Delft University of Technology

Fixing Continuous Integration Tests From Within the IDE With Contextual Information

Boone, Casper; Brandt, Carolin; Zaidman, Andy

Citation (APA)
Boone, C., Brandt, C., & Zaidman, A. (2022). Fixing Continuous Integration Tests From Within the IDE With Contextual Information. In *Proceedings - 30th IEEE/ACM International Conference on Program Comprehension, ICPC 2022: Proceedings* (pp. 287-297). Article 9796168 (IEEE International Conference on Program Comprehension; Vol. 2022-March). IEEE. https://doi.org/10.1145/3524610.3527908

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Fixing Continuous Integration Tests From Within the IDE With Contextual Information

Casper Boone
Delft University of Technology
The Netherlands
mail@casperboone.nl

Carolin Brandt
Delft University of Technology
The Netherlands
c.e.brandt@tudelft.nl

Andy Zaidman
Delft University of Technology
The Netherlands
a.e.zaidman@tudelft.nl

ABSTRACT

The most common reason for Continuous Integration (CI) builds to break is failing tests. When a build breaks, a developer often has to scroll through hundreds to thousands of log lines to find which test is failing and why. Finding the issue is a tedious process that relies on a developer's experience and increases the cost of software testing. We investigate how presenting different kinds of contextual information about CI builds in the Integrated Development Environment (IDE) impacts the time developers take to fix a broken build. Our IntelliJ plugin TESTAXIS surfaces additional information such as a unique view of the code under test that was changed leading up to the build failure. We conduct a user experiment and show that TESTAXIS helps developers fix failing tests 13.4% to 48.6% faster. The participants found the features of TESTAXIS useful and would incorporate it in their development workflow to save time. With TESTAXIS we set an important step towards removing the need to manually inspect build logs and bringing CI build results to the IDE, ultimately saving developers time.

KEYWORDS

Software Testing, Continuous Integration, Developer Assistance, IDE Plugin, User Experiment

1 INTRODUCTION

Continuous Integration (CI) is a wide-spread practice in both industry and open-source projects [20, 32, 43]. Its goal is to detect issues as soon as possible by providing feedback before a change makes it to production. This avoids defects but also increases developer productivity [24], accelerates release frequency [18, 20], and, improves communication of changes [16].

A typical CI build comprises building the application to ensure the code compiles, executing the tests to check whether the application still works as expected, and running static analysis tools [4, 35, 36] to safeguard the quality of the codebase [17]. If any of these steps fail, the whole CI build is considered to be "broken". Failing tests are the most common reason for build failures [8, 26, 38].

When a build breaks, the developer has to find and investigate the cause of the build failure. The typical steps for a developer encountering a build failure are: inspecting the build log, developing a hypothesis about why the build is failing [41], confirming this hypothesis in their local development environment, and finally implementing a fix [37]. CI build logs typically consist of hundreds to thousands of lines and contain a lot of irrelevant information [13], which leads to developers feeling overwhelmed by the amount of detail [2], and through the verbosity it becomes hard to pinpoint bugs and their causes [40]. This makes finding the root cause of the failure a tedious and challenging process that relies on a developer's experience and intuition [19, 37] which increases the cost of software testing [33].

CI platforms offer limited inspection and debugging functionality compared to a local development environment [20]. After developing an intuition with the build log on the CI platform, the developers have to switch to the context of their integrated development environment (IDE) for further investigation. Debugging assistance that obviates the need for manual build log inspection would support developers in the build-fixing process [19].

We conjecture that developers could fix broken builds faster if we provide richer, contextual information about their CI builds directly in their IDE. One advantage is that this can combine the information available to the CI server with the local information and presentation in the Integrated Development Environment (IDE) to give powerful insights about the test failures to developers. In this paper, we investigate three different types of contextual information and measure how they influence the developer's failure-fixing behavior. In particular we look at three kinds of contextual information:

(1) **Test Outcomes and Metadata** We investigate showing in the IDE *which* CI tests failed.
(2) **Test Code** We study showing the code of the test that failed.
(3) **Changed Code Under Test** Lastly, we explore showing the production code targeted by the test that changed since the last successful test run.

These three types of contextual information lead to these analogous research questions:

> **RQ1**
>
> What is the influence of presenting the **test outcomes and metadata** on the time a developer needs to fix a failing test?

> **RQ2**
>
> What is the influence of presenting the **test code** on the time a developer needs to fix a failing test?

> **RQ3**
>
> What is the influence of presenting the **code under test, where the changed code is highlighted,** on the time a developer needs to fix a failing test?

Apart from the impact on failure fixing time, we study how useful developers rate the different kinds of contextual information with the following research question:

> **RQ4**
>
> To what extent do developers consider **contextual CI information in the IDE useful**?

We develop TESTAXIS, a plugin for the IntelliJ IDE that presents CI build and test results. During CI builds, TESTAXIS captures information about test executions and coverage. The plugin notifies the developer about the build outcome, shows all failing tests with the name of the test, the failure message, and the corresponding stack trace, obviating the need to look at the build log (RQ1). We show the test code to help the developer understand the intent of the failing test (RQ2). TESTAXIS also features an overview of the relevant code under test (RQ3) by combining information about which code was executed by a test and which code was changed leading up to the build failure. We create a proof-of-concept implementation of TESTAXIS that we use to evaluate its effectiveness and perceived usefulness. A demonstration of the plugin is available at https://youtu.be/4sfnKsvqwKw.

In order to provide an answer to our research questions, we perform a within-subjects experiment. After opening questions, we ask participants to fix eight failing tests with and without the help of TESTAXIS. The closing questionnaire asks participants about the aspects they found most useful (RQ4). Our results show that TESTAXIS helps developers fix tests failing on CI 13.4% to 48.6% faster. The participants found the features of TESTAXIS useful and would incorporate it in their development workflow to save time.

In summary, we contribute:

- An evaluation of the effect of providing CI test results with additional context (such as the test code or the code under test) in the IDE on the failure-fixing time performance.
- TESTAXIS: An IDE plugin for IntelliJ Platform IDEs that presents and visualizes build and tests results with additional context to the developer[1].
- A publicly available dataset containing the data collected during the experiments [10].

2 TESTAXIS

In this section, we lay out the design of TESTAXIS and illustrate how it presents contextual information about the CI build and its tests directly in the developer's IDE.

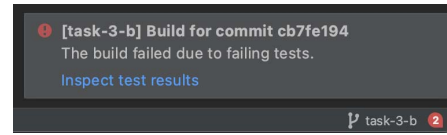[1] Available at https://github.com/testaxis/testaxis-intellij-plugin

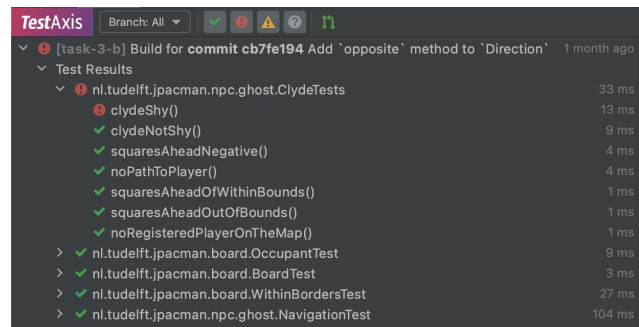**Figure 1: An example of a build notification in TESTAXIS.**

Figure 2: An example of the presentation of CI build test results in the IDE implementation of TESTAXIS.

2.1 Build Notifications

When a CI build finishes, TESTAXIS pings the developer with a notification inside their IDE, eliminating the need for a context switch to the CI platform and back. From the notification messages, the developer is immediately guided to the other features of TESTAXIS that may help solve the build failure. Figure 1 shows an example of a build notification in TESTAXIS.

2.2 Presentation of Test Results

A core feature of TESTAXIS is presenting the CI test results in a more accessible format than raw build logs, obviating the need to inspect the build log manually. Figure 2 shows an example of how TESTAXIS displays the test case executions grouped by their class, similar to the IDE's built-in test runner. This provides a familiar experience and structure to the results, making it easier to identify where in the system the failure occurs. For builds that fail due to something other than tests, TESTAXIS indicates that the build has failed due to a reason outside the scope of the tool.

2.3 Test Outcomes, Metadata and Test Code

When a developer inspects a failing test, TESTAXIS shows the test name, whether the test passed, the run time, and the execution date. As presented in Figure 3, TESTAXIS also shows the failure message and the stack trace. The stack trace includes links to the mentioned files or classes, which allows for quick navigation to the code which is not available from a CI build log.

TESTAXIS presents the source code of the test, obviating the need for manual search and navigation. Reading the test code may help developers understand the intent of the test or spot obvious mistakes quickly. Figure 4 shows an example of what this looks like.
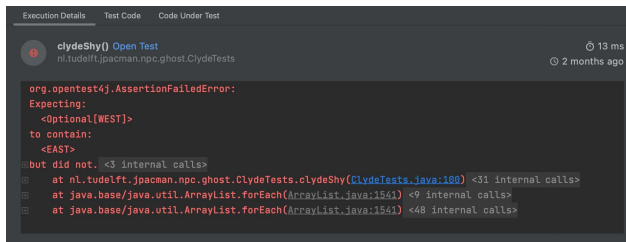
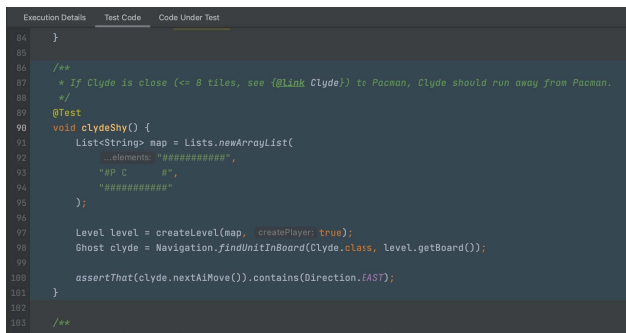Figure 3: An example of the presentation of test failure details in the IDE implementation of TestAxis.

Figure 4: An example of the presentation of test source code in the IDE implementation of TestAxis.

Figure 5: An example of how combining coverage and change information leads to more focused potential issues that require attention.

2.4 Changed Code Under Test

The goal of the changed code under test feature is to highlight the parts of the production code that are likely to contain the issue causing the test to fail. For each test, TestAxis separately collects code coverage information during the CI build, a rather cheap operation since most CI builds run the whole test suite already.

As most tests interact with multiple parts of a codebase [34], the amount of covered code could still be too large for a developer to investigate. Assuming that the test fails due to an intrinsic issue in the code [27, 28], it is likely that the issue is located in a part of the production code that was changed in the commits leading up to the build failure. Since CI builds are commonly triggered after pushing new commits, TestAxis makes use of the full change information that is available through the version control system. Figure 5 shows how TestAxis intersects code coverage and change information to identify locations of interest to the developer. See Figure 6 to see how these locations are presented within TestAxis.
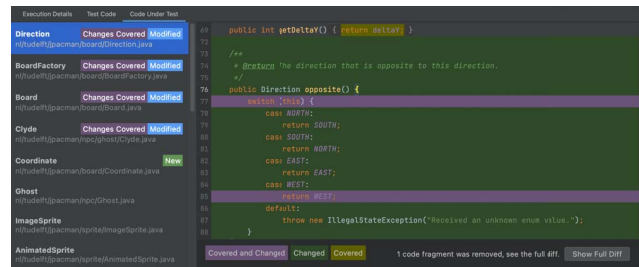

Figure 6: An example of the changed code under test feature in the IDE implementation of TestAxis.
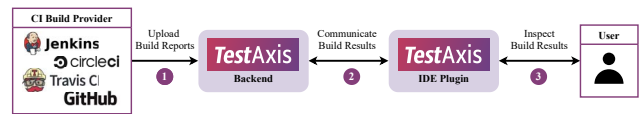
Figure 7: System Overview of TestAxis.

2.5 Implementation

We created a prototype of TestAxis which we use as part of our experiment. TestAxis consists of two main parts: the backend and the IDE plugin, see Figure 7. TestAxis receives, processes and stores the CI build results in the backend application and provides the results to the IDE plugin that presents the results to the user. A demonstration of the plugin is available at https://youtu.be/4sfnKsvqwKw.

A developer can set up TestAxis by installing the IntelliJ IDE plugin and signing in through GitHub or a TestAxis account. They configure the provided build result upload script to be run as the last step of their CI build. The upload script collects the test results and coverage results of individual tests and uploads them to the backend. By providing such a script, TestAxis does not depend on particular CI build providers. The IDE plugin presents the user with an access token that they can include in the new build step.

Both the backend and the IDE plugin are available open-source on GitHub[2].

3 STUDY DESIGN

TestAxis attempts to improve the time needed to fix broken builds by presenting contextual information about the CI build in the developer's IDE. We conduct a within-subjects experiment in which developers try out TestAxis. Our goal is to measure how the time needed to fix a failing test in a CI build is influenced by presenting different kinds of contextual information in the IDE: the test result RQ1, the test code RQ2, and the changed code under test RQ3. Furthermore, we elicit whether developers consider TestAxis useful RQ4.

3.1 Experiment Overview

Before the participants start the assignments, we ask them about their demographics and show two instruction videos. One presents the architecture and structure of the codebase of our example project JPacman and the other one explains the functionality of

---

[2]Available at https://github.com/testaxis

TestAxis. During the experiment, the participants solve four assignments *with* and four assignments *without* TestAxis. We design assignments in four different categories that target the kinds of contextual information we investigate in our study. Per category, we divide the participants into two groups so that each participant conducts one assignment per category *without* and one assignment *with* TestAxis. For each assignment, we present a CI build that failed due to failing tests. The participants have to find out which tests fail and why. Then, they have to come up with a fix. The researcher measures the time in seconds until the participant successfully fix the failing test. After each assignment, they filled out a post-assignment questionnaire where we asked about what they felt they spent the most time on. After the experiment, the participants filled out a questionnaire asking about the usefulness of TestAxis and its features.

3.2 Example Project

The assignments of the experiment ask participants to fix failing test cases that attempt to mimic test failures that occur while working on real software projects. The simulation of realistic test failures requires that the designed test cases are part of a software project that is sufficiently complex and close to real-life projects. We picked JPacman, a simple Pacman-style game implemented in Java that is written for software testing education. The codebase of JPacman is small enough to be easy to understand within a short time. However, it does not have a trivial implementation to ensure that developers do not deviate from their usual behavior. The project has a variety of unit tests, integration tests, and system/end-to-end tests that are of a quality level comparable to an industry project. The project is available open-source on GitHub[3].

3.3 Assignment Design

We designed eight assignments in four categories. The categories evaluate different aspects of the feature set of TestAxis. To keep the maximum length of each experiment session reasonable, the first three categories have a time limit of five minutes, while the last has a time limit of ten minutes. The participants execute all eight assignments, for each category one with and one without TestAxis. In the overall experiment, each assignment is executed eight times with and eight times without TestAxis.

Category 1: Test Outcomes and Metadata. For the assignments in this category, the reason for the failure can be spotted from the test failure metadata (the name of the test and the stack trace). Figure 3 shows how TestAxis presents this information. Figure 8 illustrates how one assignment from this category is presented in both GitHub and TestAxis.

Category 2: Test Code. For the assignments in this category, the reason for the failure can be spotted in the test code (Figure 4).

Category 3: Code Under Test – Simple. For the assignments in this category, the reason for the failure can be spotted in the code under test (Figure 6). In Figure 9 we present how an assignment from this category is presented in both GitHub and TestAxis.

Category 4: Code Under Test – Advanced. For the assignments in this category, the reason for the failure can also be spotted in the code under test. However, these assignments are more advanced.

To mitigate learning or order effects, we randomize the order in which the participants solve the assignments. For each of the eight assignments, we compare the performance of both groups (with and without TestAxis). To prevent any effects of the randomized group selection, we use a crossover design: For each category, we only determine an overall trend if both assignments show the same trend when comparing the times with and without TestAxis. The complete assignments are available in our replication package [10].

3.4 Participants

To conduct the experiment and gain useful insights about the results, we needed to recruit a large enough number of participants. We required participants to have experience with Java and CI. This ensures a somewhat equal baseline and the ability for participants to reflect on their CI build-fixing workflows. At the same time, we wanted a diverse group of participants and therefore used a phased participant recruitment process with different target audiences per step. We first reached out to acquaintances, which are mostly (PhD) students. Then, we placed a message on the internal messaging platform of Computer Science teaching assistants of our institution, with a similar target audience. To target industry developers, we posted a collection of tweets on Twitter illustrating the capabilities of TestAxis and asking for a developer's help to improve the project. Finally, we also posted on LinkedIn with the same target audience in mind. To thank the participants for their time and to increase engagement, we raffled four 15 euro gift cards among the participants.

In total, 16 participants signed up for the experiment. The most experienced participants have programmed for 12 years. The other participants are relatively equally distributed between 4 and 12 years. 31.3% of the participants work in industry as a software engineer, while the main occupation of the remaining participants is student or PhD student. All participants have an academic background. The current or highest education level of most participants is MSc (56.3%). The other participants were either BSc (25%) or PhD students (18.8%) at the time of the experiment.

The 16 participants consider themselves to be experienced software developers (● 4.0 ▁▁▅[4]). They are experienced with developing Java applications (● 3.8 ▁▅) in IntelliJ (● 3.9 ▁▅). Some of the participants have experience developing software applications professionally (● 3.4 ▄▅), whereas others do not have any experience in this area. The participants are less experienced with software testing (● 3.6 ▁▅). We observe that some of the participants indicated to be very experienced in testing, while others indicated to not be experienced at all. The results on the experience with CI show a different trend. The participants rate themselves as highly experienced in using CI build tools (like Travis CI, GitHub Actions, or Jenkins; ● 4.2 ▄▅) and inspecting the output logs when a build fails (● 3.9 ▁▅).

---

[3]Our JPacman fork is available at https://github.com/testaxis/jpacman

[4]To present the Likert-scale results we show the average score indicated by the purple-colored dot. The small bar chart gives a rough indication of the distribution of the answers that the participants gave.
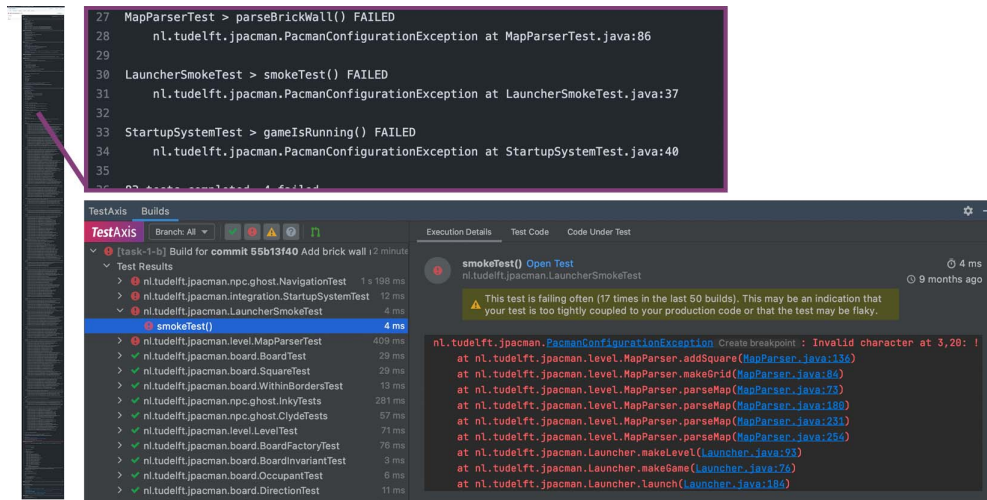
Figure 8: Illustration of Assignment 1b, where the issue to be found can be spotted from the stack trace of the failing test: The invalid character "!" in the map definition. On the left, the long stacktrace on GitHub that points to the failing test, on the bottom the failing tests and error messages as TestAxis presents them.
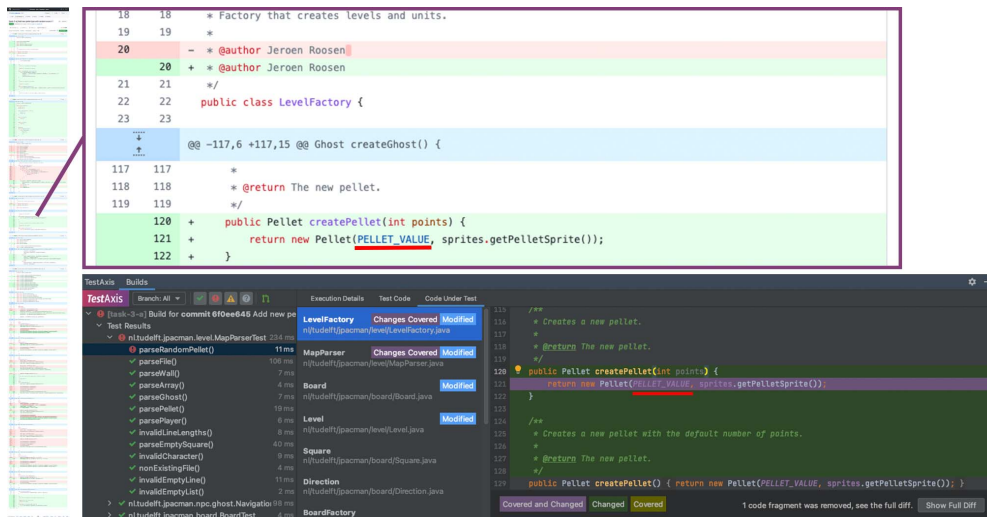
Figure 9: Illustration of Assignment 3a, where the issue to be found is in the code under test. On the left, the long unfiltered list of changes as presented in GitHub, on the bottom the prioritized list of covered changes as presented in TestAxis.

A minority of the participants have some previous experience with the software project (● 2.4 ▄▆▁).

3.5 Experiment Execution

The experiment is approved by the Human Research Ethics Committee of our university and follows the guidelines set by the committee. At the start of the experiment, participants read and sign an informed consent form indicating that they understand what data will be collected and how it will be used. Before we conducted the experiment, we first ran a pilot to evaluate the design of the experiment and improved several aspects of our design and the tools

we used. We conducted the experiment in March 2021 for three weeks in multiple sessions per day. Due to the COVID-19 pandemic, the experiment was fully remote. All sessions were individual and guided by an observer. During a session, the observer took notes of interesting things that happened or were said during the experiment and timed the assignments. A session took about 90 minutes, depending on the time needed to fill out the questionnaires or solve the assignments.
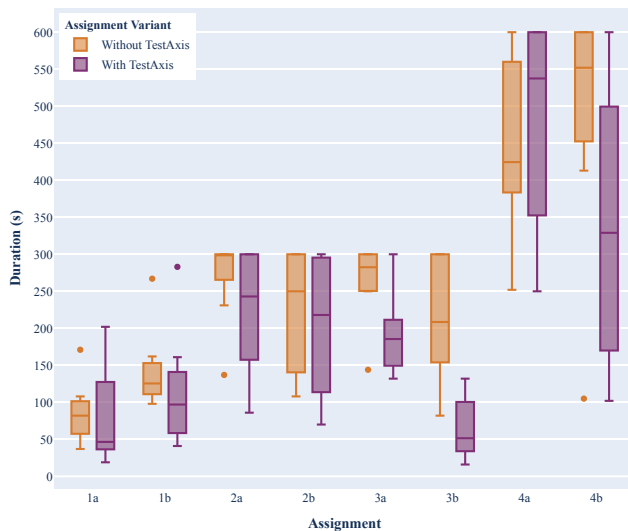
Figure 10: The failure-fixing time in seconds of both the *without* and *with* variant of all assignments. The first six assignments have a time limit of 5 minutes. For the last two assignments, the limit is 10 minutes.

4 RESULTS

In this section, we present the results of our within-subjects experiment.

4.1 Time to fix failing tests

Each of the 16 participants conducted one with/without TESTAXIS variant of all eight assignments. Each variant was thus solved by 8 participants. The observer measured the time between starting an assignment variant and fixing the issue. Figure 10 shows the results per assignment per variant. In all cases except 4a, we see that the median time to fix the issue is lower for the *with* variant than the *without* variant. For the assignments in the second and fourth category, we observe a high variability in the results. For category one, where the issue can be spot in the test metadata (Figure 3), we see an overall improvement of 13.4%. For category two, with the issue in the test code (Figure 4), this is 13.8%. The assignments in category three (simple issue in code under test, Figure 6) show the greatest performance improvement, on average 48.6%. Although the first assignment of category four (advanced issue in code under test) shows a performance decrease, on average, the performance difference of category four is 12.1%. Because of our crossover design, we cannot determine that there is an improvement in the failure-fixing time for the assignments of category four. Overall, the four assignment variants *with* TESTAXIS are conducted 22.0% faster.

The participants did not manage to solve all assignments within the time limit. When a participant hit the limit, we consider their failure-fixing time to be the maximum time of 5 minutes for categories one-three and 10 minutes for category four. We observe a high number of hit time limits for category two. In general, we see a lower number of hit time limits for the *with* TESTAXIS assignment variants, except for assignment 4a.

After each assignment, we conducted the post-assignment evaluation questionnaire. Overall, an interesting result is a decrease in the average score of having to run the test locally to get more information from 3.4 to 1.6. Also, the perceived time spent on finding out *which* test(s) failed dropped from 2.1 to 1.2, on average.

4.2 Usefulness of The Tool

After our experiments, we asked the participants how useful they found the different informational elements of TESTAXIS. The majority of the participants find that the information provided by TESTAXIS in the various features helps them understand a failure better and fix it more quickly (● 4.4 ▁▁▋). The participants consider the details tab containing meta-information such as the test name and the interactive stack trace (shown in Figure 3), as well as the changed code under test tab (as shown in Figure 6) most useful (● 4.2 ▁▁▋ and ● 4.4 ▁▁▋, respectively). Participant 5 even indicates that they *"already like just having the overview of failed tests a lot over a build log where I can see the name of the [failing] test but not much more"*. However, the participants rate the usefulness of the test code feature slightly lower (● 3.9 ▁▋▋). Participant 8 mentioned that the test code tab may be unnecessary since there is already an "Open Test" button that opens the test in the main window of the IDE. The participants signal the value of highlighting changes in the code under test tab and consider it a very important part of the code under test feature (● 4.6 ▁▁▋). Participant 13 explained why they think this feature is relevant: *"The highlights of the code under test are very important since that's what you would normally do manually by thinking about what changed and what the test could have covered. And this shows you everything automatically without any margin for error."*

5 ANALYSIS AND DISCUSSION

The goal of our investigation is to gauge whether providing context about test failures in the local IDE helps developers fix broken CI builds faster. We implemented TESTAXIS and conducted an experiment where we asked developers to solve test failures that require different kinds of contextual information. In this section, we analyze the results of our experiment, such as whether the performance improvements are statistically significant, discuss the implications of our results and propose answers to our research questions.

5.1 What is the influence of presenting the test outcomes and metadata on the time a developer needs to fix a failing test?

We designed the assignments of the first category to be solvable with only the meta-information (the name and stack trace of the failing test) presented in TESTAXIS (see Figure 3). The first category contains simple test failures where the issue can be spotted in the stack trace alone. We found that the average time participants needed to solve the two assignments decreased by 13.4% when using TESTAXIS. Also, the participants indicated they spent considerably less time figuring out which tests failed with the help of TESTAXIS. Moreover, TESTAXIS reduced the need to run the failing tests locally to get more feedback. We thus see indications that presenting a test failure in the IDE over a CI build log has a positive influence on the failure-fixing time.

RQ1

What is the influence of presenting the **test outcomes and metadata** on the time a developer needs to fix a failing test?

- -

- Developers solve test failures faster when the failure information is presented in the IDE over a CI build log. In the experiment, we saw an average performance increase of 13.4%.
- Developers indicate they need less time to find which test is failing using TESTAXIS.

5.2 What is the influence of presenting the test code on the time a developer needs to fix a failing test?

The second assignment category featured test failures due to issues in the test code. Figure 4 demonstrates how TESTAXIS presents the test code. Many participants hit the time limit in this category, 10 participants for the assignments *without* TESTAXIS and 5 for the assignments *with*. This could be caused by the participants focusing more on the code under test than on the test code, which they indicated in the post-assignment survey. The results show an average improvement of the failure-fixing time of 13.8%. We see clear indications that TESTAXIS helps to solve assignments where the issue is in the test code more quickly.

RQ2

What is the influence of presenting the **test code** on the time a developer needs to fix a failing test?

- -

- Developers solve test failures more quickly when they have quick access to the test code as part of the failure information. In the experiment, we saw an average performance increase of 13.8%.

5.3 What is the influence of presenting the code under test, where the changed code is highlighted, on the time a developer needs to fix a failing test?

We evaluate the effect of showing changed code under test (RQ3, Figure 6) in both the third and fourth category. In the third category, we observe an average improvement in the failure-fixing time of 48.6%. The test cases in this category are straightforward, and the issues are in one of the few highlighted code fragments of the changed and covered code shown by TESTAXIS. The suggestions of potential locations of the issue cut down the number of lines of code to inspect drastically compared to inspecting the full code change diff, which is likely the explanation for the significant increase in failure-fixing performance. The participants have also indicated that they spent the most time on the code under test while figuring out the cause of the failure.

The fourth category consists of two assignments where the same high-level end-to-end test fails, and the participants must find out why. These assignments are more complex than the third category and require a deeper investigation by the developer. Even though we saw an average performance improvement of 12.1%, the results for the two assignments in this category show different trends.

Assignment 4b showed an improvement in performance while assignment 4a showed a decrease in performance. By the design of the study, we can thus not conclude anything about the results for this category. We found that the more experienced developers that perform the assignment *with* TESTAXIS need more time to solve the assignment than the less experienced developers, contrary to the other assignments. A possible explanation could be that the less experienced participants find it easier to adopt new features, such as the changed code under test feature, into their workflow, whereas for more experienced developers it may be difficult to fit a new type of feature in their existing tool belt.

We see clear indications that showing the changed code under test positively influences the failure-fixing time for simple cases. Our results are inconclusive about more complex cases.

RQ3

What is the influence of presenting the **code under test, where the changed code is highlighted,** on the time a developer needs to fix a failing test?

- -

- Developers solve simple test failures more quickly when they have an overview of the changed code under test. In the experiment, we saw an average performance increase of 48.6%. This improvement is statistically significant.
- The results cannot tell us whether there is a performance impact when using the changed code under test failure for more complicated tests, such as end-to-end tests. In the experiment, we saw an average performance increase of 12.1% but cannot rule out the effect of the participant distribution per category.
- More experienced developers are less efficient than less experienced developers when using the code under test feature.

5.4 Statistical Significance of the Measured Performance Improvements

In Section 4.1, we present the performance results of all assignment variants. For all assignments except one, we observe an improvement of the average failure-fixing time when using TESTAXIS. We use the two-tailed Mann-Whitney U test [22] to analyze whether these improvements are statistically significant. We reject our null hypothesis "there is no difference between performing an assignment *without* or *with* TESTAXIS" when $p\text{-}value \leq 0.05$.

Table 1 shows the U values per assignment. It also shows p-values using a normal approximation. For assignment 3a and 3b we can reject our null hypothesis and conclude that the improvements are statistically significant. For all other assignments, we cannot conclude that the performance improvements are statistically significant.

Table 1: Statistical significance of the observed performance improvements.

	U	p	Reject H_0		U	p	Reject H_0
1a	24.0	0.215		3a	12.5	0.022	✓
1b	19.0	0.095		3b	4.0	0.002	✓
2a	20.0	0.092		4a	28.0	0.355	
2b	22.0	0.255		4b	17.0	0.059	

5.5 Impact of Running Tests Locally and Determining Which Test Failed

In all assignments with TestAxis, the participants rarely had to run a failing test locally in the IDE to get more information. Also, the time needed to find out *which tests* failed dropped significantly in the assignments *with* TestAxis compared to the ones *without*. These two general findings contributed in almost all assignments to an improvement of the failure-fixing time when using TestAxis.

> **Insight**
>
> Developers using TestAxis almost never run tests locally when to gain more details.

5.6 To what extent do developers consider contextual CI information in the IDE useful?

After the assignments, we asked the participants about the usefulness of TestAxis and the different features they used. The participants consider all three kinds of contextual information useful, with the changed code under test feature being the most useful. The participants think that TestAxis solves a real problem and that it would save them time. Most participants would make TestAxis part of their workflow, one of the participants indicated that they would consider implementing it in their workflow *"as is"* and that they would *"not add much info further as I think its strength lies in the clean and concise overview"*. Overall, the participants find TestAxis useful in helping them understand test failures better and fix them more quickly. One of the participants described their experience as *"I thought it was super useful during the experiments. I much rather preferred using TestAxis over the traditional CI logs on GitHub. What TestAxis does, in my opinion, is recreate the steps I manually take on a GitHub pull request to identify a failing test, and it does so in the IDE so I don't have to switch tabs and interrupt my workflow."*

> **RQ4**
>
> To what extent do developers consider **contextual CI information in the IDE useful**?
>
> -
>
> - The participants find TestAxis useful in helping them understand a test failure better and fix it more quickly.
> - The participants consider all three main features of TestAxis (failure details, test code, and code under test) to be useful. The (changed) code under test feature is considered to be most useful.
> - The participants believe that TestAxis solves a real problem.
> - The usage of TestAxis would save the participants time and they would make it part of their workflow.
> - The participants strongly agree that TestAxis provides benefits over inspecting CI build logs manually.

6 THREATS TO VALIDITY

To support the credibility of our results, we outline the threats to the validity of our user experiment.

6.1 Internal Validity

Internal validity indicates the reliability of the cause-and-effect relationship between the introduction of TestAxis and the observed effects in the results.

While analyzing our results, we observed a learning effect. Our participants were able to solve assignments quicker at the end of the experiment. We expected this while designing the experiment and mitigated the impact by randomizing the order of the assignments. We base the results for each assignment on both early and late executions in the experiment. Half the participants got the *without* TestAxis variant of a category first, and the other half got the *with* variant of the category first.

Creating two assignments that are similar enough to directly compare is very complex (see Section 3.3). To mitigate this, we compare the results of the *without* variant of a specific assignment against the *with* variant, executed by another group. As then the group composition could influence the results, we only consider there to be an effect of using TestAxis when the results show the same trend for both assignments of a category.

Another threat is whether the participants felt comfortable giving their honest opinions. The participants knew that their activity was observed while conducting the assignments and filling out the questionnaires. This could cause a Hawthorne effect, participants answering questions more positively [1]. While it is not possible to show that this was not the case, we do observe negative answers to some of the questions. This suggests that the participants felt at ease and comfortable sharing their opinions.

In the experiment, TestAxis is used for a short time. A longer study of teams working with the tool in real projects is needed to measure its true impact. The short time is not long enough to incorporate a new feature such as the changed code under test feature into one's workflow. Participant 14 confirmed this by saying *"For the best experience, requires a user to learn the intuitions of the tool"*.

6.2 Construct Validity

Construct validity is concerned with the degree to which a test actually measures the construct(s) it claims to be testing. The performance results are based on quantitative data, the duration of assignment executions. The timing results may be influenced by different behavior induced by the experiment environment. However, the participants agree with the statement that they used the same tactics during the assignments *without* TestAxis as they would have done outside the experiment. In a follow-up study, a more objective approach that monitors IDE usage, such as WatchDog [5], could be considered to get a better indication of which tasks are most influenced by the usage of TestAxis.

6.3 External Validity

The external validity is concerned with the generalizability of the results. As shown in Section 3.4, the participants are a diverse group with mixed backgrounds. However, the group size is relatively small, which may cause individual differences in their background to have a greater effect on the results than in a larger group of participants.

How similar the test failures in the assignments are to real test failures is an important factor for the generalizability of our results.

While the example project JPacman is smaller than most applications, it does feature an extensive test suite, modern build pipeline, and design practices such as dependency injection. The short duration of the experiment requires a project that can be understood quickly. The participants agree that JPacman allowed for interesting cases that were suitable to answer the questions. The cases we designed mimic test failures that could happen in any type of software project. The participants neither agree nor disagree that the assignments are similar to the ones they encounter in their own projects, indicating that the generalizability of the assignments is a threat to the validity of the results.

7 RELATED WORK

CI builds and test failures are well-explored topics. This section discusses a selection of the research done in these areas that is related to our work.

7.1 Assistance in Fixing Failing Tests

Beller et al. monitor the behavior of developers after observing a test failure [6]. Their results show that in more than 60% of the cases, a developer starts reading the code under test. Another 17% reads the test code first. However, after 5 seconds, a significant number of users switch focus from the IDE to another window. A possible explanation is that developers reach out to external resources to help solve the issue.

The need for such external resources could be fulfilled by providing more context around the test failure. Zhang et al. proposed an approach that explains the reasons for test failures through comments in the test code [42]. For example, it adds comments indicating which exception is thrown by a specific line. It also suggests fixes by mutating the failing tests to see if it can find a variant that would pass. Using a statistical algorithm, they determine and comment the production code most suspicious of causing the failure.

ReAssert also mutates test code to try to make the test pass [15]. The tool suggests mutations that result in a passing test as repair options. It can, for example, replace literals and change assertions. This method only works if the test code is no longer in line with the production code. If the failure is caused by a regression, mutating the test code would capture the wrong behavior of the production code.

7.2 CI Build Results in the IDE

There exist several IDE extensions that show the status of CI builds, sometimes with additional information. The plugins have different characteristics. Table 2 shows all IDE plugins displaying CI builds that we identified in the JetBrains and Eclipse Marketplace. Three of the plugins notify the developers of build status updates [9, 30, 31]. Half of the plugins only show raw information of the builds (like the status or the logs) [3, 23, 31], while the others also interpret the builds and show the test results [9, 30, 39].

Moreover, the TeamCity [30] plugin and the Hudson/Jenkins Mylyn Builds Connector [39] also provide additional insights. Table 3 shows a comparison between the test insights features of these two plugins and TestAxis. TeamCity displays which tests failed and highlights stack traces. It also offers the ability to easily rerun a test locally. The Hudson/Jenkins Mylyn Builds Connector plugin shows the test results but also provides insights on execution times and code changes made for this build.

TestAxis also gives these insights. It shows an interactive stack trace together with execution details such as the run time. While Hudson/Jenkins Mylyn Builds Connector only shows a list of changes, TestAxis incorporates these changes in the changed code under test feature that both shows which code fragments were changed and touched by the test. Furthermore, TestAxis also provides easy access to the test code to understand the intent of the test or to spot mistakes in the test itself. TestAxis is not limited to a specific CI service and can be included in the build process of any CI tool.

One might expect that showing CI test results in the IDE is not needed because developers can just execute the tests in their IDE that shows a good interface to review and inspect failing tests. However, it turns out that developers actually do not often execute tests in their IDE [6, 7]. Beller et al. also mention that *"Despite the tool overhead and a possibly slower reaction time, our low results on test executions in the IDE suggest that developers increasingly prefer such more complex setups [in which tests are run on CI servers] to manually executing their tests in the IDE."* and continue by recommending that IDE developers should improve CI integration [6].

7.3 The Augmented IDE

Our work also lies in the context of integrating additional sources of information within the IDE. A key goal here is to improve software understanding and development, and also reducing the context switches between tools that software engineers typically use. A notable examples is the work of Holmes and Begel: Deep Intellisense [21], an IDE plugin that links bug reports, emails, code changes to source code entities. Furthermore, the Eclipse plugin Hipikat [14], tries to assist newcomers by recommending problem reports, newsgroup articles, etc. related to the task at hand.

Other tools tackle the challenge of reducing the context switching from IDE to web browser. For example, Ponzanelli et al. bring Stack Overflow into the IDE [25]. Another Eclipse plugin, Fishtail [29] harnesses programmer's interactions history to bring relevant web resources into the IDE. Similarly, TestKnight is a plug-in for IntelliJ that helps developers engineer developer tests by (1) providing suggestions on which parts should still be tested, offering boilerplate test code solutions, and (3) adding support for copying and pasting test cases with suggestions on which parts to change [12].

8 CONCLUSION AND FUTURE WORK

Inspecting the results of a failing test in a CI build is a tedious process. It often requires developers to manually inspect and scroll through hundreds to thousands of lines of log output, while running tests inside an IDE offers specific, detailed, and interactive feedback on the test results. TestAxis brings CI test results to the IDE and offers a similar experience to running a test locally. Moreover, it exploits the change and coverage information available on the CI to offer additional support while inspecting test failures. In this paper, we explored how three different kinds of contextual information reduce the time needed to fix a test failing on CI.

Table 2: IDE plugins that show CI build statuses and/or results.

Name	IDE	Users	CI Service	Build Status	Build Logs	Notifi-cations	Test Results	Test Insights
TeamCity [30]	IntelliJ	778,6K	TeamCity	✓	✓	✓	✓	✓
Jenkins Control Plugin [9]	IntelliJ	204,7K	Jenkins	✓	✓	✓	✓	
IntelliJ GitLab Pipeline Viewer [31]	IntelliJ	7,2K	GitLab	✓		✓		
Github Tools [23]	IntelliJ	6,3K	Travis CI / CircleCI	✓				
GitHub Actions [3]	IntelliJ	3,1K	GitHub Actions	✓	✓			
Hudson/Jenkins Mylyn Builds Connector [39]	Eclipse	Not reported	Jenkins	✓	✓		✓	✓
TestAxis	IntelliJ	-	All	✓	✓	✓	✓	✓

Table 3: Comparison of IDE plugins with test insights.

Feature	TeamCity [30]	Hudson/Jenkins Mylyn Builds Connector [39]	TestAxis
Interactive Stack Traces	✓	Indirectly through JUnit view	✓
Display of Test Code	Link only	Link only	✓
Code Under Test			✓
Code Changes	✓	✓	✓
Changed Code Under test			✓
Raw Build Log Inspection	✓	✓	
Rerun Test	✓	Indirectly through JUnit view	Indirectly by opening test code in main window
Supported CI Providers	TeamCity	Jenkins	All

Our results show that it is helpful to present the test results and the test code in the environment of the developer. For simple failures, we saw a statistically significant improvement in failure fixing time when presenting the code under test which was changed since the last successful build. For complex failures, our results were diverging, possibly because more experienced developers took more time to incorporate the new feature into their workflow, or because the assignments were too difficult for our participants in the scope of our experiment. Overall, the developers judged TestAxis as a useful tool which they would integrate into their workflow and emphasized the power of the unique presentation of changed code under test. A central advantage of TestAxis is the combination of information available on the CI and the familiar, local presentation in the IDE. This enables us to give developers powerful insights into their test failures.

While our study has shown that TestAxis can have a positive effect on the type of cases we presented, more work is needed to confirm its usefulness and performance improvement in real-life projects. Due to the design of our study, we could only see indications that there is *an effect* after introducing TestAxis but not that this effect is *caused by* TestAxis. Our experiment should be repeated as a controlled experiment with a larger sample size or as a longitudinal study on a real software project to achieve stronger conclusions. Further, TestAxis could be extended to provide additional context for builds that fail for other reasons than tests, such as dependency errors or static analysis warnings. A combination with links to the code or automatic fixing suggestions could also help leverage the unique combination of CI and IDE in these cases.

ACKNOWLEDGMENTS

REFERENCES
[1] J. G. Adair. 1984. The Hawthorne Effect: A Reconsideration of the Methodological Artifact. *Journal of Applied Psychology* (1984), 69(2):334. https://doi.org/10.1037/0021-9010.69.2.334
[2] Anunay Amar and Peter C. Rigby. 2019. Mining historical test logs to predict bugs and localize faults in the test logs. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Montreal, Quebec, Canada, 140–151. https://doi.org/10.1109/ICSE.2019.00031
[3] Andrey Artyukhov. 2020. GitHub Actions. https://plugins.jetbrains.com/plugin/13793-github-actions
[4] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. 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)*. IEEE Computer Society, 470–481. https://doi.org/10.1109/SANER.2016.105
[5] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2019. Developer Testing in the IDE: Patterns, Beliefs, and Behavior. *IEEE Transactions on Software Engineering* 45, 3 (March 2019), 261–284. http://doi.org/10.1109/TSE.2017.2776152
[6] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 179–190. http://doi.org/10.1145/2786805.2786843
[7] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2015. How (Much) Do Developers Test?. In *37th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 559–562. https://doi.org/10.1109/ICSE.2015.193
[8] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 356–367. https://doi.org/10.1109/MSR.2017.62
[9] David Boissier, Yuri Novitsky, and Michael Suhr. 2011. Jenkins Control Plugin. https://plugins.jetbrains.com/plugin/6110-jenkins-control-plugin
[10] Casper Boone. 2021. TestAxis Replication Package. *Zenodo* (Sept. 2021). https://zenodo.org/record/5526015
[11] Casper Boone. 2021. *TestAxis: Save Time Fixing Broken CI Builds Without Leaving Your IDE*. Master's thesis. Delft University of Technology. http://resolver.tudelft.nl/uuid:f8375d5f-3bbd-4559-863b-6951e9d6bab0
[12] Cristian-Alexandru Botocan, Piyush Deshmukh, Pavlos Makridis, Jorge Romeu Huidobro, Mathanrajan Sundarrajan, Mauricio Aniche, and Andy Zaidman. 2022. TestKnight: An Interactive Assistant to Stimulate Test Engineering. In *Proceedings of the 44th International Conference on Software Engineering (ICSE Companion)*. ACM. To appear.
[13] Carolin E. Brandt, Annibale Panichella, Andy Zaidman, and Moritz Beller. 2020. LogChunks: A Data Set for Build Log Analysis. In *MSR '20: 17th International Conference on Mining Software Repositories (MSR)*. ACM, 583–587. https://doi.org/10.1145/3379597.3387485

[14] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. 2004. Learning from project history: a case study for software development. In *Proceedings of the 19th Conference on Computer Supported Cooperative Work (CSCW)*. 82–-91.

[15] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. 2009. ReAssert: Suggesting Repairs for Broken Unit Tests. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 433–444. https://doi.org/10.1109/ASE.2009.17 ISSN: 1938-4300.

[16] John Downs, Beryl Plimmer, and John G. Hosking. 2012. Ambient awareness of build status in collocated software teams. In *2012 34th International Conference on Software Engineering (ICSE)*. 507–517. https://doi.org/10.1109/ICSE.2012.6227165 ISSN: 1558-1225.

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

[18] D. Goodman and M. Elbaz. 2008. "It's Not the Pants, it's the People in the Pants" Learnings from the Gap Agile Transformation What Worked, How We Did it, and What Still Puzzles Us. In *Agile 2008 Conference*. 112–115. https://doi.org/10.1109/Agile.2008.87

[19] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 197–207. http://doi.org/10.1145/3106237.3106270

[20] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 426–437. http://doi.org/10.1145/2970276.2970358

[21] Reid Holmes and Andy Begel. 2008. Deep Intellisense: a tool for rehydrating evaporated information. In *Proceedings of the International working conference on Mining software repositories (MSR)*. ACM, 23–-26.

[22] Henry B. Mann and Donald R. Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60. ISBN: 0003-4851 Publisher: JSTOR.

[23] Diego Marcher. 2019. Github Tools. https://plugins.jetbrains.com/plugin/13366-github-tools

[24] Ade Miller. 2008. A Hundred Days of Continuous Integration. In *Agile 2008 Conference*. 289–293. https://doi.org/10.1109/Agile.2008.8

[25] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. 2013. Seahawk: stack overflow in the IDE. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 1295–1298.

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

[27] Gema Rodríguez-Pérez, Andy Zaidman, Alexander Serebrenik, Gregorio Robles, and Jesús M. González-Barahona. 2018. What if a bug has a different origin?: making sense of bugs without an explicit bug introducing change. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 52:1–52:4. https://doi.org/10.1145/3239235.3267436

[28] Gema Rodríguez-Pérez, Gregorio Robles, Alexander Serebrenik, Andy Zaidman, Daniel M. Germán, and Jesus M. Gonzalez-Barahona. 2020. How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering* 25, 2 (March 2020), 1294–1340. https://doi.org/10.1007/s10664-019-09781-y

[29] Nicholas Sawadsky and Gail C Murphy. 2011. Fishtail: from task context to source code examples. In *Proceedings of the 1st Workshop on Developing Tools as Plug-ins*. 48–51.

[30] JetBrains s.r.o. 2007. TeamCity IntelliJ Plugin. https://plugins.jetbrains.com/plugin/1820-teamcity

[31] Simon Stratmann. 2020. IntelliJ GitLab Pipeline Viewer. https://plugins.jetbrains.com/plugin/13799-intellij-gitlab-pipeline-viewer

[32] Daniel Ståhl and Jan Bosch. 2014. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software* 87 (Jan. 2014), 48–59. https://doi.org/10.1016/j.jss.2013.08.032

[33] G. Tassey. 2002. *The economic impacts of inadequate infrastructure for software testing*. National Institute of Standards and Technology.

[34] Fabian Trautsch, Steffen Herbold, and Jens Grabowski. 2020. Are unit and integration test definitions still valid for modern Java projects? An empirical study on open-source projects. *Journal of Systems and Software* 159 (Jan. 2020), 110421. https://doi.org/10.1016/j.jss.2019.110421

[35] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empir. Softw. Eng.* 25, 2 (2020), 1419–1457. https://doi.org/10.1007/s10664-019-09750-5

[36] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C. Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, 38–49. https://doi.org/10.1109/SANER.2018.8330195

[37] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. 2020. Every build you break: developer-oriented assistance for build failure resolution. *Empirical Software Engineering* 25, 3 (May 2020), 2218–2257. https://doi.org/10.1007/s10664-019-09765-y

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

[39] Paul Verest. 2013. Hudson/Jenkins Mylyn Builds Connector. https://marketplace.eclipse.org/content/hudsonjenkins-mylyn-builds-connector

[40] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. 2019. A Conceptual Replication of Continuous Integration Pain Points in the Context of Travis CI. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 647–658.

[41] Andreas Zeller. 2005. *Why Programs Fail: A Guide to Systematic Debugging*. Morgen Kaufmann.

[42] Sai Zhang, Cheng Zhang, and Michael D. Ernst. 2011. Automated documentation inference to explain failed tests. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 63–72. https://doi.org/10.1109/ASE.2011.6100145 ISSN: 1938-4300.

[43] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The impact of continuous integration on other software development practices: A large-scale empirical study. In *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 60–71. https://doi.org/10.1109/ASE.2017.8115619