants agreed to participate in a follow-up interview. We recorded and transcribed these interviews and then performed card sorting to analyze the answers to our open interview questions [42]. We started by splitting the answers into individual statements, grouped common arguments, and finally organized these arguments hierarchically. We will use statements from the participants later, when we discuss the results of our experiment, and indicate which participant made them (e.g., I3 is interviewee with id 3, *S5* is survey participant with id 5).

3.3 Understandability of Build Breaks

Our first research question was how build summarization can improve understandibility. To answer this question, we evaluate the ratings of our participants for the generated summaries of Bart. We visualize the answers in three *diverging stacked bar charts* [37] that illustrate their rating regarding the *understandability* of the summaries (Figure 4), their *relevance* (Figure 5), and

their *applicability* to the build break (Figure 6). We use the *Likert* values *very high*, *above average*, *average*, *below average*, and *very low*.

Understandability: Figure 4 shows how participants rate the understandability of BART's summaries compared to the raw build logs that are provided by JENKINS. All participants agree across the board that the understandability of the build break summaries is at least *above average*, with the majority saying that it is *very high*. Only in two cases, one for compilation and one for dependencies a participant found the build summary *below average* and *average* respectively. Specifically the participant found that BART's summary for the dependency error was comparable to the default build-log presentation in JENKINS and that it did not improve.

The developers seem to agree that BART's summaries helps them to better understand the build log. One of the participant describes the actual build logs as *"cryptic" (I8)*, which could be caused by a lack of experience in reading it. However, the overloading amount of information that is contained in a build log is a recurring theme in the answers of our participants, even from experienced developers. Another participant said that *"Maven logs tend to be verbose, having a quick summary [...] greatly reduces the time needed to find and correct a build failure" (I5)* and another one that *"[BART] helps to find the programming errors quickly" (I4)* and *"a structured summary is way easier to grasp than many unstructured lines of text" (I4)*.

> *Our participants almost unanimously agree that* BART*'s build summaries improve the understandability of build logs.*

Relevance & Applicability: Figures 5 and 6 illustrate relevance and applicability of the proposed solution hints from STACKOVERFLOW. The solutions hints for *compilation* and *code analysis* breaks were mostly positively rated. Most our participants found their relevance and applicability *above average*, more than half of them rated them *very high*. However, two participants found the applicability of the solution for the *code analysis* break *below average* and one of them, according to the background information a very skilled developer, has also considered the relevance of the solution *below average*. The one participant that has considered the solution hint for the *compilation* build break as *very low* has little programming experience and uses JAVA only occasionally. We assume that he simply did not understand the suggested hint.

> *Most study participants find the solution hints for build breaks caused by* compilation *and* code analysis *errors relevant and applicable.*

In case of the *dependency* build break, the participants do not agree on a rating for the relevance and applicability of the solution hints. The ratings are centered around *average*, some of the participants find the suggestions relevant and applicable (one participant considered it even *very high*), while others rate it *below average*. Two participants even think that the applicability of the solution hints is *very low*. One of them is no frequent JAVA user, but the other one has a very strong background in `Java` programming, so a lack of expertise alone does not explain the different ratings.

> *Suggested solutions for* dependency *errors are often not considered as valuable hints by our participants.*

Our respondents were also not convinced about the relevance and applicability of solution hints for *testing* build breaks. Most of our participants consider them *below average* or even *very low* when compared to the original build log. One possible explanation is that the available information in the build log of a failing *testing* build, i.e., the name of the failed test, is project specific. This makes it unlikely to find related solutions for such local errors in external resources without taking other information into account.

> *Testing-related build failures are project specific. The build log alone is not sufficient to identify related external resources.*

3.4 Resolution Time of Build Failures

Our second research question was whether Bart can reduce the time that is required to fix a build. To answer this questions, we have asked our participants to fix the failing builds of the second task while measuring the required time. Table 3 illustrates the results of this experiment. It shows the average time to repair the different build failure types with and without the support of Bart and the corresponding standard deviations. We computed the coefficients of variation (CV) [9] for all the combination of build failure types and fixing modalities. All our calculated CV values are smaller than the traditional threshold 1 (in fact, all values are < 0.5), which shows that the measured times are centered around the mean. For this reason we refer to the average values in the following. While the previous research question has revealed that the ratings for relevance and applicability of Bart's solution hints differ between the build break types, the results of the second task shows that using Bart leads to a substantial reduction from 20% to up to 62% in the required time to fix a build across all scenarios. We will discuss the different break types individually to explain what seems to be a contradiction at a first glance.

Code Analysis and Compilation: The study participants found that Bart's summaries improve understandability and that the solution hints are both relevant and applicable. The positive ratings can also be confirmed in the practical task. The time to fix a build break could be reduced by 62% for build breaks related to *code analysis* and by 20% for build breaks related to *compilation* errors.

The error messages of both *compilation* and *code analysis* are self-explanatory, but a certain degree of expertise is needed to understand them. Fortunately, the exact same error messages and warnings appear in other projects as well, so it is easy to find information online that provides context to understand the error message. Our solutions hints are able to enhance the description of a warning or even replace it and the developers get an explanation of an error or of a violated rule without having to look it up on

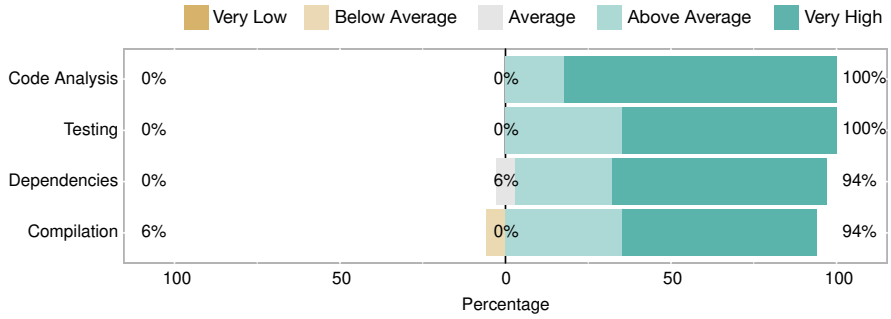


Fig. 4: Understandability of Summaries

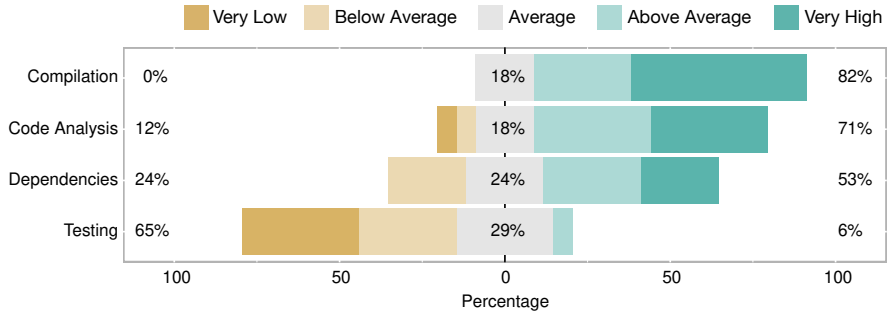


Fig. 5: Relevance of Proposed Solution

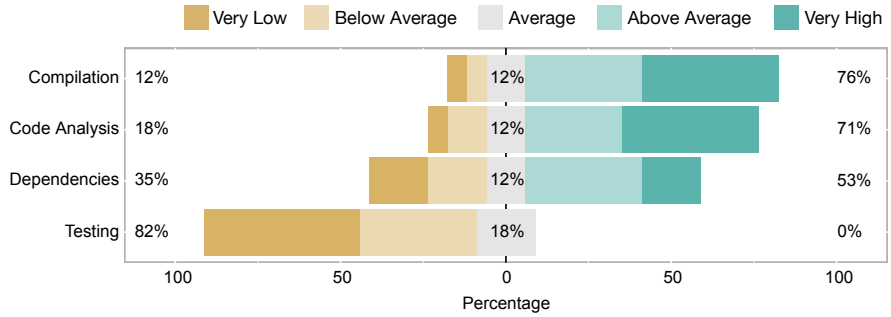


Fig. 6: Applicability of Proposed Solution

external resources. This aspect is particularly useful when the developer is not used to a specific code analysis tool.

Our participants found it very helpful that Bart integrates all required information to understand the meaning of a rule violation *"Less searching for the relevant part in the error message, hence faster bug resolution" (I2)*. Moreover, the links to StackOverflow are highly appreciated when the meaning of a warning is non-obvious. *"In the less obvious error causes, the stack overflow extracts prove to be very useful" (I5)*. In these cases, the StackOverflow discussion about the proper solution can provide additional context informa-

tion to understand the problem. The StackOverflow solution hints speed up the development process, because *"You can often get the information from bart without having to search the internet" (I6).*

> *In addition to the summary, solution hints can provide the required context that helps with understanding the cause of a build break.*

Dependency and Testing: The relevance and applicability of the suggested solution hints were not considered useful for *dependency* breaks and *testing* failures. These low ratings can easily be explained though. A search for the corresponding error message would either return many unrelated resources (e.g., cases in which other developers had trouble with some other dependency) or none (because the error message of a test failure is project specific). However, we could still see a substantially reduced fixing time for both categories. The improvement for *dependency* related build breaks (43%) has even been the second most significant reduction among all considered cases.

When considering error messages in both categories, it becomes apparent that both are typically very self-explanatory. The errors immediately point to the missing dependency or name the failing test. The required action to fix such issues is straightforward: search for the missing library in the *Maven Repository* and add it to the build file or fix the failing test, respectively. The participants that fixed such kind of breaks have reported that the reorganization of the information contained in the build logs significantly reduced the amount of time needed to understand the cause of a build break. One of our participants stated that *"[*Bart *is] mostly a timesaver, not really a skill enhancer. Carefully reading the log usually allows the extraction of the same information" (I5).* Another participant found that *"directly serving the relevant solution, the debugging process is drastically sped up" (I1).*

Another important aspect that affects the time to fix a build is the debugging environment. Previous work has shown [16] that CI server like Jenkins do not provide sufficient support to debug a build break. According to our participants, however, Bart summaries *"add more capabilities to the environment" (I6)* compared to the raw build logs and *"might make debugging unnecessary when the bug becomes evident" (I2).* They report that *"if some tests fail, the* Bart *output can be helpful in finding out why" (I5).*

> Dependency *breaks and* testing *failures seem to be easy to understand. Providing a good summary that highlights the locality of the issue seems to be the most crucial factor on fixing time.*

Overall, it seems that the different build break categories require different support strategies. Some categories benefit from links to external resources that provide additional context about an error (e.g., *compilation* errors), others benefit more from an improved summarization (e.g., *testing* failures). Bart combines both in one tool and substantially reduces the time to fix a build break across all scenarios in our study, on average by 37%.

Table 3: Resolution Time per Build Failure Type (Avg.: Average, Std. Dev.: Standard Deviation)

Failure Type	**excl. Bart (s)**		**incl. Bart (s)**		**Reduction (%)**
	Avg.	Std. Dev.	Avg.	Std. Dev.	
Testing	436	154	334	33	23%
Compilation	187	19	150	32	20%
Dependencies	223	74	127	16	43%
Code Analysis	280	109	106	39	62%
Overall Average					37%

3.5 General Feedback on Bart

Most of our participants (76%) benefited from using our build summaries during failure resolution. 53% reported that the big advantage provided by Bart is the speed up in solving those failures. One participant stated that *"especially with very large build logs and more complex errors bart really helps to reduce the time needed to figure out where the error happened" (I7)*. For 23% of our participants, the main advantage of using build failure summaries is an easier analysis of the root cause of the failure because *"a structured summary is way easier to grasp than many unstructured lines of text" (I4)*. 12% were not sure about Bart's benefits and another 12% did not think that Bart provides particular advantages. 63% like Bart because they have to read less information in order to fix a build failure. This is strictly connected to the most cited benefit provided by Bart, i.e., a faster build resolution. 16% also appreciated having links to relevant StackOverflow discussions that can point them to the right solution. They found those links useful especially when the problem is quite complex to solve as stated by a participant *"Additional links to related StackOverflow posts can be helpful in fixing more complex problems" (I9)*. 21% really like the overview of the failure provided by Bart that highlights the root cause of the failure. The last feedback that we received was about the applicability of Bart in the actual development context of the participants. 35% would immediately install our tool. 23% would like to install Bart, but they are not using Jenkins or Maven in their current projects. Furthermore, one participant (I4) believes that he would install Bart for big projects and another would use it only if there is *"there is no installation overhead" (I2)*. Finally, 12% would maybe install Bart, 6% do not know, and 24% do not want to install our tool mainly because it does not apply to their context.

4 The Developer's Perception of Build Fixing Practice

The results presented in the previous section show that Bart can provide assistance that improves the developer's performance when fixing build failures. Besides a summarization of the error cause, which seemed to be universally

helpful across all failure scenarios, BART also provides solution hints that help developers to derive fixing strategies. However, the results of our follow-up questionnaire have shown that the applicability of these solution hints differs between the failure categories. While they seem to be helpful for some failure categories (e.g., *compilation* and *code analysis*), the results are no longer unanimous for *dependency*-related failures, and our solution hints do not work for *testing*. To further investigate and explain this observation, we analyze the developers' perception of build failures. We first identify the characteristics of typical workflows for fixing build failures in various categories and answer RQ_3. After this, we answer RQ_4 by exploring what types of build failures developers consider more difficult to fix. We will use these findings to discuss the current limitations of our STACKOVERFLOW-based approach and we will envision future solution-hint generators.

4.1 Survey on the Perception of Build Failures

The *goal* of this study was to understand how people react to build failures and what types they consider hard to fix. In order to fulfill our goal, we sent out a survey. In the following, we describe our survey's questions and the demographics of our participants.

Structure: Our survey consisted of 15 questions and was divided into 4 sections. In the first section, *Background*, we collected the demographics of our participants. In the next two sections illustrated in Table 4, we surveyed developers on the difficulty of solving build failures and on their reaction to the most common build failure types. We wanted to identify whether it is harder to inspect the root cause of the build failures instead of understanding the problem and plan the fix (Q1.1) and for which types (Q1.2 and Q1.3). In Q2.1-Q2.4 questions, we asked our participants to reflect on their workflows for solving compilation, dependencies, testing, and code-analysis build failures. In the last section, we let our participants add opinions on the survey and report other build failure types that researchers should investigate more.

Recruitment: Our survey was implemented using GOOGLE FORMS. To recruit participants we posted our survey on the Question and Answer Site REDDIT [36], on which we targeted 15 specific subforums (so-called "subreddits"), namely `r/DevOps`, `r/LearnProgramming`, `r/WebDev`, `r/iOSProgramming`, `r/Docker`, `r/learnJava`, `r/Python`, `r/cpp_questions`, `r/dotnet`, `r/javscript`, `r/ruby`, `r/swift`, `r/coding`, `r/androiddev`, and `r/csharp`. We selected these communities because they (i) allow users to post surveys, (ii) have a large number of active subscribers, thus increasing our potential audience (e.g., the `r/DevOps` subreddit has approximatively 300 members that are online daily), and (iii) include members that frequently encounter build failures. We also sent an email to the mailing lists of MAVEN (`users@maven.apache.org`, `dev@maven.apache.org`), ANT (`user@ant.apache.org`), and JENKINS (`jenkinsci-users@googlegroups.com`). The sur-

Table 4: Survey Questions. (MC: Multiple Choice, O: Open answer)

Section	Summarized Question	Type	#
What makes it hard to fix a build?			
Q1.1	What typically takes longer, finding relevant information or understanding the problem to derive a fix?	MC	101
Q1.2	For which kind of build failures does it take long to find the relevant information?	O	101
Q1.3	For which kind of build failures does it take long to understand the problem and to derive a fix?	O	101
How you would react to a build failure notification in the following scenarios?			
Q2.1	Scenario 1: A compilation error broke the build.	O	101
Q2.2	Scenario 2: The build broke due to a dependency error.	O	101
Q2.3	Scenario 3: The execution of a test has failed.	O	101
Q2.4	Scenario 4: A build failed, because a quality concern was detected.	O	101

vey was available for 2 months in two different periods (from mid-October to mid-November 2018 and March 2019).

Demographics: A total of 101 respondents completed the entire questionnaire. Among all survey respondents, 88.1% work as professional developers. 3% are spare-time developers and 4% are students. The remaining part of our participants (4.9%) consists of build engineers, DevOps engineers, site-reliability engineers, and industrial researchers. 86.2% of our respondents rate their programming experience *High* or *Very High*. In most of the cases (58.4%), the highest qualification is a Bachelor Degree, while 16.8% hold a Master Degree, and 4% have a Ph.D. degree. Self-studied developers represent 14% of our respondents. Our participants report that 26.7% encounter build failures frequently and 6.9% very frequently. The majority (48.5%) reports encountering build failures occasionally.

4.2 How Do Developers Approach Different Types of Build Failures?

The fact that solution hints work for some categories of build failures, but not for others, suggests that developer approach these kinds of build failures differently. To better understand these differences, we have asked our participants to describe the typical workflow in which they would address several kinds of build failures, under the assumption that they do not know from the start what caused the problem. Participants could give an open answer and were supposed to illustrate the individual steps.

4.2.1 Methodology

To identify the typical workflows for the four main build failure categories that are covered by Bart, i.e., *compilation*, *dependencies*, *testing*, and *code*

analysis, we have conducted an open-coding approach [42] to encode the described workflows from the open answers of our participants. Given that the output of the coding is not simple labeling, but a complex graph describing the workflow, we had to adapt the traditional methodology. Two authors of this paper have performed the following steps to extract the workflows.

Common Vocabulary: The first step in our methodology was to establish a shared vocabulary, i.e., a set of activities that we accept in the extracted workflows. Both authors individually inspected 80 answers that span over 80 participants and all four scenarios to create two individual sets of activities. In a joined discussion, the authors merged both lists, eliminated duplicates, agreed on activity labels, and clarified overlapping cases. We ended up with the following list of activities. The high-level concepts are only used for grouping and are not used as labels.

Start & End: All workflows have the label `Error notification` as the start node and the label `End` to indicate a fixed build.

Locating: Activities in this category are related to understanding the error message and to finding further information about the error.

- Most of the time, developers open the build log, either via a web browser or through clicking a link in an email. The first step is then to `Locate the Error` in the log to understand the type of error.
- We extract an `Understand details` activity, every time the developer state that they look for information in the build log, like line numbers or test names.
- Many developers state that they `Reproduce the problem`, either locally on their development machine or on a build host, to collect further information.

Inspect Changes: Once the error message is understood, developers often continue to inspect the changes of the last commit or differences in the environments. We differentiate the following activities.

- The most common activity to understand the root cause of an error is to `Inspect code changes` that have recently been introduced.
- Another frequent activity is to `Inspect the Config` to understand recent changes or differences in, for example, tools and environment.
- Developers check the version control system to `Identify committers`, either to ask them for details or to delegate the fix of the build failure.
- Several developers state that they `Inspect process documentation` to understand the reason for a change, for example, through release notes, a changelog, or an issue tracker.
- Some developers mention that they `Inspect the build history` to extract historic information about past builds, like previously failing builds.

Investigation: Developers perform various steps to further investigate the failing build and understand more details about the error cause.

- Many developers `Read/Debug` the affected source code. We combine both activities in one node because it is often not clear which one is meant.

- Automated Static-Analysis Tools (ASAT) can provide diagnostic information about a build, e.g., the dependency graph. We extract a `Use ASAT` activity when developers apply them to find hints towards a solution.
- Sometimes, the build log is not enough and developers need to `Consult external logs`, e.g., other log files or screenshot of the failing application.
- Often as a measure of "last resort", developers perform a `Websearch` to find more information or an illustration of the problem online.
- In some cases, developers state that they `Lookup the documentation` to understand the meaning of a particular error message.
- To get further information about a change, it is sometimes necessary to `Contact an expert`. This person is typically either familiar with the affected part of the codebase or is the committer of the latest changes.

Solution Attempt: In the final phase of a workflow, developers state how they approach the fix. We differentiate between five alternative activities.

- The most obvious activity is when a developer states that, after understanding the problem, it is clear how to change the project to `Fix` the build.
- Various answers mention that they would use `Automated Fixes`, for example, an environment's clean-up, dependencies' update, or an automated refactoring.
- A re-occurring strategy is to `Delegate the Fix` to someone else, usually to the original developer who committed the change that caused the build failure.
- The answers contain descriptions of very strict workflows, in which developers `Revert commits` to fix the build failure as quickly as possible.
- Developers, that come to the conclusion that the build failure is either irrelevant or an incorrect state, `Ignore` the problem, e.g., by removing a failing test case or by ignoring a quality check.

Verify: Several authors describe steps to verify the solution, both locally and remotely. We use the label `Verify` to represent these cases.

Exclude: Several of the answers had to be excluded from our analysis. Instead of extracting a complete workflow, we then only extracted a single label.

- We extract the label `Not a problem in practice`, whenever participants state that this particular problem does not occur in their typical workflows.
- If we can not extract meaningful information (e.g., in case of an empty answer), we treat the entry as an `Invalid answer`.
- In several cases, participants are not able to explain how they would approach the problem in the stated scenario (`I don't know`).

This set of activities was stable during the open coding session. The authors did not find any description that could not be encoded with these labels.

Coding Rules: In a first training iteration, the authors have encoded the 80 answers (20 answers per scenario) that have been used in the vocabulary step. This step has first been performed individually. Afterward, the results have then been discussed and merged into a joined encoding. As part of this merge

discussions, the authors have agreed on a set of coding rules that decide corner cases, in which the authors disagreed in their separated coding sessions. The authors then performed a second training iteration separately (120 answers, 30 per scenario), using these rules to test their applicability. The resulting workflows were discussed again, agreed on, and the rules have slightly been refined. The authors ended up with the following set of rules.

1. The initial state of a workflow is always an `Error Notification`. The end state is always the `End` node, which indicates a fixed build. The participants do not know what has caused the error, so unless developers explicitly mention that they do not check, we include the `Locate Error` activity, through which they understand the type of error that has occurred.
2. `Locate Error` informs about the error type (e.g., the build failed due to a compilation problem), but it does not provide enough information for a `Fix` (i.e., `Locate error` → `Fix` is an invalid transition).
3. Just stating to read the log does not define any solution strategy and is considered an `Invalid Answer`. Also, ambiguous statements are considered invalid. If missing steps are implicit and clear from a mentioned strategy, we fill the gaps. However, incomplete cases (e.g., when a "branch" in a case distinction is missing) are also considered `Invalid Answer`.
4. We consider the answers of all four scenarios together, since some participants do not repeat details that are mentioned before.
5. Running the debugger requires to `Reproduce the Problem` first. Please note that `Read/Debug` can also just mean reading the code, which does not require to `Reproduce the Problem`.
6. The activity `Identifying committers` needs a reason. Typically, this is either done to `Delegate a Fix` or to `Contact the Expert` to understand further details.

These rules remained stable during the following open-coding session and both authors used them to encode all remaining answers separately.

Validation: A workflow consists of *nodes* and *directed edges*. The nodes have labels that correspond to the established vocabulary, edges connect nodes to indicate a sequence of activities. To make the individual workflows comparable, we wrote them down in the Dot syntax for directed graphs. For example, a "diamond" workflow that does `a` and then either `b` or `c`, but ultimately always `d`, can be expressed in the simple graph $G\ \{\ a{\rightarrow}b{\rightarrow}d;\ a{\rightarrow}c{\rightarrow}d;\ \}$.

After encoding all remaining 204 open answers separately, we validated the reliability of our process. We could not calculate the traditional *Inter-Rater Reliability* [7], because the resulting workflows are more complex than simple labels that are either correct or not. Instead, we calculated the similarity of the resulting workflows. We use a normalized hamming distance to calculate the similarity of two encoded workflows [38]. The contained nodes and edges of the graph build a set of graph elements, the more of these elements a workflow has, the larger it is. When comparing two workflows, the set-size of the larger workflow defines the maximum distance n_{max} between both workflows.

We count the number of elements n_{shared} that both workflows have in common. The *normalized* distance is then defined as $d = n_{shared}/n_{max}$, which is bounded by 0 (completely different workflows) and 1 (identical workflows).

After the coding phase of the remaining 204 open answers, we achieved a perfect agreement in 105 cases (51.5%). The workflows, in which we disagreed had an average similarity of 50.0%. Without looking at the other solution, both authors re-coded theses cases and many cases turned out to be trivial mistakes like forgetting the `End` node. After fixing these obvious mistakes, the agreement increased to an exact match in 145 cases (71.1%), with an average similarity of 53.3% for the workflows with disagreement. All remaining disagreements were solved in a joined coding session.

Aggregation: After agreeing on the sorting results for all 101 participants, we merged the individual workflow descriptions into joined representations. The general approach was to simply overlay all workflows and to count how often each of the nodes and edges occurs in the dataset. The result was a graph with high complexity that was caused by various infrequently named transitions. To remove rare activity sequences that blur the overall picture, we have filtered activity sequences (i.e., edges) that were described less than three times. A straightforward implementation of this filtering would just leave out these edges from the graph, which would not only result in inconsistent counts, but it can also result in invalid workflows with "gaps". To preserve this consistency and, at the same time, to preserve as much information from the workflows as possible, we performed a *branch elimination*: We checked for each workflow whether it contained any of the filtered edges. If found, the edge was removed and so were all the previous and following edges that were not reachable otherwise. As an illustration, consider the previous example graph G. Assuming that the edge $c \rightarrow d$ is rare and should be filtered. In this case, we would not only remove $c \rightarrow d$, but also $a \rightarrow c$, which would otherwise create a "loose end". We preserve the sequence $a \rightarrow b \rightarrow c$ in the graph, but count the workflow as a *partially filtered* case. In case the *branch elimination* removes all steps, we filter the whole workflow and count it as a *filtered* case.

The results of this aggregation are shown for the four different scenarios in Figures 7-10. Both the nodes and the edges contain counts that indicate how often the respondents have mentioned it. For the edges, we adapt the line width depending on the count to highlight the most common flows.

4.2.2 Extracted Workflows

In this section, we describe the workflows that we have identified for the main build failure categories. We illustrate the main steps that lead our participants to the resolution of the failures and we discuss the differences among the resulting workflows.

Compilation Failures (Figure 7)*:* After being notified developers typically locate the error that informs about the compilation failure. In many cases, they need to understand further details from log such as the line number where the

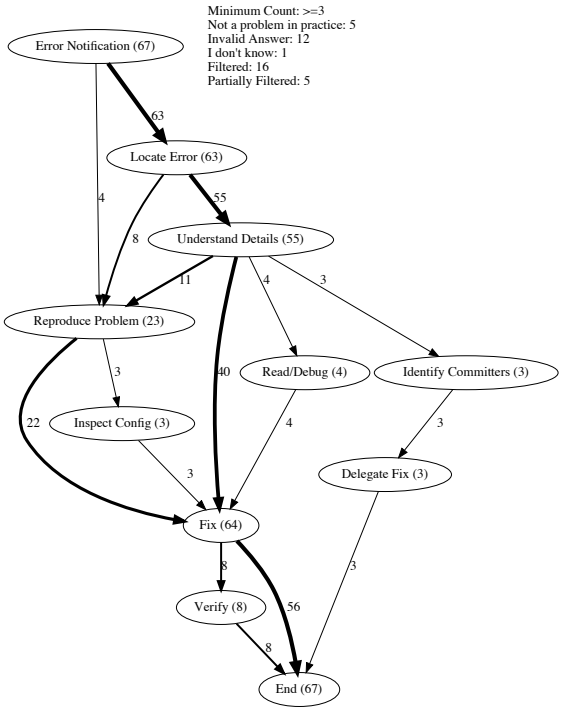


Fig. 7: The Process of Fixing *Compilation* Build Failures

error occurs. This is often sufficient to immediately fix the problem. However, a few more steps are sometimes needed. Developers reproduce the failure locally (even immediately after the error notification or locating the error) and inspect changes made to environments such as a recently-introduced new version of the compiler. Others read the source code to increase their knowledge about the part of the system affected by the error. A few developers identify the authors of the last commits included in the failed build and ask them to fix the failure. Finally, only 8 developers verify the fix before committing it. Overall, compilation failures are considered relevant by the majority of our participants. Only 5 of our participants do not encounter these types of failures.

Dependency Failures (Figure 8)*:* Understanding the type of failure is the first step performed by developers when they run across a dependencies failure. Before directly applying the fix, the majority of our developers investigate additional information in the build log such as which dependency is missing or broken. In some cases, the information contained in the log is not enough to fully understand the problem and fix it. Thus, developers reproduce the failure

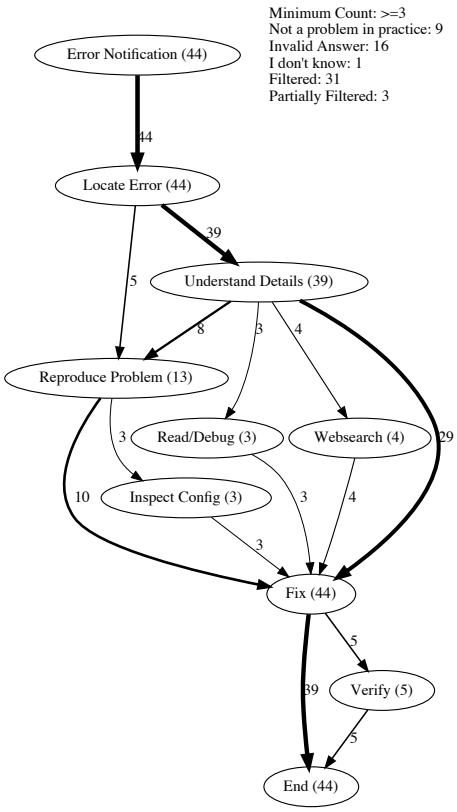


Fig. 8: The Process of Fixing *Dependencies* Build Failures

locally (some of them avoid to fully inspect the log before) and depending on the severity of the problem they look at the differences between the local and remote environments (e.g., the dependency is available locally but not on the remote server). Others read source code requesting the dependency reported in the log or they search the Internet for similar failures. Finally, in a few cases, they verify the applied fix before committing it. Also in this case, only 5 participants consider dependencies failures as a problem not occurring in practice.

Testing Failures (Figure 9)*:* Developers start fixing testing failures identifying the build failure type in the log. While a minority immediately reproduce the failure locally or start identifying people committing the last changes, many developers inspect the build log searching for the tests that did not succeed. Then 4 possible directions are taken before the problem is fixed. The majority of developers decide to investigate the test failure locally. They reproduce the

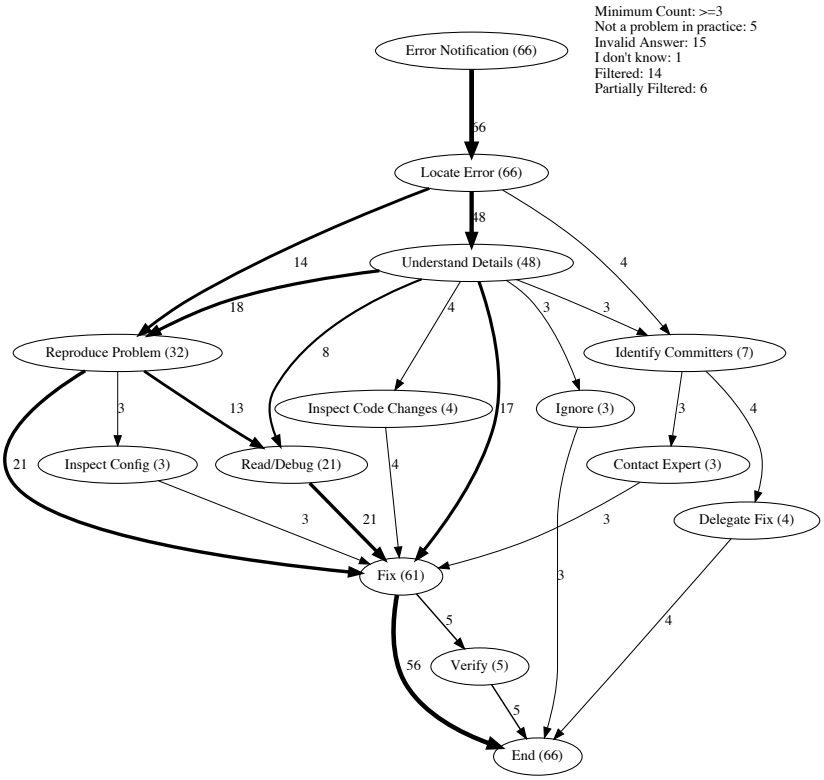


Fig. 9: The Process of Fixing *Testing* Build Failures

last build and, if needed, debug the failed test or inspect the configuration files. Other developers inspect the history of code changes to identify (i) which was the previous version of the test or the code under test and (ii) which were the previous developers working on them. The goal of identifying committers is two-fold. On the one hand, developers can contact them to ask questions about the failure and generally to receive support while fixing the test. On the other hand, developers can delegate the fix to them. Surprisingly another direction is to ignore the failure. This happens when the code under test is not used or the test is obsolete or flaky (see Section 4.3). The last option is to immediately fix the problem based on the information retrieved from the build log. In a few cases, the fix is also verified. Testing failures are experienced almost by all participants. Only 5 developers do not consider them a real problem in practice.

Code-Analysis Failures (Figure 10)*:* Code analysis are the failures with the highest percentage of "Not a problem in practice" answer (from 27 partici-

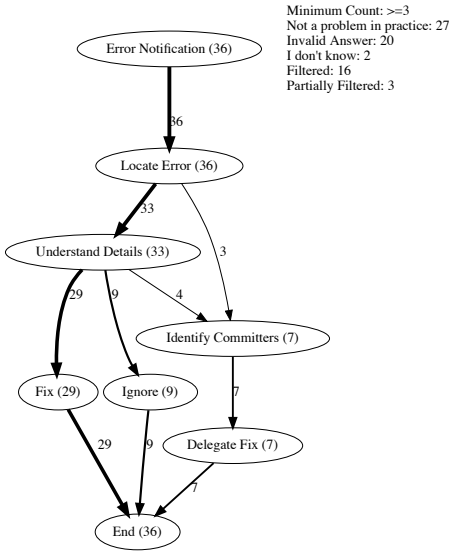


Fig. 10: The Process of Fixing *Code-Analysis* Build Failures

pants). These participants do not check for code quality during the build or, when it is checked, poor code quality is not set as a failing condition for the build. As regards the resolution workflow developers start investigating the type of the occurred failure. Then they either inspect the log to have more details about the warnings raised by the static analysis tools that provoked the failure or identify the committers of the last changes included in the failed build. The former step is used to judge the failure. If the violations are considered relevant developers apply the fix using the description available in the log. Otherwise, developers decide to ignore the violations and disable them in the configuration files of the invoked static analysis tools. The goal of the latter (that is sometimes performed without understanding more details from the log) is to delegate the fix to the authors of the last changes.

4.2.3 General Insights and Actionable Results

After discussing the four merged workflows separately for each scenario, we now reflect on observations that affect multiple of these workflows. Some interesting insights only appear when the various workflows are being compared.

Build failures are very rarely solved in a collaborative effort. Few developers mention that they get in contact with the authors of the "implicated" commits while fixing compilation, testing, and code-analysis failures. In the case of compilation and code-quality issues, the only purpose is to delegate

the fix to them. For some testing failures that they feel competent to fix, developers get in contact with the committers to have more information about the change causing the failure and to receive feedback about the resolution strategy. Developers always try to solve dependencies failures themselves.

Developers often reproduce the failure locally to ease the task of inspecting the root cause in the code. If a build failure cannot be reproduced some developers also start inspecting differences between local and remote environments, suspecting that they caused the failure. While for other build failure types understanding details from the log is often sufficient to fix the error, most of the developers reproduce a testing failure locally before fixing. This means that the typical textual description of the errors contained in the log is only a starting point for the investigation. To devise a resolution strategy developers also need to understand the semantic of the changes causing a testing failure and possibly debug the failure to identify bugs captured by the test. In such a workflow a better summarization of the build log would not help to plan a fixing strategy and solution hints are needed. However, websearch (which is the basis of our solution hints as described in Section 2.3) is rarely used and only to search for solutions to dependencies failures.

Code-analysis failures are considered "second-class" failures by many developers. They do not encounter build failures due to code-quality issues and such mistakes are typically unimportant/omittable and not worth scheduling during the build. Even when code-analysis is a proper step of the build, developers typically inspect the log not only to understand the issue causing the failure (that is typically easy to fix based on the description) but also to judge the violations and decide whether fix or ignore them.

Ignoring a failure is a much more common reaction for code-analysis failures. From the answers, the main reason seems to be that the high false-positive rate makes it necessary to ignore false warnings. Also, test failures are sometimes ignored, but here changing requirements are often named as the cause. The test is then ignored to give time to clarify requirements or fix implementation.

Developers often trust their devised solution. Only a few developers verify their solution, before integrating it into the master branch. This contradicts one of the key principles of CI, that is "run private builds" [8]. According to the principle, developers should emulate an integration build on their local workstation after committing their changes in order to prevent new build failures on the mainline.

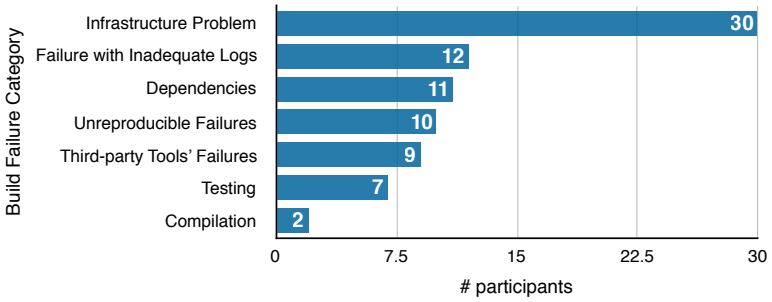
Coming back to our initial observation on how StackOverflow-based solution hints do not work for some categories, the most interesting insight from our study is that searching the internet for solutions is not a common practice. In the case of compilation, dependencies, and code-analysis failures developers often rely *only* on the information contained in the log. This confirms that a summarization approach pointing to the relevant information in the log can avoid the burden of inspecting thousands of lines of code and be sufficient to plan the fix. In the case of testing, developers need instead additional information to devise a fix strategy. However, our solution hints

were considered irrelevant. Based on the workflow in Figure 9, the reason is that developers typically reproduce the failure locally and debug it to guide the resolution process. Testing failures are too project-specific to search the internet for general solutions. Future extensions of BART should ease the reproducibility of build failures and its solutions hints provide developers with the semantic content of the change causing the test failure.

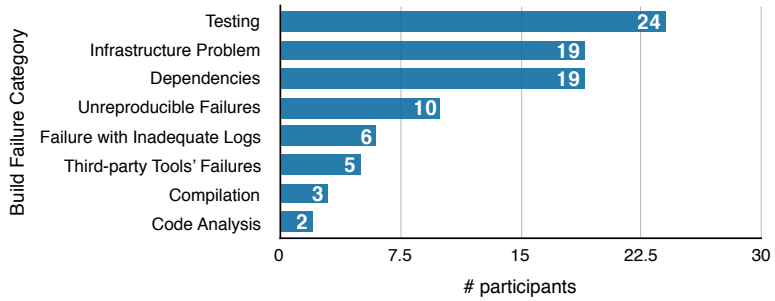
4.3 What Types of Build Failures Are Hard to Fix?

In the previous section, we have analyzed the most common workflows while solving build failures. Our participants typically start collecting relevant information about the problem and then try to understand it. In most of the cases build logs are enough to locate the error while especially in case of testing failures developers also need to reproduce the failure locally. As the answer to Q1.1, 62.4% of our respondents believe that understanding the problem is the hardest part. Instead, 19.8% think that finding relevant information is the most time-consuming activity, while 17.8% spend the same time both on collecting information relevant to the failure and on understanding the problem. We further investigate those two phases of the build resolution process and ask our participants in Q1.2 and Q1.3 which build failures are more difficult to inspect (i.e., require more time to collect information about their root causes) or which ones are instead harder to understand. We performed card-sorting to identify the recurring concepts in the answers to these questions and obtained the results presented in Figures 11a and 11b.

Failures Hard to Inspect: Most of our participants consider *infrastructure problems* as the hardest to inspect (Figure 11a). They struggle to locate misconfigurations of the CI workflow that causes failures (e.g., the CI server is not able to fetch new changes from the remote repository) or locating errors that are caused by a poor definition of the build scripts. The second most cited failures (*failures with inadequate logs*) in Figure 11a do not relate to a particular category of failures. Failures sometimes happen on servers that are external to an organization, producing logs that cannot be accessed by developers. And build logs are often simply not enough to fully locate the cause of the failure. In the presence of *Dependency* breaks, looking for conflicting dependencies takes a long time for 11 participants. Also the fourth most-mentioned failure (*Unreproducible failures*) is not specific to a particular failure category. These failures cannot be reproduced on the local machine of the developers and their build log is often not sufficient to fully locate the failure's cause. Developers need to replicate the error locally to generate additional log statements (e.g., by making MAVEN logging more verbose with the `-v` option) or reports (e.g., JUNIT reports). Reproducing failures is sometimes impossible because of a significant difference between developer's and the remote build's environments. Other failures that developers find hard to investigate are caused by defects of the *third-party tools* that are used during the build. These failures are usually caused by unmet requirements (e.g., a specific version of the compiler is



(a) ... Finding Relevant Information



(b) ... Understanding the Problem.

Fig. 11: Build Failure Types Our Participants Struggle-with while ...

missing). Several participants reported *test* and *compilation* failures without providing a specific reason. Finally, 8 participants answered that looking for the error cause is straightforward (independently from the build failure type) while other 2 spend a lot of time in understanding all types of build failures.

Failures Hard to Understand: Figure 11b illustrates the build failures for which our participants state that understanding is more difficult when deriving a fix. Despite finding the error that causes a *testing* failure is hard only for 7 participants (see Figure 11a), those failures are mentioned as the hardest to figure out. In this category, many developers mention integration and flaky tests. The second hardest type of failure is *infrastructure problems*. Developers report misconfigurations of the CI workflow and poor-defined build scripts as reasons. Many developers spend a long time not only on locating the outdated dependency but also on finding the right fix strategy. This is the reason why *Dependency* failures are also considered hard to fix. As in Figure 11a, also in Figure 11b 10 developers report *unreproducible failures*. The impossibility to reproduce the failure not only makes it hard to locate the error, but slows the understanding process down. Developers cannot verify (and refine) a hypothesized fix strategy without running a new build on the remote server. Third-party tools' failures sometimes require to replace a particular tool due

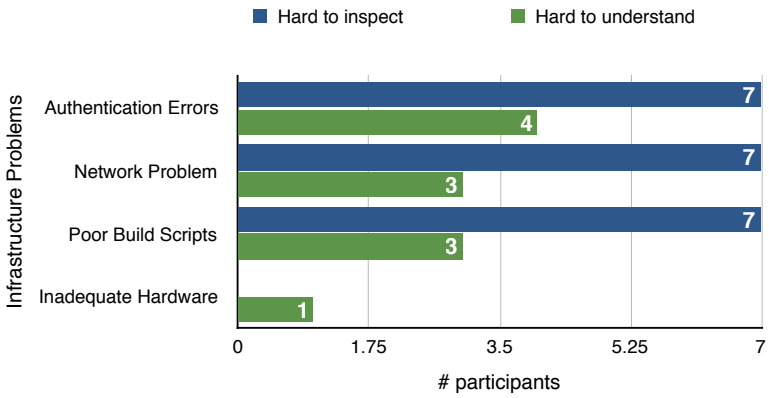


Fig. 12: Root causes of infrastructure-related build failures.

to incompatibility with others used during the build or, in case of defects, as a temporary fix before the next update. This can take a long time and involve multiple developers. Finally, a few participants mention also *compilation* and *code-analysis* failures as the most time-consuming to understand. Only four participants consider fixing all build failures a trivial task while one considers all types of build failures difficult to fix.

Cross-cutting Failures: While the others can be directly mapped to known build failure types [51], *infrastructure problems*, *failures with inadequate logs*, and *unreproducible failures* are cross-cutting categories that are not associated with a particular stage of the build process. A *failure with inadequate logs* indicates the problem of having build failure logs that do not contain enough information to start inspecting the failure's cause (sometimes the build log is not accessible). *Unreproducible failures* are build failures that cannot be reproduced in the developer's machine. *Infrastructure problems* are heterogeneous and are also perceived by many developers as both hard to inspect and understand. Figure 12 illustrates which are the *infrastructure problems* reported by our participants as answers to Q1.2 and Q1.3. Note that the figure does not include answers that refer generally to infrastructure problems without specifying the problem.

Different machines are involved in a CI workflow. Some of them are also not directly controlled by an organization. *Authentication errors* provoked by wrong credentials or *network problems* bringing about the unavailability of servers are mentioned by several participants as causes of build failures. Poor build scripts (e.g. scripts which contain hard-coded credentials or other security smells, can be susceptible to security breaches [33]) contribute to generating build failures. One participant mentions *inadequate hardware*. It can slow down the development process and generate server timeouts.

Discussion of Painful Build Failures: Previous work [51] has shown that the most recurrent types of build failures are compilation, dependencies, testing,

and code-analysis failures. While only a few participants mention compilation and code-analysis, our study on the perception of build failures reveals that testing and dependencies failures are also considered particularly hard to understand. These results confirm the importance of assisting developers during these types of failures and especially in case of testing failures that are considered as the most difficult among all the mentioned failures. Our build summarization approach increases the understandability of both categories. Together with a few "known" categories, new ones emerged from our analysis as the most painful categories. Those failures do not relate to a specific build phase and are not the direct consequence of committed code changes. They are caused by choices made while configuring the build process.

Developers use configuration files to define which tools are needed during the build and which are the requirements of the environments where the build is performed [13]. Depending on how the build process has been configured developers can encounter failures that are related to the infrastructure, i.e., the way those tools are connected and invoked during the build process. As shown in Figure 12, those failures can be caused by authentication or network errors between connected tools that are often hosted on different servers. Poorly defined build scripts or configuration files that define the build process lead to the introduction of smells, that are symptoms of deeper problems in the build process definition [13]. For example, build smells are indicative of security weakness [33] or misuse of CI features [13]. Based on our results, the presence of these smells also decreases the understandability of build failures. In the presence of less comprehensive build scripts, developers have difficulties understanding what to change in the configuration to resolve the failure.

Developers can control the amount of information that is generated by tools invoked during the build process through log statements. Having short logs reduces the amount of information developers have to read, but increases the time needed to find relevant information and makes the failures even harder to be understood. Our summarization approach tackles this problem pointing developers directly to relevant information in verbose logs.

Developers can also configure the environment, in which the build is performed. Those environments are usually very different from the local environment, in which developers implement code changes. For example, using several operating systems or different versions of a compiler. Reproducing an error is a crucial step towards understanding, but when a failure occurs on the build server, developers find it very difficult to reproduce the error in cases, in which local and remote environments are significantly different.

> *Among the most recurrent types of build failures, testing and dependencies are considered hard to fix. Furthermore, other failures that are not build-phase specific and are instead caused by the misconfiguration of the build process are considered painful.*

5 Discussion

In this section, we discuss the main findings of our study and their implications for researchers and practitioners.

For the RQ_1, we found that summaries improve the understandability of the build logs as it happens for source code [29]. Developers unanimously agree that build summaries improve the understandability of build logs across the most recurrent categories of failure.

In RQ_2, we discovered that improving the understandability of the build failures has a direct impact on the time needed to solve them. Across all the categories covered in our study, build summaries speed up the build fixing process: the time to fix is reduced by 23% in case of testing failures, by 20% in presence of compilation errors, by 43% in case of dependency-related issues, and even 62% in case of code violations. On average, developers using our tool spend 37% less time on solving build failures.

Among the hints included in our summaries, there is a category of hints that are generated by mining build failure solutions from StackOverflow discussions. The applicability and relevance of the proposed solutions are quite high in case of compilation and code analysis, less in case of dependencies failures. Furthermore, most of the times the solutions proposed for testing failures are not considered valuable. In RQ_3, we found that this is mainly caused by the lack of semantic information. Testing failures are project-specific and search for the root cause on the internet is not effective.

Although solution hints from the internet are ineffective, in RQ_4 we observed that developers struggle mainly while fixing testing and infrastructure-related failures. They find it difficult to derive a solution strategy for those failures and more effective solution hints that leverage other sources than the internet might be beneficial. Infrastructure-related failures do not refer to a particular build step and they are not directly caused by committed changes. They are issues provoked by the *smelly* configuration of a CI process.

Implications and Future Work: Our findings have important implications for both researchers and vendor of CI servers. Existing tools provide a build overview, but refer developers to the build logs for details, for example, to understand the reason for a build failure. Our results suggest though that build logs are difficult to understand and that integrating summaries of the build failure into the build overview can support the comprehension process.

Despite not automating manual activity, we show that providing solution hints that link to external resources (i.e., different from the build logs) can be useful to developers. They can provide additional context that can help to derive a solution strategy, especially when the root cause of a build failure is unclear or when the solution is non-trivial. So far, our infrastructure only considers StackOverflow discussions as external resource. Future hint generators could consider other resources, like generated reports or information about deployed libraries, to create a more holistic picture of the failure.

A better context awareness of the summarization tool might help to overcome existing limitations, e.g., solution hints for testing failures. Based on the build fix workflows analysis, developers frequently inspect previous changes to understand the cause of the failure or simply for learning from solutions applied to the same problems in the past. We argue that historical information and the semantics of a change can be an important source of hints for suggesting fixes to build failures.

This work introduces a technique to support developers when fixing a build break by providing them with summarization and solution hints. However, some build breaks cannot be reproduced locally and need to be solved on the server. Future work should investigate new ways of bridging this gap by considering differences between the remote CI environment and the local IDE environment when searching for solution hints. Novel debuggers, tailored to CI workflows, might help to improve the effectiveness when fixing build breaks.

Code-analysis failures are typically perceived as easy to fix, yet, they are the ones that benefit most from the adoption of our tool. Our participants also consider solution hints for code-analysis failures to be very relevant in the build-fix process. A comparison to other failure types indicates that developers tend to first judge the code-analysis failure, to decide whether the failure is worth fixing or whether it should be ignored. Solutions hints can support this decision by providing more details about the warning. For example, in case of the violation 'line is too long', a link to a related discussion of appropriate line length might help the developer to decide whether the warning is indeed a violation. Tool vendors can further support this step by ensuring that violation descriptions contain relevant contextual information.

For specific types of build failures, i.e., infrastructure-related failures, that are caused by weaknesses of the adopted development pipeline rather than by the change committed by a developer future hint generators should be conceived. In particular, by extracting information from different nodes (e.g., servers) of the pipeline those generators must be able to provide developers with the location of the error.

Assistance tools like Bart do not only have a positive effect on the developer that fixes the build. Supporting the individual developer has the potential to increase team productivity, because it reduce the team downtime that would normally be caused by the build break. Future work should investigate novel build log summarization techniques to reduce the required time even further.

6 Threats to Validity

The work presented in this paper was carefully planned and executed, but several threats to validity exist for our results. In the following, we will discuss them and our mitigation strategies.

Threats to *internal validity* concern the confounding factors that might have affected our results. We did not deploy our tool in a real industrial environment and only created versions of the software with artificial errors to

evaluate our tool, which might not be representative of real errors. We tried to mitigate this by injecting bugs that resemble the most common causes of build breaks [40,51]. Also, injecting realistic faults is a common trade-off in the design of many other studies (e.g., [31], [11]), so we strongly believe that our setting is valid. Another aspect that might affect the reliability of our results is the complexity of systems considered during the analysis. We tried to reduce this threat by considering build breaks in our study, which belong to projects that are not too big, but at the same time representative of real systems. It is also possible that our participants didn't fully understand the questions in our questionnaire. We have reserved time before starting, to allow participants to ask questions about the experimental procedure. Another threat is the manual time measurement that could introduce a bias. However, the substantial differences that we have measured far exceed the imprecision of the manual measurement. Other build summarizers might exist and requiring our participants to read a plain-text build log could introduce a bias in our experiment. However, we are not aware of any frequently used summarization tools and we think that using the information that is available in a standard Jenkins installation represents a valid baseline for our comparison. The subject understands the treatment. However, we could not really hide the treatment. Given that all participants have experience with Maven, every solution different from inspecting the raw build log would have been considered the treatment. In CI typically breaks are caused by recent changes, but our subjects do not have access to the history. However, breaking changes are hidden in big commits that tangle several changes, so a developer must first understand the error in order to know what to look for in the commit. Learning effects might blur the results because Bart is always used in the second task. We took strategies to mitigate this potential bias. Participants have been coached in the beginning and we made sure that all participants have the required knowledge to fix build problems (in particular, all participants have used Maven before). Developers have worked on tasks that affect different areas in the systems and the workflows to address these problems differ, so we expect low learning effects.

Threats to *external validity* concern the generalizability of our findings. We considered only four types of build breaks in our study. However, those represent the most relevant and recurrent categories of build breaks that have been observed [26,40,6]. Furthermore, the participants in our study could be unrepresentative of all kind of developers. We mitigated such threat trying to reach for both survey and controlled experiment people with different programming skills, to make general consideration about beginner and expert developers. Our tool, Bart, is the first implementation of an approach for build logs summarization. The current design of our study only looks at errors introduced by users. Future work should expand the scope and investigate build errors that are caused by the environment of the build server (e.g., different locale settings). The results presented in this work might not generalize beyond the considered build failure types.

7 Related Work

This paper is related to three lines of research: works on build failures, source-code summarization, and mining Q&A sites. In the following, we will discuss the most related previous works from these areas and relate them to the work presented in this paper.

Build Failures: Prior studies have investigated the nature of build breaks. Miller [26] found that the most recurrent causes of build failures in Microsoft projects are poor code quality, testing failures, and compilation errors. Other researchers [6,35] studied the frequency of different build failure types in open-source projects, finding that builds generally fail because of failed test cases. In our study, we focused on the most common build failure types according to those studies. While several works focused on one particular type of build failure, e.g., code analysis [56] or compilation [40], Vassallo et al. [51] proposed a broader taxonomy of build failures. They have analyzed 418 Java-based projects from a financial organization and 349 Java-based OSS projects and have identified differences and commonalities of failures between industrial and open-source projects. Because we summarized Maven build logs of Java projects, we decided to reuse this taxonomy to categorize our build failures. Kerzazi et al. [19] have analyzed 3,214 builds in a large software company over a period of 6 months to investigate the impact of build failures on the development process. They observed a high percentage of build failures (17.9%), which aggregate to a cost of more than 2,000 man-hours when each failure needs one hour of work to be fixed. Vassallo et al. [48] found that the problem of fixing build failures is even becoming more relevant in open-source projects. Developers often cause build failures on the release branch and it takes (on median) 17 hours to them. Some of these failures are noisy and complex [12]. Given that some builds have a misleading or incorrect status, developers are required to carefully inspect the build outcome to verify the presence of passing jobs. Thus, build failures slow down the release pipeline and decrease team productivity because they interrupt implementation activities [52]. This was one of the motivations for our study: providing developers with a tool able to support them while fixing build failures making the recovery process faster.

Existing plugins try to achieve the same goal. For example, Log Parser [1] is a Jenkins plugin that allows developers to add custom parse rules in the form of regular expressions. Matching parts of the build log are then highlighted for the developer. Bart pursues a different goal. It automatically selects the relevant information with no effort required from developers and organizes this information in summaries and by linking external information. Researchers proposed approaches to automatically repair builds that break due to dependency [23], testing-related issues [45], and that can be fixed by generating patches that are applied to buggy build scripts [15,22]. The focus of Bart is *developer-oriented* and complementary to automated approaches. We assume that very often developer interaction is required to fix a build.

Therefore, we try to empower the developer by improving build log understandability though summarization and linking to external resources.

Source-Code Summarization: During their regular work, developers have to cope with a large amount of external data [14], e.g., bug reports or source code, which is produced during software development. They need support while trying to comprehend such data and summarization techniques can facilitate this process. Several techniques have been proposed to summarize source code [29]. Haiduc et al. [14] proposed automatic source code summarization leveraging the lexical and structural information in the code. Moreno et al. [27] conceived a technique to automatically generate human-readable summaries for JAVA classes relying on class and method stereotypes in conjunction with ad-hoc heuristics. Other approaches generate summaries from source code artifacts, such as code fragments [55] or code usage examples [28]. Moreover, Panichella et al. [31] studied the impact of test case summaries on the number of fixed bugs, proposing an approach that automatically generates summaries of the portion of code exercised by each individual test. Other researchers focused on the summarization of build reports [34] or user reviews [41]. Our approach complements these approaches and presents a novel summarization approach for another important software development artifact, i.e., the build log.

Mining Q&A Sites: Question and answer websites like STACKOVERFLOW have been analyzed by several researchers to provide developers with helpful data during software development. Ponzanelli et al. [32] enhance the IDE with PROMPTER, a tool that automatically captures the code context in the IDE to propose related STACKOVERFLOW discussions. BART is very similar to this work, it is integrated into the build server and acquires contextual information about failing builds to assist developers with deriving a fix. Other researchers, investigated the impact of using STACKOVERFLOW on development workflows. Vasilescu et al. [46] analyzed the interplay between STACKOVERFLOW activities and code changes on GITHUB. While a switch to STACKOVERFLOW interrupts the coding, they were able to show a correlation between visits of STACKOVERFLOW and code changes. Developers seem to frequently switch between their IDE and STACKOVERFLOW when they get stuck, which supports our assumptions of Section 2. Finally, other researchers generated summaries for JAVA classes [54] and methods [47] by mining source code descriptions on STACKOVERFLOW. We also extract information from STACKOVERFLOW but follow a different goal, i.e., providing hints for build fixes.

8 Summary

This paper presented BART, a system that supports developers in understanding build failures and effectively fixing them. BART works on the build log, summarizes build failures, and provides solution hints using data from STACKOVERFLOW. We conducted an empirical study with 17 developers to assess the effect of BART on repairing build breaks. Our results show that developers find

BART useful to understand build breaks and that using BART substantially reduces the time to fix a build break, on average by 37%. To explain observations in our experiment, we have also conducted a qualitative study to better understand the workflows and information needs of developers fixing builds. We found that typical workflows differ substantially between various error categories and that several uncommon build errors, for example, errors related to the infrastructure, are very hard to investigate and to fix. These findings will be useful to inform future research in this area.

Acknowledgements We would like to thank all the study participants. C. Vassallo and H. Gall acknowledge the support of the Swiss National Science Foundation for their project SURF-MobileAppsData (SNF Project No. 200021-166275).

References

1. Log parser plugin. `https://wiki.jenkins.io/display/JENKINS/Log+Parser+Plugin`. Accessed: 2018-02-08
2. Active JPA: A Simple Active Record Pattern Library in Java that Makes Programming DAL Easier. `https://github.com/ActiveJpa/activejpa/`. Accessed: 2018-02-08
3. BART: Jenkins-Plugin. `https://plugins.jenkins.io/bart`. Accessed: 2019-07-24
4. Bavota, G., Gravino, C., Oliveto, R., De Lucia, A., Tortora, G., Genero, M., Cruz-Lemus, J.A.: Identifying the weaknesses of uml class diagrams during data model comprehension. In: Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems, MODELS'11, pp. 168–182. Springer-Verlag, Berlin, Heidelberg (2011). URL `http://dl.acm.org/citation.cfm?id=2050655.2050673`
5. Beller, M., Bholanath, R., McIntosh, S., Zaidman, A.: Analyzing the state of static analysis: A large-scale evaluation in open source software. In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 470–481 (2016). DOI 10.1109/SANER.2016.105. URL `http://dx.doi.org/10.1109/SANER.2016.105`
6. Beller, M., Gousios, G., Zaidman, A.: Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In: International Conference on Mining Software Repositories (2017)
7. Cohen, J.: A coefficient of agreement for nominal scales. Educational and psychological measurement **20**(1), 37–46 (1960)
8. Duvall, P., Matyas, S.M., Glover, A.: Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley (2007)
9. Everitt, B.: The Cambridge dictionary of statistics. Cambridge University Press, Cambridge, UK; New York (2002). URL `http://www.worldcat.org/search?qt=worldcat_org_all&q=052181099X`
10. Fongo: Faked Out In-Memory Mongo for Java. `https://github.com/fakemongo/fongo/`. Accessed: 2018-02-08
11. Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated unit test generation really help software testers? a controlled empirical study. ACM Transactions on Software Engineering and Methodology (TOSEM) **24**(4), 23 (2015)
12. Gallaba, K., Macho, C., Pinzger, M., McIntosh, S.: Noise and heterogeneity in historical build data: an empirical study of travis CI. In: ASE, pp. 87–97. ACM (2018)
13. Gallaba, K., McIntosh, S.: Use and misuse of continuous integration features: An empirical study of projects that (mis)use travis ci. IEEE Transactions on Software Engineering pp. 1–1 (2018). DOI 10.1109/TSE.2018.2838131
14. Haiduc, S., Aponte, J., Marcus, A.: Supporting program comprehension with source code summarization. In: ICSE (2) (2010)
15. Hassan, F., Wang, X.: Hirebuild: an automatic approach to history-driven repair of build scripts. In: ICSE, pp. 1078–1089. ACM (2018)

16. Hilton, M., Nelson, N., Tunnell, T., Marinov, D., Dig, D.: Trade-offs in continuous integration: Assurance, security, and flexibility. In: Proceedings of the 25th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2017, p. To Appear (2017)
17. Hilton, M., Tunnell, T., Huang, K., Marinov, D., Dig, D.: Usage, costs, and benefits of continuous integration in open-source projects. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 426–437 (2016)
18. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley Professional (2010)
19. Kerzazi, N., Khomh, F., Adams, B.: Why do automated builds break? an empirical study. In: 30th IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 41–50. IEEE (2014). DOI 10.1109/ICSME.2014.26. URL `http://dx.doi.org/10.1109/ICSME.2014.26`
20. LaToza, T.D., Venolia, G., DeLine, R.: Maintaining mental models: A study of developer work habits. In: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pp. 492–501. ACM, New York, NY, USA (2006). DOI 10.1145/1134285.1134355. URL `http://doi.acm.org/10.1145/1134285.1134355`
21. Likert, R.: A technique for the measurement of attitudes. Archives of psychology (1932)
22. Lou, Y., Chen, J., Zhang, L., Hao, D., Zhang, L.: History-driven build failure fixing: how far are we? In: ISSTA, pp. 43–54. ACM (2019)
23. Macho, C., McIntosh, S., Pinzger, M.: Automatically repairing dependency-related build breakage. In: Proc. of the International Conference on Software Analysis, Evolution, and Reengineering (SANER), p. To appear (2018)
24. Maple, S.: Java tools and technologies landscape report 2016. ZeroTurnaround post (2016). URL `https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016/`
25. Maven. `http://maven.apache.org/`. Accessed: 2018-02-08
26. Miller, A.: A hundred days of continuous integration. In: Proceedings of the Agile 2008, AGILE '08, pp. 289–293 (2008)
27. Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L.L., Vijay-Shanker, K.: Automatic generation of natural language summaries for java classes. In: ICPC, pp. 23–32. IEEE Computer Society (2013)
28. Moreno, L., Bavota, G., Penta, M.D., Oliveto, R., Marcus, A.: How can I use this method? In: ICSE (1), pp. 880–890. IEEE Computer Society (2015)
29. Moreno, L., Marcus, A.: Automatic software summarization: the state of the art. In: ICSE (Companion Volume), pp. 511–512. IEEE Computer Society (2017)
30. Myers, G.J.: The art of software testing (2. ed.). Wiley (2004)
31. Panichella, S., Panichella, A., Beller, M., Zaidman, A., Gall, H.C.: The impact of test case summaries on bug fixing performance: an empirical investigation. In: ICSE, pp. 547–558. ACM (2016)
32. Ponzanelli, L., Bavota, G., Penta, M.D., Oliveto, R., Lanza, M.: Mining StackOverflow To Turn The IDE Into A Self-Confident Programming Prompter. In: MSR (2014)
33. Rahman, A., Parnin, C., Williams, L.: The seven sins: Security smells in infrastructure as code scripts. In: 41st International Conference on Software Engineering (ICSE). IEEE/ACM (2019)
34. Rastkar, S., Murphy, G.C., Murray, G.: Summarizing software artifacts: a case study of bug reports. In: ICSE (1), pp. 505–514. ACM (2010)
35. Rausch, T., Hummer, W., Leitner, P., Schulte, S.: An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In: Proceedings of the 14th International Conference on Mining Software Repositories, MSR'17, p. nn. ACM, New York, NY, USA (2017)
36. Reddit. `https://www.reddit.com/`. Accessed: 2018-02-08
37. Robbins, N.B., Heiberger, R.M.: Plotting likert and other rating scales. In: Proceedings of the 2011 Joint Statistical Meeting, pp. 1058–1066 (2011)
38. Robinson, D.: An Introduction to Abstract Algebra. De Gruyter textbook. Walter de Gruyter (2003). URL `https://books.google.it/books?id=Yj3ApD8TeCUC`
39. Sentry Java: A Sentry SDK for Java and other JVM languages. `https://github.com/getsentry/sentry-java/`. Accessed: 2018-02-08

40. Seo, H., Sadowski, C., Elbaum, S.G., Aftandilian, E., Bowdidge, R.W.: Programmers' build errors: a case study (at Google). In: Proc. Int'l Conf on Software Engineering (ICSE), pp. 724–734 (2014). DOI 10.1145/2568225.2568255. URL http://doi.acm.org/10.1145/2568225.2568255
41. Sorbo, A.D., Panichella, S., Alexandru, C.V., Shimagaki, J., Visaggio, C.A., Canfora, G., Gall, H.C.: What would users change in my app? summarizing app reviews for recommending software changes. In: SIGSOFT FSE, pp. 499–510. ACM (2016)
42. Spencer, D.: Card sorting: Designing usable categories. Rosenfeld Media (2009)
43. StackOverflow: "Maven". https://stackoverflow.com/questions/tagged/maven. Accessed: 2018-02-08
44. Treude, C., Barzilay, O., Storey, M.A.: How do programmers ask and answer questions on the web? (nier track). In: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pp. 804–807. ACM, New York, NY, USA (2011). DOI 10.1145/1985793.1985907. URL http://doi.acm.org/10.1145/1985793.1985907
45. Urli, S., Yu, Z., Seinturier, L., Monperrus, M.: How to design a program repair bot? insights from the repairnator project. CoRR **abs/1811.09852** (2018)
46. Vasilescu, B., Filkov, V., Serebrenik, A.: Stackoverflow and github: Associations between software development and crowdsourced knowledge. In: SocialCom, pp. 188–195. IEEE Computer Society (2013)
47. Vassallo, C., Panichella, S., Penta, M.D., Canfora, G.: CODES: mining source code descriptions from developers discussions. In: ICPC, pp. 106–109. ACM (2014)
48. Vassallo, C., Proksch, S., Gall, H.C., Penta, M.D.: Automated reporting of anti-patterns and decay in continuous integration. In: ICSE, pp. 105–115. IEEE / ACM (2019)
49. Vassallo, C., Proksch, S., Zemp, T., Gall, H.C.: Un-Break My Build: Assisting Developers with Build Repair Hints. In: International Conference on Program Comprehension (2018)
50. Vassallo, C., Proksch, S., Zemp, T., Gall, H.C.: Replication Package for "Every Build You Break: Developer-Oriented Assistance for Build Failure Resolution" (2019). DOI 10.5281/zenodo.3346615. URL https://doi.org/10.5281/zenodo.3346615
51. Vassallo, C., Schermann, G., Zampetti, F., Romano, D., Leitner, P., Zaidman, A., Penta, M.D., Panichella, S.: A tale of CI build failures: An open source and a financial organization perspective. In: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017, pp. 183–193 (2017). DOI 10.1109/ICSME.2017.67. URL https://doi.org/10.1109/ICSME.2017.67
52. Vassallo, C., Zampetti, F., Romano, D., Beller, M., Panichella, A., Di Penta, M., Zaidman, A.: Continuous delivery practices in a large financial organization. In: 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 41–50 (2016)
53. Vos, T.E.J., Tonella, P., Prasetya, W., Kruse, P.M., Bagnato, A., Harman, M., Shehory, O.: FITTEST: A new continuous and automated testing process for future internet applications. In: CSMR-WCRE, pp. 407–410. IEEE Computer Society (2014)
54. Wong, E., Yang, J., Tan, L.: Autocomment: Mining question and answer sites for automatic comment generation. In: ASE, pp. 562–567. IEEE (2013)
55. Ying, A.T.T., Robillard, M.P.: Code fragment summarization. In: ESEC/SIGSOFT FSE, pp. 655–658. ACM (2013)
56. Zampetti, F., Scalabrino, S., Oliveto, R., Canfora, G., Di Penta, M.: How open source projects use static code analysis tools in continuous integration pipelines. In: Proceedings of the 14th International Conference on Mining Software Repositories, pp. 334–344. IEEE Press (2017)