In industrial embedded software, are some compilation errors easier to localize and fix than others?

Han Fu*†, Sigrid Eldh*‡, Kristian Wiklund*, Andreas Ermedahl*†, Philipp Haller† and Cyrille Artho†
*Ericsson AB, Stockholm, Sweden
Email: {han.fu, sigrid.eldh, kristian.wiklund, andreas.ermedahl}@ericsson.com
† KTH Royal Institute of Technology, Stockholm, Sweden
Email: {phaller, artho}@kth.se
‡ Mälardalen University, Västerås, Sweden

***Abstract*—Industrial embedded systems often require specialized hardware. However, software engineers have access to such domain-specific hardware only at the continuous integration (CI) stage and have to use simulated hardware otherwise. This results in a higher proportion of compilation errors at the CI stage than in other types of systems, warranting a deeper study.**

To this end, we create a CI diagnostics solution called "Shadow Job" that analyzes our industrial CI system. We collected over 40000 builds from 4 projects from the product source code and categorized the compilation errors into 14 error types, showing that the five most common ones comprise 89 % of all compilation errors. Additionally, we analyze the resolution time, size, and distance for each error type, to see if different types of compilation errors are easier to localize or repair than others.

Our results show that the resolution time, size, and distance are independent of each other. Our research also provides insights into the human effort required to fix the most common industrial compilation errors. We also identify the most promising directions for future research on fault localization.

***Index Terms*—continuous integration, software build, compilation error, fault localization**

I. Introduction

Agile development [1] relies on continuous integration (CI) [2]. In large industries, many agile teams submit code changes concurrently. A previous paper [3] shows that dependency issues in hardware-in-the-loop testing result in a significant portion of build failures. This is because the hardware platform is developed alongside the software system. Toward the end of the development cycle, hardware prototypes are made available to the CI test bed. Software developers can then integrate their changes with the new platform. The tests of that integration can only be done in full on the CI platform, as it would be prohibitively expensive to procure a hardware prototype for each developer.

Therefore, industrial hardware-software co-development fundamentally differs from typical open-source projects often studied in the literature. Compilation errors occur very frequently at this integration stage and are responsible for most of the CI failures [3].

Our goal is to investigate opportunities to enhance fault localization and program repair by addressing these problems in this unique setting and streamlining the process. To locate a faulty module, analyzing and repairing the error can account for a significant part of the development effort, leading to lower productivity, higher costs, and a potentially longer time to market [4], [5]. Hence, automation of fault localization can be a key differentiating ability for any business or activity [6].

In this paper, we set out to investigate the main research question, which has five sub-research questions: **RQ.** *What are the key factors contributing to compilation errors in the context of industrial embedded system CI?*

We studied the localization and resolution of compilation errors in industrial embedded systems to identify the potential of automatic remedies for typical errors.

Our proposed CI diagnostics solution, called "Shadow Job", is designed to address our research questions. We analyze over 40000 builds and identify 14 compilation error types representing 98 % of the failed compilations. Amongst all errors, five most common ones comprise 89 % of all compilation errors. The main reason for the high rate of compilation errors is the disparity in the hardware and software development setups, prior to the CI compilation process. Our results shows the potential of adopting fault localization and automatic program repair techniques for these kinds of issues in CI systems.

Our contribution focuses on conducting a thorough analysis of compilation errors to inform our efforts in developing automatic solutions. This analysis encompasses spatial and temporal aspects, including the examination of resolution times, size, and distance by tracing the corresponding fixes.

These metrics provide information about the time taken to resolve errors, the extent of code modifications required, and the spatial distribution of fixes within the code base. Our analysis reveals that the long resolution times for frequent error types do not necessarily result in larger resolution sizes.

Our paper is structured as follows: Section II presents the background and motivation of our research. Section III

This work was partially supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

provides an overview of related work on compilation errors in CI and dependency errors. Section IV details our study design, outlining the employed methodologies. Section V presents the results of our study. In Section VII, we draw conclusions based on our findings. Section VIII outlines future work.

II. Background

Continuous Integration (CI) automates the integration of code changes from multiple contributors and teams into a shared software project. CI revolves around regularly merging code changes into a central repository, triggering builds and tests through automated tools.

A. Industrial context

For large industrial embedded systems, it is common for teams to submit a high volume of code to serve multiple product variations in a centralized CI system. Despite successful local compilation and testing, developers may encounter several dependency issues during compilation in the CI.

Fig. 1 illustrates a notable distinction between the setups used by software engineers and the CI environment in a hardware and software co-development context. Most effort is spent on integrating the software with the hardware [7].

During the software development phase, engineers make use of both software and hardware simulators to establish essential functionality. Subsequently, they transition to submitting their code into the CI loop, where it undergoes quick and thorough testing in conjunction with a hardware prototype. The adoption of a centralized CI environment, serving both software and hardware, offers convenience for globally distributed development teams. Unfortunately, this unified setup often introduces an interface mismatch between the software engineer's initial setup and the CI environment.

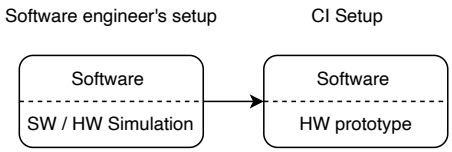


Fig. 1: Development Dependency

Many industry projects involve distributed teams, where team members are working in different locations or time zones. Miscommunications or misunderstandings about code changes can lead to undeclared items when changes are integrated.

Due to the fast-paced and intense delivery cycle, the development of hardware and software is typically conducted asynchronously. To aid developers, prioritization is given to the setup of hardware in the centralized Continues Integration (CI) environment. In the early stages, the software team primarily relies on simulators for development, compilation, and local testing. A mismatch becomes evident between hardware and software development, when developers commit to the CI. The discrepancy is characterized by a significant misalignment in development progress or objectives, which can pose challenges

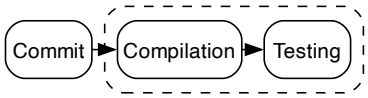


Fig. 2: CI pipeline

for achieving synchronization and integration between hardware and software components.

The gap between the CI build system and the local development environment cannot be ignored. Addressing this disparity is crucial for ensuring consistent software behavior and efficient debugging in the context of our development process. In the context of this study, we introduce Listing 1 as an illustrative case exemplifying a communication breakdown between the hardware and software development. Within this example, the variable `err` is defined in a separate file.

Listing 1: Code Example

```
1  Error: cannot convert 'ProductError' to
       'const char*'
2  Error code:
3     CATCH_THROW_ERROR(err);
4  Fixing code:
5     if (err)
6     {TRACE_ERROR(SSTR(err)); return false;}
```

Notably, the error message in this instance underscores an attempt to pass a `ProductError` object as an argument to a function explicitly designed to accept a `const char*`. The `ProductError` object originates from the hardware design. However, the original software design did not account for the `ProductError` attribute, resulting in a failure within the CI process. Consequently, this incongruity leads to a failure within the CI compilation process. Our findings shed light on the resource requirements for resolving various compilation errors, providing developers with insights to effectively prioritize their efforts in addressing these issues.

B. The CI pipelines under study

Continuous integration seeks to make integrating changes from multiple contributors easier by sharing smaller updates more frequently. The developer submits a code change to be integrated with other system components, which is then tested to ensure that no regression has occurred. It is possible to quickly provide feedback to the developer on the quality of a code change, with the correct architecture and test planning.

Implementing a CI system for embedded software can provide immediate feedback, reducing error correction costs when multiple teams are working on the same project [8]. The CI process in this context is commit-based, with a build triggered as soon as a commit is pushed.

As shown in Fig. 2, a *pipeline* is a series of actions that run sequentially, and the actions are always run together as an atomic unit. The steps that form the CI pipeline consist of distinct subsets of tasks named commit, compilation, testing, and delivery. The CI under study is running with software that is written in C/C++.

Our study was centered on four key projects: three test frameworks and one core product, each with a decade-long implementation history. This deliberate selection offers us a holistic perspective on the Ericsson CI system. All four projects undergo testing and verification using the same CI infrastructure, despite involving globally distributed developers and development teams. This setup enables the shadow job machinery to comprehensively collect data across all projects.

Definitions: A *commit* is a set of files that pushes the latest changes of the source code to the main branch. A *patch* is a file containing the set of differences between two versions.[1] A *patch* is associated with a *commit*. *Compilation* consists of code formatting, static analysis, and the actual compilation. *Testing* executes unit and integration tests on the product.

III. Related work

CI has seen substantial recent development, capturing attention from both research and industry. Hilton et al. [9] find that CI is widely used within the most popular open-source projects. Studying its usage, cost, and benefits, they find that CI helps projects release twice as often. Vasilescu et al. [10] study 246 GitHub projects using CI aiming for productivity and quality, showing that CI helps detect more bugs by core developers. Vassallo et al. [11] compare CI build errors in 349 Java open-source projects and 418 projects in an industrial organization. They reported that open-source projects' most frequent failure types are testing, compilation, and dependency issues. Furthermore, industrial projects' most frequent failure types are testing, release preparation, and static analysis.

Beller et al. [12] investigate the impact of CI and conclude that CI build failures are caused by test failures. The test execution part of the test-debug-fix cycle is already automated to a large extent in the software industry [13]. This is partially due to the agile "revolution", which had brought expectations of quick feedback to developers on product quality. As a result, continuous integration has emerged as a critical productivity-enhancing tool in the software industry over the last decade [14].

Relatively little research has been conducted on compilation errors in industrial CI environments.

A. Compilation errors in CI

The feedback from the compiler is one of the most important influences in software development. Zhang et al. [15] study compilation errors on 3 799 open-source projects. They investigate the most common compilation error types and their fix time. They manually analyze 325 broken builds to summarize fix patterns of the ten most common compilation error types. Previous research [16] has highlighted that compilation error messages can be challenging to interpret and may not be as effective, especially for novice developers. Compilation error messages can be notoriously difficult to comprehend, as shown by Rosen et al. [17]. In the context of novice programmers, the work of Traver [18] shows that compilers can detect common programming errors, but they usually do not pinpoint error locations accurately.

Seo et al. [19] studied compilation errors in Google's build process, focusing on Java and C++ environments in a software-centric, cloud-based system. Our research identifies dependency issues as the main cause of compilation errors, constituting 84 % of errors across 7 categories. While aligning with Seo's findings, our embedded system and fully automated remote CI system result in significantly different dependency error proportions.

Previous research has shown the prevalence of compilation errors in industrial CI systems [3]. Notably, our study investigates a hardware-based platform, unlike Seo's study, which was based on a purely software platform. Our results align with Seo's in terms of error categorization, as we also classify errors into 5 categories. However, our approach to calculating resolution time differs from Seo's methodology. Specifically, we calculate resolution time from the last failure CI build, preceding the successful build, rather than from the first failure CI build. This modification allows us to mitigate the influence of unrelated code changes in our analysis. These differences underscore the importance of our investigation within our specific context.

Barrak et al. [20] study on 27 675 Travis CI builds of 15 GitHub projects. They identified that features from the build history, author, code complexity, and code/test smell dimensions are the most important predictors of build failures. Ivens et al. [21] conducted an analysis of 18 industrial projects within a software company, wherein they calculated 13 metrics for each project based on existing literature related to build failure analysis. Their findings revealed significant correlations between the factors under study and the duration required for correcting build failures. They observed that build failures involving a higher number of modified lines of code and files tended to necessitate longer correction times. Previous research has explored CI issues in closed-source projects within industrial settings, and our study takes a more specialized approach by investigating the intricacies of compilation within embedded systems. Our research addresses a critical gap in understanding the CI process within this niche area.

B. Dependency errors

Software bugs have been studied extensively, yet rarely have studies been conducted on dependency bugs. Kerzazi, Khomh, and Adams [22] study build failures. Our previous work shows that dependency errors take a significant portion of industrial build failures [3]. Fischer-Nielsen et al. [23] characterize the dependency bugs in the Robot Operating System (ROS) and study the pervasiveness and potential solutions of these bugs. Khazem et al.'s research [24] into how portable software can be given changes to the toolchain (C/C++ compiler) or standard C library demonstrates the challenge of producing reproducible platform-independent software. Zakaria et al. [25] explore mechanisms to find dependencies of High-Performance Computing (HPC) in the context of a taxonomy of software distribution. Resolving dependency errors is a

[1] As the project involves several repositories, we consider a patch to be the union of the outputs of each intra-repository *git diff* command.

known subject in software configuration management [26]. To date, there has been a lack of sufficient work in identifying the precise locations of dependency errors.

C. Fault localization with log parsing

Fault localization techniques are widely proposed and developed in a broad spectrum. Wong et al.'s study catalog provides a comprehensive overview of such techniques and discusses critical issues and concerns pertinent to software fault localization [6].

Log parsing is crucial for fault localization, but manual log structuring using rule-based approaches is time-consuming and not scalable [27]. Academic research has proposed automated log analysis techniques like LogRAM, DeepLog, and LogTools [28]–[30]. However, limited research has been conducted in complex industrial settings.

Logs serve as a valuable resource for diagnosing system failures. Prior research has explored reconstructing failed executions or differentiating execution flows based on log data [31]–[33]. Another line of work focuses on identifying root-cause-related log messages by comparing logs during failure periods with reference logs without failures. Notable approaches in this realm include LogCluster [34], Log3C [35], and Onion [36]. The aforementioned studies lack systematic analysis within the industrial CI system, let alone a specific focus on compilation errors.

Ziftci and Reardon [37] study integrating fault localization techniques into a continuous integration system at Google. Their work indicates that various fault localization techniques are unsuitable for rapid development cycles, as at Google. Hassan and Zhang [38] carry out a study on a large software project at IBM, using classifiers to predict whether a build would pass a certification process.

Our study builds upon these prior efforts by combining log parsing and commit tracing techniques to gain insights into compilation errors within an industrial CI context.

IV. Study Design

A. CI diagnostics solution

We propose a CI diagnostics solution that analyzes the outcomes of each CI invocation without disrupting the original pipeline. This solution, which analyzes the build outcomes, is referred to as a *shadow job*. A shadow job is a duplicate of a configuration that runs in parallel with the main job. Its design enables it to gather data from each execution step without disrupting the original CI pipeline. It is essential that our diagnostic solution does not disrupt the regular CI process; furthermore, it should be efficient.

In highly active product environments, adding new steps to the main job can be both expensive and resource intensive. A more practical approach is to design a separate CI diagnostic solution that operates in parallel, utilizing a low-cost server, since the allocated hardware and software resources are already in use when the main job is triggered. This parallel design enables the implementation of the diagnostic solution without impacting the main job, ensuring scalability and the independent addition of new functionalities to the shadow job. By decoupling the diagnostic process from the main job, the solution can operate efficiently, providing valuable insights without disrupting the primary development workflow.

As illustrated in Fig. 3, the shadow job collects data from commits, source code, and compilation logs in the CI system. In step 1, the shadow job gathers commits that merge into the main branch of the product. Within each commit, the shadow job collects the patch that triggers the compilation errors and the patch that fixes the compilation errors for later comparison. In step 2, the shadow job collects the compilation build logs. Step 1 and step 2 are interconnected, since one commit with multiple changes in step 1 triggers the compilation build log in step 2. The shadow job collects and analyzes the build logs based on merged commits. If a commit is merged, the shadow job follows patches to locate each build log triggered by each patch.

In step 3, if a build fails, the shadow job extracts the error messages and the corresponding error code position (line number in the code) to obtain the error code position. In step 4, the shadow job extracts the error correction patch's delta code position (line number in the code). Similar to steps 1 and 2, steps 3 and 4 are interconnected.

In step 5, the shadow job compares the errors and delta code positions. In step 6, we calculate the resolution time, resolution size, and resolution distance.

We design the following studies to answer five research questions about the prevalence of different compilation errors (RQ1), the resolution time, size, and distance of different types of fixes (RQ2–4), and whether these metrics are correlated (RQ5):

1) RQ1: What are the most common compilation errors in an industrial CI system?

As our previous research [3] shows, compilation errors are a major contributor to CI build errors. We want to understand industrial CI systems' most common compilation errors and some of their causes. If the build fails, we collect the corresponding commit Change-ID for each build. In addition, the corresponding exception types are stored.

Next, error messages from the build log are systematically categorized into distinct classes, encompassing Dependency, Syntax, Type Mismatch, Semantic, and Others [19], facilitating a comprehensive analysis of their characteristics. Because builds with more than two compilation errors are rare, we can analyze data from patches that fix broken builds quite reliably, as most fixes target one or two compilation errors. Furthermore, we assume that the patch leading to the first successful build after a series of failures is the patch that fixes the compilation error.

2) RQ2: What is the resolution time for fixing different compilation errors?

Here we measure developers' elapsed time on fixing different compilation error types, corresponding to the collected compilation error builds. Additionally, we highlight and describe example instances of fixes.

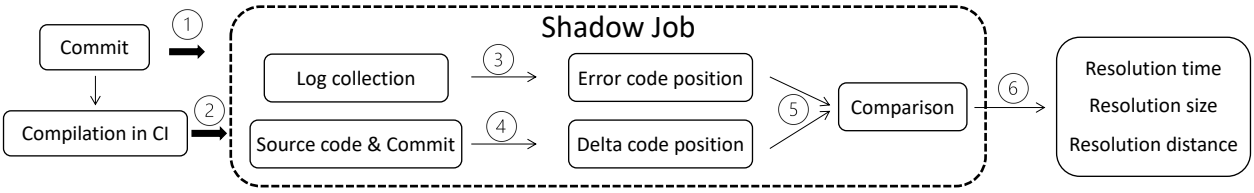


Fig. 3: Shadow Job

It is important to note that resolution time can be influenced by various human factors and the specific error-handling processes employed by different companies or organizations. Within large and complex organizations, multiple developers may be involved in resolving errors, which can potentially introduce permission issues when code changes span across different functions.

As discussed in Section II, there is often a mismatch between software and hardware design at the early stages of development. Consequently, even if software developers successfully compile and test their code locally with the simulator, there is still a possibility of compilation failures in the CI environment. In such cases, developers may need to wait for an updated version of the hardware design in the CI system.

Our research is centered on establishing the relationship between the commit that initiates the compilation error and the commit responsible for its resolution. In contrast, other studies, such as the one by Seo et al. [19], focus on the time between the completion of the first failing build and the start of the subsequent successful build. This approach may be useful in analyzing the overall cost of resolving compilation errors and avoids underestimating the resolution time. Our method accounts for the fact that the fix process has to be managed and delegated, and we assume that intermediate commits may correspond to unrelated tasks. Therefore, we specifically consider the failing build preceding the successful build within a series of builds.

Figure 4 illustrates how we measure the resolution time. In this example, there are at least two failing builds before the first successful build. We measure the time from the beginning of the lastest failing build to the beginning of the successful build, capturing the duration of the resolution process. While this may under-approximate the time taken to resolve a problem, we believe it is more accurate in our situation, taking into account the embedded development process.

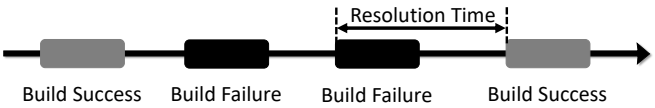


Fig. 4: Resolution Time

3) RQ3: What is the size of error corrections?
In addition to analyzing resolution time, we also look at the resolution size of each fix. We utilize the information in the failed build log, including the commit ID and the faulty line of code in the faulty files. With this information, we can extract the faulty file. Next, we proceed to extract the first subsequent successful compilation build. We can determine the resolution size by comparing the faulty file with the corresponding file from the successful build. The number of deletions and additions to the file determines the resolution size as follows:

Size 0 indicates either a permission change or an alteration of a binary file. Size 1 indicates a single line of code being deleted or added. Size 2 indicates a single line of code being modified, one line being added and another one deleted, or two lines being deleted or added, respectively.

4) RQ4: What are the resolution distances for different error types?
We aim to analyze the distance between the lines of code where the fixes are made and the lines of code indicated by the error messages. This information helps us identify the code sections that need to be modified in order to address the compilation errors effectively. Therefore, we investigate the feasibility of automatically locating error corrections. The failed build log includes a commit id and faulty line of code of faulty files. With this information, we extract faulty lines of code and their build environment to reproduce the faulty build. We also extract the patch within one merged commit that leads to the failed CI build caused by compilation error as the error code base with the number E. We then extract the next patch that leads to the CI build passing the compilation stage as the fixing code base, with the number F_n. Finally, we compare the error code base and fixing code base to have the code difference. In addition, we also extract the indication of the error line of code from the CI failure log.

We calculate the resolution distance as shown in Equation 1. We may discover multiple fixing locations in one patch with line numbers F_1, F_2, ..., and F_n. We then take the minimum of all distances between the compilation error location E and the nearest fix F_i. Thus, the resolution distance D is the minimum value in a tuple $(D_1, D_2, ..., D_n)$.

$$\begin{aligned} D &= \min(D_1, D_2, \ldots, D_n), \quad \text{where} \\ D_i &= |F_i - E| \quad \text{for } i = 1, 2, \ldots, n \end{aligned} \tag{1}$$

5) RQ5: Which types of compilation errors are suitable to apply fault localization and automatic program repair in industrial embedded systems?
We want to understand the correlation between resolution distance and resolution size. Based on their frequency of occurrence, we focus our investigation on the top four error types. These error types are deemed significant for potential

automated fault localization and automated program repair. Furthermore, we have gathered sufficient data to explore any potential correlation between these error types and their resolution attributes.

To calculate the correlation matrix based on resolution distance and each error type's corresponding resolution size and time, we use the Pearson correlation coefficient [39] as a statistical measure that quantifies the strength of the linear relationship between different continuous variables.

B. Data collection

Our data collection was conducted on a single project at Ericsson over the course of a year. This project is highly active and involves the development of embedded software.

As our data were collected from the centralized CI system, our data collection lacks errors encountered when a developer compiles locally. We therefore do not count errors that developers can resolve locally before committing.

V. Study Results

A. Results for RQ1

To answer RQ1, we automatically collect the build logs through the shadow job in step 2 in Fig. 3. We map the error messages into 14 compilation error types based on the Yocto project [40] compiler configuration files. The Yocto Project is an open-source collaboration project that provides templates, tools, and methods to help create custom Linux-based systems for embedded products.

Key insight of RQ1:

Dependency issues contribute to 76 % of all compilation errors, highlighting the challenges in reconciling the disparities between the embedded CI system and local development environments.

Table I shows 14 types of compilation errors and their proportion distribution. The ratio of compilation error types ranges from 0.36 % to 40.05 %. Subsequently, we classify the different error types into 5 classes, as illustrated in Fig. 5.

As can be seen from Table I, the top five types of compilation errors are responsible for 89.25 % of all errors. Error type *was not declared* takes the majority of compilation errors, with 40.05 %. This is significantly higher than other pure software projects reported by Seo et al. [19]. This error message indicates that the compiler has encountered a reference to a variable or function that has not been declared. The reason this cannot be found in the local environment before commit is that there is a gap between the local environment and the CI environment, especially at the early stage of development.

We want to understand the implications of the gap between the local and CI development environment. We adopt the classification of compilation errors from the literature [19] to categorize different error types into classes—the classes of each error type are shown as the second column in Table I.

TABLE I: Compilation Error Statistics

No.	Error type	Class	%
1	was not declared	Dependency	40.05
2	has no member named	Dependency	20.18
3	expected X before Y token	Syntax	11.77
4	does not name a type	Dependency	8.89
5	no declaration matches	Type mismatch	8.36
6	no such file or directory	Dependency	2.76
7	ld returned	Dependency	2.21
8	invalid conversion	Type mismatch	1.53
9	unused variable	Dependency	1.14
10	does not have any field named	Type mismatch	0.82
11	cannot allocate an object of	Semantic	0.73
12	of non-class type	Other	0.71
13	cannot convert	Type mismatch	0.49
14	static assertion failed	Syntax	0.36

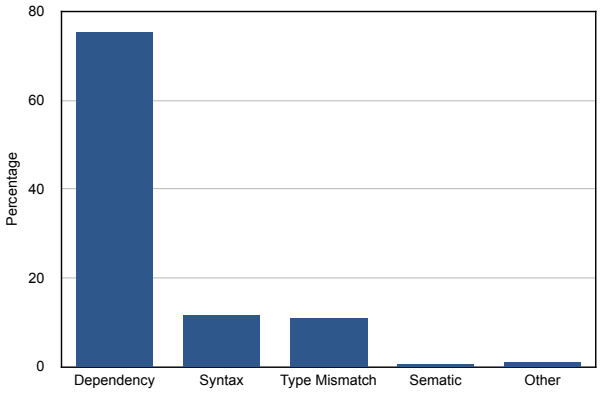


Fig. 5: Compilation error class

Grouping these subclasses of faults allows us to identify common characteristics and similarities, indicating that they can be addressed using similar automated solutions. *Dependency* refers to errors that are missing packages, libraries, or other resources. Error types like *was not declared*, *has no member named*, *does not name a type*, *no such file or directory*, *ld returned*, and *unused variable* fall into class *Dependency*. *Syntax* refers to errors that occur when the compiler or interpreter encounters code that does not follow the rules or syntax of the programming language. Error types like *expected X before Y token* and *static assertion failed* fall into class *Syntax*. *Type mismatch* refers to a mismatch between the types of variables or expressions used in an operation or assignment. Therefore, error types like *no declaration matches*, *invalid conversion*, *does not have any field named*, and *cannot convert* fall into class *Type mismatch*. *Semantic* error indicates a problem with the meaning or interpretation of the code rather than its syntax or dependencies. Therefore, *cannot allocate an object of* falls into class *Semantic*. Error type *of non-class type* indicates that the variable involved in the operation is not a class or struct type. We categorize it as *Other*.

Fig. 5 shows the distribution of different classes. It shows the *Dependency* class takes 76 %. Our finding further sub-

stantiates our previous research [3]. The proportion is also significantly higher than purely software projects reported by Seo et al. [19]. The gap between local and CI development environments with embedded systems in the loop is much larger then we expect. In addition, we also see that *Type mismatch* and *Syntax* have similar proportions, around 12 %. Our result is therefore different from Seo et al. [19] reporting that *Type mismatch* is the second biggest class.

B. Results for RQ2

This section examines each error type's resolution time. Fig. 6 shows the box plot for resolution times. The box is bounded by the 25 and 75 percentiles, and the line within the box is the median value. We normalize resolution times to a scale of [0,1] for confidentiality reasons.

Errors that occur more frequently tend to require a longer resolution time. For example, the error types of *was not declared* and *has no member named* exhibit a noticeably longer resolution time compared to other error types. Additionally, their distribution has a higher variance than that of other error types. In our CI system, each modification involves various patches; these patches not only fix compilation errors but also introduce different functionalities. Consequently, a larger dataset leads to a wider distribution of resolution times. This aligns with the fact that the *was not declared* error accounts for 40.05 % of errors, as illustrated in Table I.

Key insight of RQ2:

The observation that compilation errors require a significant effort to resolve despite their high frequency is counterintuitive and surprising. One would typically expect that the more frequently a specific type of error occurs, the faster the resolution process would be.

Based on their frequency of occurrence, we focus our investigation on the top four error types, which collectively constitute 81 % of our dataset.[2] These error types are deemed significant for potential automated fault localization and automated program repair, given their substantial representation in our data.

Certain error types, such as *no declaration matches*, *no such file or directory*, and *unused variable*, are relatively simple to resolve, as indicated by shorter resolution times. As the frequency of errors decreases, it typically leads to a reduction in resolution time. Error types like *ld returned* (a linker error) take longer to fix due to their complexity. Linker errors often arise when a project has complex dependencies between different modules or libraries. These dependencies can be difficult to identify and resolve, especially in larger projects. In an embedded system, each hardware has its own specific set of libraries. Therefore, during local development, the software is typically compiled with the libraries specific to the hardware being used. This approach is more practical and efficient than providing cross-compilation for all different hardware configurations, which would be costly and time-consuming.

[2]Due to the sensitive and proprietary nature of the industrial context, we are unable to share specific detailed data, including the total number of errors.

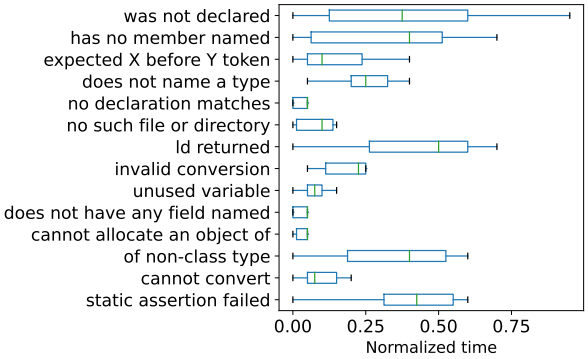


Fig. 6: Resolution time statistics

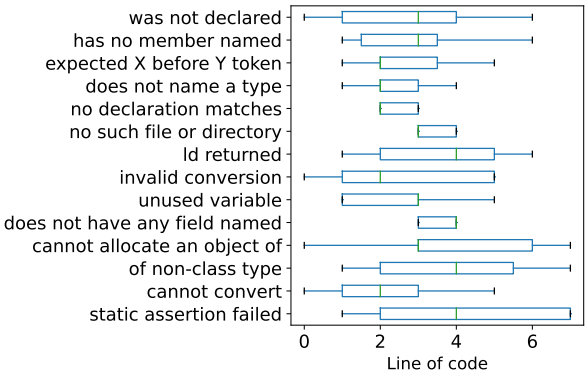


Fig. 7: Resolution size statistics

C. Results for RQ3

In this section, we examine the resolution size of each error type. Remember that we count both additions and deletions here, so four lines typically correspond to two deleted and two added lines (which can be an edit within two lines); changes in binary files count as zero lines. Similar to Fig. 6, Fig. 7 displays a box plot illustrating the distribution of resolution sizes for each error type.[3] Additionally, Fig. 8 presents the distribution of resolution sizes for all errors, categorized into 0–7 lines of code.

Key insight of RQ3:

The majority of resolution size to compilation errors are changes between 1 and 4 lines.

[3]In case two quartiles have the same number of data points, the quartile boundaries are calculated using the average between the largest element of the lower quartile and the smallest element of the higher quartile, which sometimes results in fractions.

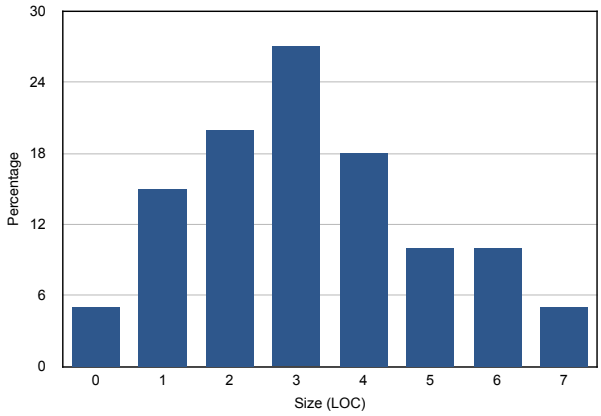


Fig. 8: Resolution size distribution

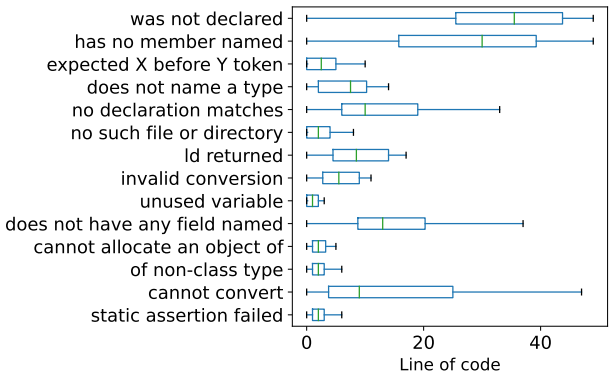


Fig. 9: Resolution distance statistics

As shown in Fig. 7, the most prevalent error type, identified as *was not declared*, tends to have a relatively small resolution size, typically ranging from 1 to 4 lines of code. The top 5 error types generally have a smaller resolution size, usually fewer than 4 lines of code. The error type *ld returned* leads to a noticeably larger resolution size and time, which is due to the complexity of the linking process. Linking processes can be intricate, particularly in a large embedded system with numerous dependencies. Additionally, linker errors may arise because of platform-specific differences. Reproducing the same platform can be challenging when the developer does not have the same environment as the CI system.

As depicted in Fig. 8, the most frequent resolution size observed is 3 (in approximately 27 % of the cases). The second most common resolution size is 2, constituting around 20 % of the occurrences. Larger resolution sizes such as 5, 6, and 7 are much less common.

D. Results for RQ4

We measure the resolution distance in RQ4 in Section IV-A4. Fig. 9 displays a box plot illustrating the distribution of resolution distance for each error type.

As shown in Fig. 9, the error types *was not declared* and *has no member named* exhibit a relatively large distance between them. The *was not declared* error type demonstrates a larger resolution distance. The difference in resolution distance between the *was not declared* and *has no member named* errors can be attributed to the specific nature of the corrections required. In the case of the *was not declared* error, the correction typically involves adding a declaration at the beginning of the file, rather than modifying the specific code line indicated by the error message.

The *was not declared* error type typically occurs 25 to 42 lines away from the error message location, while the *has no member named* error type primarily appears 15 to 38 lines away from the error message location. This discrepancy can be attributed to the fact that header files often provide declarations for functions and classes that are defined in separate source files. As a result, resolving the *was not declared* error may require modifications at the beginning of the file, leading to a larger resolution distance.

Key insight of RQ4:

Many frequently occurring error types are close to the reported location and therefore good targets for automatic fault localization. However, the top two most frequent error types require a full syntactic source code analysis to locate automatically due to their large resolution distance.

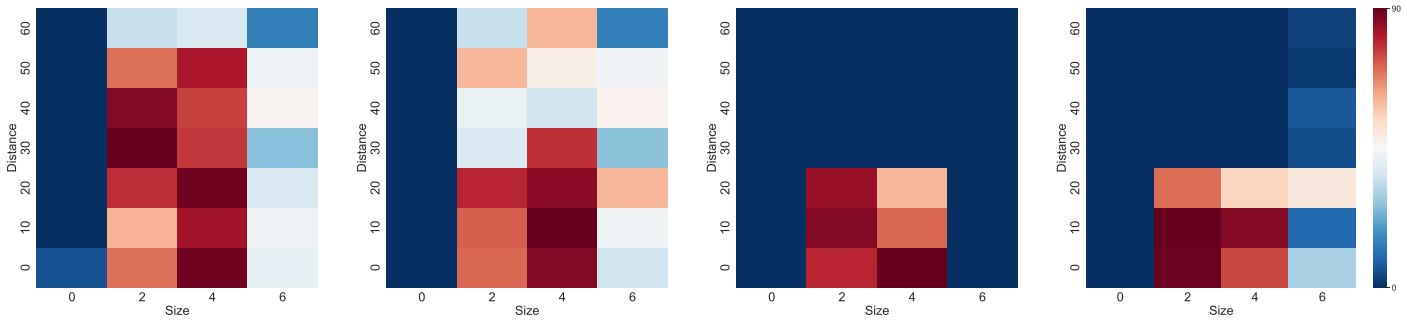
E. Results for RQ5

The results of RQ2 in Section IV-A2 and RQ3 in Section IV-A3 indicate that the top 4 error types are more representative and exhibit greater consistency. Moreover, the top 4 error types account for 80.89 % of the dataset, making it reasonable to focus our discussion primarily on these error types.

The top 4 error types demonstrate a relatively smaller resolution size and a distance of usually up to 50 lines from the location of the compilation error. While they do not require extensive modification to be fixed, many fixes for the top 2 error types are “far away” from the error message location in terms of resolution distance and size. This finding suggests that the error types *was not declared* and *has no member named* may be challenging for automatic fault localization techniques.

Due to their larger resolution distances and sizes, these error types may require more manual intervention, code analysis, and extensive code modifications to rectify. As a result, automated fault localization methods that primarily rely on localizing errors based on error messages or limited code regions may not be effective in accurately pinpointing the root cause of these particular error types.

We evaluate the correlation between resolution distance, size, and time for each of the four most common error types. Table II presents the correlation values for the top 4 error types and overall error types, measuring 3 types of correlations: distance to size, distance to time, and size to time. Fig. 10 illustrates the heatmap for the top 4 error types based on resolution sizes and distances.



(a) was not declared (b) has no member named (c) expected X before Y token (d) does not name a type

Fig. 10: Heat map showing the distribution of fix size and distance for the four most common compilation errors

TABLE II: Correlation Between Key Attributes

Error	**Distance–Size**	**Distance–Time**	**Size–Time**
was not declared	0.227	0.034	−0.287
has no member named	0.047	−0.091	0.155
expected X before Y token	0.053	0.114	0.057
does not name a type	0.220	−0.040	0.082
All errors	−0.069	0.192	0.077

In Table II, the first row represents the correlation values for the error type *was not declared*. The highest positive correlation is between distance and size, with a value of 0.227. The highest negative correlation is between size and time, with a value of −0.287. These values are all low, showing that resolution distance, size, and time are almost unrelated to each other. A small fix that is close to the location of the compilation error may still be hard to find for a human.

The findings align with the heatmap presented in Fig. 10 (a)-(d), where each cell's shade corresponds to the color bar on the right side of the heatmap. The bottom shade represents the minimum value, while the top shade represents the maximum value.

In Fig. 10 (a), the first column displays a nearly consistent color, indicating that errors rarely occur when the resolution size is 0. Moving to the second column, where the resolution size is two lines of code, the color changes based on the frequency of occurrence. Specifically, when the resolution size is two lines of code, the majority of resolution distances cluster around 30 lines of code.

Key insight of RQ5:

Resolution distance, size, and time are independent of each other.

The fixes for the error type *has no member named* tend to cluster around a resolution size of 2–4 lines of code and a resolution distance of 0–20 lines of code, as shown in Fig. 10 (b). Compared to the error type *was not declared*, fixing *has no member named* appears to require relatively less effort. The heatmaps in Fig. 10 (c) and (d) indicate slightly varying resolution distances for different errors, but the resolution sizes for the most common errors are quite similar.

F. Threats to validity

1) Internal: In a complex industrial embedded CI system, dependency-related failures are prevalent [3]. Extracting data from real-use code prevents us from rerunning executions with specific environments after discarding outdated dependency build packages, limiting our adoption of fault localization and automated program repair in such a CI system.

Furthermore, a single commit often includes multiple changes. When calculating resolution distance and size, we consider only the files indicated by the error message, potentially leading to occasional miscalculations. Given that most errors are related to dependency declarations, we find this approach suitable.

While our method of calculating resolution time aligns with the development process according to our judgment, a more precise measurement would involve accounting for working hours, capturing the actual time designers spent fixing errors.

The current approach to computing change size, encompassing both code and binary changes, may pose challenges. To enhance clarity and ensure the visibility of binary changes, it is suggested to consider a separation between changes in code and binaries.

2) External: In calculating both resolution time and resolution distance, it is inevitable that the human factor influences the study results. Variances in development pace, work methodologies, developers' habits, and the distribution of teams and developers are key factors that can affect the outcomes. To mitigate these influences, conducting in-depth analyses through interviews with developers emerges as a valuable approach.

In addition to this, the correlation coefficient choice depends on data characteristics and assumed relationships between variables. Opting for the Pearson correlation, suited to linear relationships in our data, may limit the study scope to

linear dependencies. Exploring alternative metrics like min-max resolution time and distance, employing machine learning models, and utilizing different analyses such as SHAP [41] can provide a broader perspective on metric dependencies.

VI. Discussion

Our analysis of over 40000 builds reveals that a significant portion of industrial build failures can be attributed to a dependency on its hardware-in-the-loop CI system. The integration of software with hardware prototypes in CI poses a challenge in bridging the gap between the CI and local development environments. The complexities introduced by hardware-in-the-loop testing and the difficulty in reproducing the hardware-in-the-loop environment accurately within the CI system contribute to build failures that are hard to resolve. Addressing this mismatch is crucial for improving the stability and reliability of the build process in an embedded industrial CI system.

Our finding of resolution time reveals that defects that occur more frequently tend to require a longer resolution time. Therefore, it would be highly advantageous to prioritize the development of automated solutions for resolving the most frequently occurring error types.

A lengthy resolution time for the top 4 errors does not necessarily mean a large resolution size. In fact, the majority of fixes involve changes of only 1 to 4 lines of code. This quantification of resolution size inspires developers by highlighting the success of small fixes. Existing automated program repair approaches designed for one-line changes can be applicable in addressing these compilation errors efficiently.

Further examination of the top 4 error types shows that resolution time, size, and distance are independent attributes. This suggests that compilation errors possess inherent statistical characteristics that render them more conducive to automated detection and remediation.

That says no specific type of compilation error lends itself to a single simple strategy for automated fixes. Due to this, we should prioritize frequently occurring errors. In light of the absence of such data, it is advantageous to prioritize techniques that specifically target the most common error types.

In summary, our study suggests that automated fault localization and program repair efforts show promise for the identified error types. However, addressing resolution distance and size separately is crucial for developing effective techniques in these contexts. Our results indicate a significant potential for implementing automatic fault localization techniques for compilation errors, emphasizing the need for tailored approaches.

VII. Conclusion

We conducted a statistical analysis on over 40000 builds in a highly active product using a scalable CI diagnostics solution called “Shadow Job.” We extracted compilation errors into 14 error types and classified them into 5 classes. The results highlight the significance of the compilation step in industrial-embedded development CI systems. 76 % of compilation errors are due to dependencies between hardware and software and a mismatch between the embedded environment (offered by the CI system) and the local development environment.

Our study also categorized, quantified, and analyzed compilation errors based on three key attributes: resolution time, size, and distance. More frequent compilation errors require more time to be resolved, although the size of the fixes is always small, mostly within one or two lines of changes. The fix location can be relatively far away from the location of the compilation error, but resolution time, size, and distance are not correlated among each other.

The fact that the five most frequent compilation errors make up 76 % of all compilation errors suggests that a collection of specialized strategies to prevent or automatically locate and repair these five errors can be useful.

VIII. Future work

In future work, a key focus will be on the practice of automatic fault localization and automatic program repair, specifically targeting the reduction of compilation errors caused by configuration issues and dependency problems. Large modular systems often have numerous dependencies reflected in their configuration settings, which are typically stored in different formats that have evolved independently.

To further enhance our ”Shadow Job” implementation, we aim to expand its capabilities beyond diagnosis and providing valuable error location information. Our goal is to develop an automated solution that can automatically fix identified errors. For instance, for multi-line compilation errors in the C language, one approach could involve using a neural network-based approach, such as DeepFix [42]. Alternatively, a novel approach would involve leveraging a large language model like Keep [43] to address these errors.

By incorporating advanced techniques such as neural networks or large language models, we can enhance the automated resolution capabilities of our system, facilitating the efficient and effective handling of multi-line code fixes. This approach holds promise for reducing the manual effort required to fix complex compilation errors and improving the overall development process.

We also intend to extend the study to other projects and/or companies. This will address the external validity threats and apply the findings to more CI pipelines to validate our results.

References

[1] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, “Agile software development methods: Review and analysis,” *CoRR*, vol. abs/1709.08439, 2017.

[2] L. Chen, “Continuous delivery: Huge benefits, but challenges too,” *IEEE Softw.*, vol. 32, no. 2, pp. 50–54, 2015.

[3] H. Fu, S. Eldh, K. Wiklund, A. Ermedahl, and C. Artho, “Prevalence of continuous integration failures in industrial systems with hardware-in-the-loop testing,” in *IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE 2022 - Workshops, Charlotte, NC, USA, October 31 - Nov. 3, 2022*, pp. 61–66, IEEE, 2022.

[4] L. Jonsson, “Increasing anomaly handling efficiency in large organizations using applied machine learning,” in *35th International Conference on Software Engineering, ICSE ’13, San Francisco, CA, USA, May 18-26, 2013* (D. Notkin, B. H. C. Cheng, and K. Pohl, eds.), pp. 1361–1364, IEEE Computer Society, 2013.

[5] N. Kerzazi and F. Khomh, "Factors impacting rapid releases: an industrial case study," in *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014* (M. Morisio, T. Dybå, and M. Torchiano, eds.), pp. 61:1–61:8, ACM, 2014.

[6] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Software Eng.*, vol. 42, no. 8, pp. 707–740, 2016.

[7] V. Antinyan, M. Staron, W. Meding, P. Österström, E. Wikstrom, J. Wranker, A. Henriksson, and J. Hansson, "Identifying risky areas of software code in Agile/Lean software development: An industrial experience report," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014* (S. Demeyer, D. W. Binkley, and F. Ricca, eds.), pp. 154–163, IEEE Computer Society, 2014.

[8] S. Planning, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology*, vol. 1, 2002.

[9] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "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, Singapore, September 3-7, 2016* (D. Lo, S. Apel, and S. Khurshid, eds.), pp. 426–437, ACM, 2016.

[10] B. Vasilescu, Y. Yu, H. Wang, P. T. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in GitHub," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015* (E. D. Nitto, M. Harman, and P. Heymans, eds.), pp. 805–816, ACM, 2015.

[11] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. D. Penta, and S. Panichella, "At 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, IEEE Computer Society, 2017.

[12] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: an explorative analysis of Travis CI with GitHub," in *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017* (J. M. González-Barahona, A. Hindle, and L. Tan, eds.), pp. 356–367, IEEE Computer Society, 2017.

[13] V. Garousi and F. Elberzhager, "Test automation: Not just for test execution," *IEEE Softw.*, vol. 34, no. 2, pp. 90–96, 2017.

[14] S. Stolberg, "Enabling agile testing through continuous integration," in *2009 Agile Conference, Chicago, IL, USA, 24-28 August 2009* (Y. Dubinsky, T. Dybå, S. Adolph, and A. S. Sidky, eds.), pp. 369–374, IEEE Computer Society, 2009.

[15] C. Zhang, B. Chen, L. Chen, X. Peng, and W. Zhao, "A large-scale empirical study of compiler errors in continuous integration," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019* (M. Dumas, D. Pfahl, S. Apel, and A. Russo, eds.), pp. 176–187, ACM, 2019.

[16] B. A. Becker, P. Denny, R. Pettit, D. Bouchard, D. J. Bouvier, B. Harrington, A. Kamil, A. Karkare, C. McDonald, P. Osera, J. L. Pearce, and J. Prather, "Compiler error messages considered unhelpful: The landscape of text-based programming error message research," in *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education, ITiCSE-WGR 2019, Aberdeen, Scotland Uk, July 15-17, 2019* (B. Scharlau, R. McDermott, A. Pears, and M. Sabin, eds.), pp. 177–210, ACM, 2019.

[17] S. Rosen, R. A. Spurgeon, and J. K. Donnelly, "PUFFT - the PURDUE university fast FORTRAN translator," *Commun. ACM*, vol. 8, no. 11, pp. 661–666, 1965.

[18] V. J. Traver, "On compiler error messages: What they *say* and what they *mean*," *Adv. Hum. Comput. Interact.*, vol. 2010, pp. 602570:1–602570:26, 2010.

[19] H. Seo, C. Sadowski, S. G. Elbaum, E. Aftandilian, and R. W. Bowdidge, "Programmers' build errors: a case study (at Google)," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014* (P. Jalote, L. C. Briand, and A. van der Hoek, eds.), pp. 724–734, ACM, 2014.

[20] A. Barrak, E. E. Eghan, B. Adams, and F. Khomh, "Why do builds fail? - A conceptual replication study," *J. Syst. Softw.*, vol. 177, p. 110939, 2021.

[21] G. Silva, C. I. M. Bezerra, A. G. Uchôa, and I. Machado, "What factors affect the build failures correction time? A multi-project study," in *Proceedings of the 17th Brazilian Symposium on Software Components, Architectures, and Reuse, SBCARS 2023, Campo Grande, Brazil, September 25-29, 2023*, pp. 41–50, ACM, 2023.

[22] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pp. 41–50, IEEE Computer Society, 2014.

[23] A. Fischer-Nielsen, Z. Fu, T. Su, and A. Wasowski, "The forgotten case of the dependency bugs: on the example of the robot operating system," in *ICSE-SEIP 2020: 42nd International Conference on Software Engineering, Software Engineering in Practice, Seoul, South Korea, 27 June - 19 July, 2020* (G. Rothermel and D. Bae, eds.), pp. 21–30, ACM, 2020.

[24] K. Khazem, E. T. Barr, and P. Hosek, "Making data-driven porting decisions with Tuscan," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018* (F. Tip and E. Bodden, eds.), pp. 276–286, ACM, 2018.

[25] F. Zakaria, T. R. W. Scogland, T. Gamblin, and C. Maltzahn, "Mapping out the HPC dependency chaos," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022* (F. Wolf, S. Shende, C. Culhane, S. R. Alam, and H. Jagode, eds.), pp. 34:1–34:12, IEEE, 2022.

[26] N. Ratti and P. Kaur, "A conceptual framework for analysing the source code dependencies," in *Advances in Computer and Computational Sciences* (S. K. Bhatia, K. K. Mishra, S. Tiwari, and V. K. Singh, eds.), (Singapore), pp. 333–341, Springer Singapore, 2018.

[27] K. Rodrigues, Y. Luo, and D. Yuan, "CLP: efficient and scalable search on compressed text logs," in *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021* (A. D. Brown and J. R. Lorch, eds.), pp. 183–198, USENIX Association, 2021.

[28] H. Dai, H. Li, C. Chen, W. Shang, and T. Chen, "Logram: Efficient log parsing using nn-gram dictionaries," *IEEE Trans. Software Eng.*, vol. 48, no. 3, pp. 879–892, 2022.

[29] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, eds.), pp. 1285–1298, ACM, 2017.

[30] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019* (H. Sharp and M. Whalen, eds.), pp. 121–130, IEEE / ACM, 2019.

[31] X. Fu, R. Ren, S. A. McKee, J. Zhan, and N. Sun, "Digging deeper into cluster system logs for failure prediction and root cause diagnosis," in *2014 IEEE International Conference on Cluster Computing, CLUSTER 2014, Madrid, Spain, September 22-26, 2014*, pp. 103–112, IEEE Computer Society, 2014.

[32] J. Lou, Q. Fu, S. Yang, J. Li, and B. Wu, "Mining program workflow from interleaved traces," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010* (B. Rao, B. Krishnapuram, A. Tomkins, and Q. Yang, eds.), pp. 613–622, ACM, 2010.

[33] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Trans. Software Eng.*, vol. 47, no. 2, pp. 243–260, 2021.

[34] Q. Lin, H. Zhang, J. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume* (L. K. Dillon, W. Visser, and L. A. Williams, eds.), pp. 102–111, ACM, 2016.

[35] S. He, Q. Lin, J. Lou, H. Zhang, M. R. Lyu, and D. Zhang, "Identifying impactful service system problems via log analysis," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-*

09, 2018 (G. T. Leavens, A. Garcia, and C. S. Pasareanu, eds.), pp. 60–70, ACM, 2018.

[36] X. Zhang, Y. Xu, S. Qin, S. He, B. Qiao, Z. Li, H. Zhang, X. Li, Y. Dang, Q. Lin, M. Chintalapati, S. Rajmohan, and D. Zhang, "Onion: identifying incident-indicating logs for cloud systems," in *ESEC/FSE 2021: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021* (D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, eds.), pp. 1253–1263, ACM, 2021.

[37] C. Ziftci and J. Reardon, "Who broke the build? automatically identifying changes that induce test failures in continuous integration at Google scale," in *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*, pp. 113–122, IEEE Computer Society, 2017.

[38] A. E. Hassan and K. Zhang, "Using decision trees to predict the certification result of a build," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pp. 189–198, IEEE Computer Society, 2006.

[39] A. Rutherford, "Applied multiple regression/correlation analysis for the behavioral sciences," *British Journal of Mathematical & Statistical Psychology*, vol. 56, p. 185, 2003.

[40] H. Khandelwal, P. Mankodi, and R. Prajapati, "Enhancement of automation testing system using Yocto project," in *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*, vol. 1, pp. 697–700, 2017.

[41] S. M. Lundberg and S. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA* (I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, eds.), pp. 4765–4774, 2017.

[42] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade, "Deepfix: Fixing common C language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA* (S. Singh and S. Markovitch, eds.), pp. 1345–1351, AAAI Press, 2017.

[43] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," *CoRR*, vol. abs/2304.00385, 2023.