Understanding Build Issue Resolution in Practice: Symptoms and Fix Patterns

Yiling Lou
HCST, CS, Peking University
Beijing, China
yiling.lou@pku.edu.cn

Zhenpeng Chen
HCST, CS, Peking University
Beijing, China
czp@pku.edu.cn

Yanbin Cao
HCST, CS, Peking University
Beijing, China
caoyanbin@pku.edu.cn

Dan Hao*
HCST, CS, Peking University
Beijing, China
haodan@pku.edu.cn

Lu Zhang
HCST, CS, Peking University
Beijing, China
zhanglucs@pku.edu.cn

ABSTRACT

Build systems are essential for modern software maintenance and development, while build failures occur frequently across software systems, inducing non-negligible costs in development activities. Build failure resolution is a challenging problem and multiple studies have demonstrated that developers spend non-trivial time in resolving encountered build failures; to relieve manual efforts, automated resolution techniques are emerging recently, which are promising but still limitedly effective. Understanding how build failures are resolved in practice can provide guidelines for both developers and researchers on build issue resolution. Therefore, this work presents a comprehensive study of fix patterns in practical build failures. Specifically, we study 1,080 build issues of three popular build systems *Maven*, *Ant*, and *Gradle* from Stack Overflow, construct a fine-granularity taxonomy of 50 categories regarding to the failure symptoms, and summarize the fix patterns for different failure types. Our key findings reveal that build issues stretch over a wide spectrum of symptoms; 67.96% of the build issues are fixed by modifying the build script code related to plugins and dependencies; and there are 20 symptom categories, more than half of whose build issues can be fixed by specific patterns. Furthermore, we also address the challenges in applying non-intuitive or simplistic fix patterns for developers.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Build systems, Build failure resolution, Empirical study

*Corresponding author.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409760

ACM Reference Format:
Yiling Lou, Zhenpeng Chen, Yanbin Cao, Dan Hao, and Lu Zhang. 2020. Understanding Build Issue Resolution in Practice: Symptoms and Fix Patterns. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409760

1 INTRODUCTION

Build systems play an essential role in modern software development, facilitating developers' build activities by automatically transforming source code to executable software. Meanwhile, build failures occur frequently both in commercial and open-source software systems [17, 22, 27] and postpone software development activities with non-negligible costs [17]. Multiple previous studies [17, 32] show that developers spend non-trivial efforts in resolving their encountered build failures. To facilitate manual resolution, studies on automated build failure fixing are emerging in recent years. For example, Al-Kofahi *et al.* [6] proposed a fault localization approach for Makefiles based on dynamic execution trace analysis; Lou *et al.* [18] utilized program analysis and search-based strategies to fix general-type build failures. Although promising, existing automated techniques have been shown to have limited effectiveness in practice (especially for general-type build failures) and the state-of-the-art technique can fix only a small ratio of build failures (i.e., 18%) [18].

To provide developers and researchers with practical guidelines on this challenging problem, understanding how build failures are resolved by developers in practice is helpful. Fortunately, two recent studies have shed some light on this problem. Zhang *et al.* [32] studied fix patterns in compiler errors in build process; Macho *et al.* [19] investigated frequent resolution patterns in dependency-related build failures. However, both studies target at only a specific failure type, leaving many other categories of build failures unexplored. So far there is no comprehensive understanding of fix patterns across different types of build failures.

Therefore, in this paper, we conduct a comprehensive study on general build failures, considering both symptoms and fix patterns. We collect a dataset of 1,080 Stack Overflow (SO) posts related to build issues in three mainstream Java build systems *Maven*, *Gradle*, and *Ant* [14, 20, 26, 27]. Based on the dataset, we manually summarize build failure symptoms and fix patterns from SO question

descriptions and accepted answers; we also distill frequent topics from how-to questions to show the build knowledge that developers lack. We focus our study on the following questions that we believe could provide insights for developers and researchers.

RQ1: topics in how-to questions. How-to questions present the build knowledge that developers are inexpert at, which tend to induce future build failures. By studying frequent topics in how-to questions, we uncover the difficulties that developers face in configuring their builds and the vulnerabilities that should be addressed by automated tools and techniques.

RQ2: symptoms of build failures. By constructing a comprehensive taxonomy of build failure symptoms, we suggest frequent failure types and the symptoms neglected by previous work, which should be addressed by the future study.

RQ3: fix patterns of build failures. By studying fix patterns in each symptom category, we summarize their characteristics and commonalities, and provide insights for both developers and researchers about principled ways in build failure resolution.

Our main findings reveal a high diversity in build issue symptoms (i.e., 50 categories); and most (69.68%) build issues are related to *dependency resolution*, *parse*, *output execution*, *compilation*, and *assemble*, widely covering 25 different symptom categories; moreover, *output execution* covers 14.41% of the build issues, which are often neglected by previous studies. We also find that 67.96% build issues are fixed by patterns applied in build script code related to plugins and dependencies, among which the most two frequent patterns are correcting plugin setting and adding missing dependency; besides, for 20 symptom categories, more than half of their build issues can be fixed by specific patterns. In addition, we discuss the challenges in applying non-intuitive and simplistic fix patterns. The summary of all findings and implications is listed in Table 1.

As the first comprehensive study on build issue resolution, this paper makes the following contributions:

- Revealing the difficult build topics for developers, which may become vulnerabilities for inducing future build failures.
- Constructing a fine-granularity and comprehensive taxonomy of build failure symptoms via manual classification, which facilitates the fixing pattern analysis for build failures.
- Summarizing fixing patterns for different build failure symptoms with practical guidelines for developers and researchers in build failure resolution domain.

2 BACKGROUND

Build systems are responsible for transforming project source code into a collection of deliverables (e.g., executable software or distributable library) [20]. With the increasing demand on the building process, build systems become more complex by including various functionalities besides the basic ones (e.g., compile or assemble). For example, static code analysis, code coverage collection, and mutation testing can be integrated into the build process via third-party plugins. To facilitate the understanding of such complex build systems, we draw Figure 1 to present the build workflow by summarizing the commonalities of mainstream Java build systems *Maven*, *Ant*, and *Gradle*.

Components. *Build system* leads the entire build process following developers' instructions. The inputs of build process usually

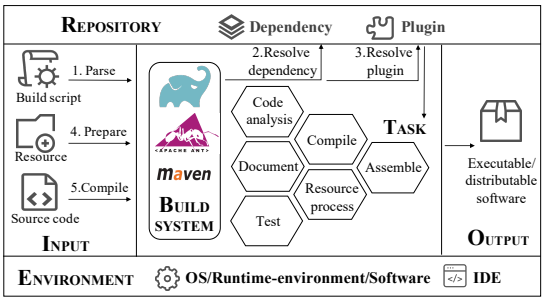

Figure 1: Build workflow

consist of *build script*, *source code*, and *resource*, while the output is assembled software in a distributable or executable format such as Jar, Ear, War, or Apk. More specifically, a build script is a collection of developers' instructions written in a script language, which varies among different build systems. For example, *Gradle* requires a build script written in Groovy DSL while a *Maven* script is written in XML format. The resource refers to the artifacts necessary for the build process (e.g., images). As for the other required non-local artifacts (e.g., dependencies or plugins), the build system downloads them from external repositories. Besides, a successful build process also relies on the *build environment*, referring to the operation system, run-time environment (e.g., memory), network setting, or other installed software. We include IDE in the build process, since besides command lines, build tasks can also be invoked in IDE. Build process is associated with these components, and thus build failures may come from any of them or their interaction.

Workflow. Once the build system is launched, it first parses and validates the input build script (e.g., checks syntax and resolves properties defined in the script code). Then build system prepares the required dependencies and plugins by downloading them from specified location (e.g., central/remote/local repository). The subsequent build process is a sequence of build tasks and each of them is actually an invocation of certain plugin. For example, *Maven* utilizes the plugin `maven-compiler-plugin` to perform compilation task and the `maven-deploy-plugin` to deploy distributable outputs; *Gradle* is capable of building Android applications with various extended functions in the plugin Android. Various build tasks are shown in Figure 1, such as test, code analysis, and document generation; tasks of compilation and resource preparation are also based on plugin invocations, but they are put outside the box since they are common build tasks and often invoked by default. After all the build tasks finish, besides the by-product (e.g., reports generated by specific plugins), the main output (e.g., an executable project) is available for execution or distribution. We regard a build as successful, only when the workflow finishes without interruption, and all the tasks and outputs behave as developers' expectation; otherwise, we consider it as a build failure.

3 METHODOLOGY

To understand how developers resolve build issues, we analyze the build-related posts on Stack Overflow (SO). Figure 2 illustrates the overview of the methodology used in our study.

Table 1: Summary of findings and implications

Findings about how-to build issues	**Implications**
F.1 Developers ask a wide range (11 high-level categories) of how-to topics on build systems, and *grammar* (27.25%), *common tasks* (19.14%), *I/O* (13.51%), *dependencies* (12.61%), and *command line* (8.56%) are asked most frequently.	**I.1** The frequently asked how-to topics imply the vulnerable components in a build script, which may be given priority attention for build failure detection and fix techniques.
F.2 Most (80.84%) questions are about build script programming and some developers complain about limited help from documents and tutorials.	**I.2** The documentation of plugins, libraries, and build tools should be improved in terms of the completeness, usability, and readability; code generation techniques may be adopted to facilitate build script generation and completion.
Findings about build issue symptoms	**Implications**
F.3 We construct a taxonomy of 50 symptom categories for build issues; and 25 error categories in *dependency resolution* (19.57%), *parse* (15.48%), *output execution* (14.41%), *compilation* (12.04%), and *assemble* (8.17%) are frequently asked by developers (69.68% in total).	**I.3** Build issue resolution is challenging due to the diversity in symptom categories; techniques for detecting and fixing general build issues should cover as broad spectrum of the categories as possible; researchers should pay attention to these categories that developers find difficult to fix.
F.4 *Output execution* (14.41%) covers build issues with delayed exposure and 65.67% of them encounter *class loading error*.	**I.4** To uncover these issues earlier: (1) introduce early examination mechanism for build outputs; (2) check the configuration consistence between run-time and build-time environments.
Findings about fix patterns	**Implications**
F.5 More than half (67.96%) of the build issues are fixed by patterns related to plugins and dependencies, covering 37 out of 50 symptom categories. The top two frequent fixing patterns are correcting plugin setting (18.27%) and adding missing dependency (13.54%).	**I.5** Our suggestion to developers is checking plugin settings and absence of dependency declaration first. Our suggestion to researchers is assigning high suspiciousness values to the script code related to plugin and dependency in designing fault localization techniques, and adding more fix operations and ingredients on these components in designing automated repair techniques.
F.6 20 out of 50 symptom categories have frequent fixing patterns, such as adding missing dependency to resolve class loading error and changing build tool version to solve plugin apply error.	**I.6** The frequent pairs obtained from our study may serve as common fix strategies for both manual and automated build issue resolution.
F.7 There is no common fix pattern for most symptom categories (60.00%) and each category has more than three fix patterns in average (range from 1 to 10), indicating that a non-negligible part of build issues are fixed case by case.	**I.7** There is no silver bullet for fixing arbitrary types of build issues. Therefore, researchers may facilitate automated resolution via embedding more prior-experience rules (including the corner cases in our studies) or mining more cases from big data to expand fix strategies.
F.8 Most (79.78%) fix patterns are simple and contain only a few lines of modification but require comprehensive and up-to-date knowledge on third-party resources to apply.	**I.8** Automated resolution techniques are suggested to include autonomous updating mechanism to update their embedded fix strategies that are related to external resources, so as to keep stable effectiveness.
F.9 Some symptom categories are fixed by non-intuitive patterns (e.g., illegal symbol fixed by changing plugin version) and in these cases developers often find it challenging to localize the causes.	**I.9** We suggest tool vendors to provide deeper hints for build issues to assist developers' resolution. Moreover, the existence of non-intuitive patterns indicates build errors with fault localization challenges.

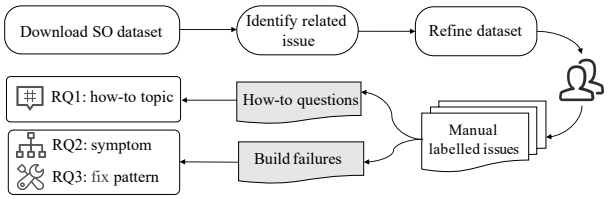

Figure 2: Overview of the methodology

3.1 Data Collection

It has been a common practice for SE researchers to get insight into developers' concerns on different SE issues by mining related posts from SO [5, 7, 8, 23, 30, 33]. In our study, we use SO as the data source because: (i) as one of the most popular community-driven Q&A websites, the users in SO range from novices to experts, increasing the diversity of the analyzed issues; (ii) developers often seek for help in SO after they cannot find solutions in documents or internet search, leading to more unsolvable and non-trivial build problems in our analyzed data; (iii) SO inherently contains build issues with implicit symptoms which are often hard to be captured in reproduced or historical build data, increasing comprehensiveness of the dataset. We construct our dataset of build issues from SO in the following steps.

Step 1: download Stack Overflow dataset. We download SO dataset $\mathcal{S}_{all}$ from the official Data Dump in December 2019 [2], which covers 18,597,996 SO posts from July 31, 2008 to December 1, 2019. We keep the post identifier, question body, answers, and tags in the metadata of each post for our study.

Step 2: identify build-related posts. An SO post usually has 1 to 5 tags [8], which indicate the belonging domains of the post. Therefore, we utilize a set of build-related tags to identify and extract build-related posts from $\mathcal{S}_{all}$. Similar to previous work [8], we construct a set of build-related tags as follows.

(1) We start with a tag set $\mathcal{T}_{ini}$ that includes our initial build-related tags. As this study focuses on three mainstream Java build systems *Maven*, *Gradle*, and *Ant* that are widely studied in previous work [14, 20, 26, 27], we define an initial tag set $\mathcal{T}_{ini} = \{ant, mvn, gradle\}$.

(2) We extract a subset of posts $\mathcal{S}_{part}$ whose tags match at least one tag in $\mathcal{T}_{ini}$ and identify more build-related tags based on $\mathcal{S}_{part}$. Specifically, we construct a set of candidate tags $\mathcal{T}_{cand}$ by extracting all the tags of the posts in $\mathcal{S}_{part}$.

(3) To remove noisy tags from $\mathcal{S}_{part}$, we only keep the tags that are significantly relevant to build systems. We use two metrics *significance* and *relevance* from previous work [8, 30] and calculate for each tag t in $\mathcal{T}_{cand}$ with Equations 1 and 2. A tag t is significantly relevant to build systems if its two metrics are higher than specific thresholds. To avoid omitting relevant tags, we adopt the lowest thresholds used in previous work [8] and only the tags whose significance is higher than 0.005 and whose relevance is higher than 0.05 are kept in $\mathcal{T}_{cand}$. At last, the first two authors manually inspect each remaining tag in $\mathcal{T}_{cand}$ and remove the ones irrelevant to build systems; and besides three initial tags, the final tag set $\mathcal{T}_{final}$ consists of extra 15 tags, such as "*build.gradle*", "*gradle-plugin*", "*gradlew*", "*pom.xml*" and "*m2eclipse*".

$$Significance(t) = \frac{|\{p|p \in \mathcal{S}_{part} \wedge p\ with\ t\}|}{|\mathcal{S}_{part}|} \quad (1)$$

$$Relevance(t) = \frac{|\{p|p \in \mathcal{S}_{part} \wedge p\ with\ t\}|}{|\{p|p \in \mathcal{S}_{all} \wedge p\ with\ t\}|} \quad (2)$$

All the posts with at least one tag in $\mathcal{T}_{final}$ are regarded as build-related posts.

Step 3: refine dataset. To ensure the quality of selected posts and to reduce the ambiguity in following analysis, we further filter the selected posts by only keeping the ones with an accepted answer, which is the answer marked as accepted by the questioner. Out of the remaining 25,814 posts, we randomly select a statistically significant sample ensuring a 95% confidence level ± 5%. The final dataset for our study consists of 1,080 posts (i.e., 336 for *Ant*, 370 for *Maven*, and 374 for *Gradle*).

3.2 Manual Labelling

Each post in the dataset is manually classified as *false positive* (i.e., it is unrelated to build issues or its accepted answer is useless according to the questioner's additional comments) or assigned a set of labels describing (i) the *how-to topic*, which is the summarization of the how-to question description, (ii) the *failure symptom*, which shows what the failure looks like according to the question description, and (iii) the *fix pattern*, which tells how a build issue is fixed distilled from the accepted answer. Each of these labels is optional. In particular, if the post is raising a how-to question (e.g., asking help to implement a specific build task or discussing conceptual build knowledge) rather than an encountered build failure, only the *how-to topic* label is necessary for such a *how-to question*; otherwise, for the post with a concrete build failure encountered by the questioner, it is labelled with both the *failure symptom* and the *fix pattern*.

The manually labelling is conducted following an open coding procedure [24] by the fist two authors with 5-year experience of software development. Moreover, we synchronize the label set among participants. In particular, the two authors independently tag each issue assigned to her with an existing label from the current label set or create a new label when the issue cannot be categorized to any existing label. The newly-created label is instantly updated into the label set and can be used by other participants then. Although, in principle, this is against the notion of open coding, little is known about general-type build fixing and the granularity of previous error category is too coarse in our study. Hence, to avoid the number of labels from growing excessively following previous studies [4, 16], we adopt the compromise without introducing substantial bias. In this process, the conflict ratio between two authors is 25.65%.

In the cases where there is a conflict between the two authors, a third arbitrator who is practised in build knowledge with 7 years of build system experience, would be introduced to label the issue. In case of further disagreement among the three participants, the conflicts are discussed and solved among all participants. We follow this rigorous procedure until all issues reach agreement and the final label results are checked by all participants.

Through the preceding process, except the 171 false positive posts, 444 how-to posts are labelled with how-to topics and 465 build failures are labelled with symptoms and fix patterns. To answer RQ1, we analyze 444 how-to posts and distill the frequent topics in Section 4. For RQ2, we construct a taxonomy of build failure symptoms based on the labels. All the authors proceed to group similar codes into categories and create a hierarchical taxonomy of challenges. The grouping process is iterative, in which they continuously go back and forth between categories and questions to refine the taxonomy. The frequent and non-trivial symptom categories in our taxonomy are discussed in Section 5. For RQ3, we study the characteristics of fix patterns for each symptom category in Section 6. For convenience, we list the main findings and implications drawn from this paper (including RQ1, RQ2, and RQ3) in Table 1.

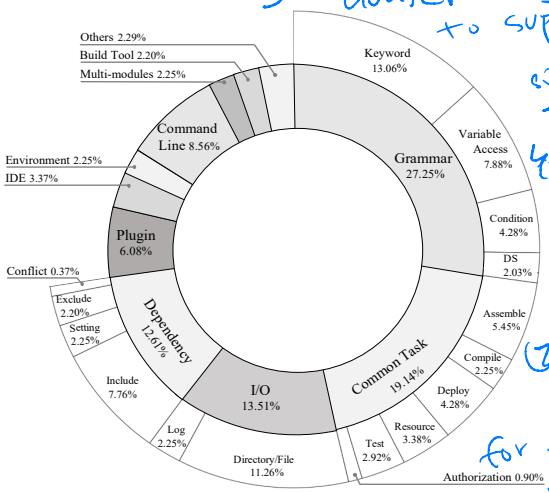

Figure 3: Topics in how-to build issues

4 HOW-TO TOPICS IN BUILD ISSUES (RQ1)

Figure 3 shows the hierarchical how-to topics in build issues with corresponding percentages. According to the figure, we observe a high diversity in how-to topics that developers ask, which can be grouped into 11 high level categories.

Grammar is the topic asked most frequently (27.25%) by developers: specifically, 13.06% of the questions are about the usage of the default keywords in the build script language; 7.88% are about how

to access (i.e., write/read) variables in a build script; 4.28 % of the questions ask for code examples of conditions (e.g., loop/if-else); and the rest are questions on the data structure (e.g., array iteration). Besides, 19.14% how-to questions are about the difficulties that developers encounter when they attempt to write build code to implement *common build tasks* (e.g., compile, test, deploy, or assemble); a non-negligible number (13.51%) of questions are related to file system operations, since resource arrangements occur frequently during the build process; 12.61% are concerned about customizing *dependencies* to the project; 6.08% questions are on the usage of specific third-party plugins; and 8.56% of the questions are about the interoperability between the build system and the command line (e.g., "*how to pass parameters in command line to the build process?*"). The questions vary from naive puzzles (e.g., grammar) to very particular problems (e.g., usage of some specific plugin). It may attribute to the co-existence of novices' and experts' posts on Stack Overflow, both of which are important and noteworthy.

Overall, most (80.84%) questions are about build script programming, indicating that a non-trivial number of developers have difficulties in writing a complete version of build scripts for their projects. Hence, build script generation or completion tools are helpful for developers in programming build scripts, where modern code generation techniques [11, 15, 31] may be adopted since build scripts are often semi-structured and most build tasks can be specified concisely. Furthermore, we observe that developers complain about documents and tutorials in their SO posts (e.g., "*Neither the documentation had any sort of straightforward example...*" [1]), indicating that the completeness, usability, and readability of documents in third-party plugins, dependencies, and build tools should be improved [4]. In addition, to a certain extent, the how-to topics with high frequency imply the build components that developers are unfamiliar with, which are vulnerable and subject to introducing defects in the further build script evolution.

For RQ1, see Findings F.1 and F.2, as well as Implications I.1 and I.2 in Table 1.

5 SYMPTOMS OF BUILD ISSUES (RQ2)

5.1 Taxonomy and Distribution

Figure 4 shows the hierarchical taxonomy of symptoms manually constructed in our study. Nodes are in descending gray-level along with their depth in hierarchy (e.g., leaf nodes are in white). Each leaf node represents a category and its non-white parent node that consists of multiple categories is an *inner-category*. For example, *download error (C.1)* is an inner-category that can be further divided into two categories: *connection error (C.1.1)* and *license error (C.1.2)*. The number in the top right corner refers to the number of issues in that category. In total, our taxonomy consists of 15 inner-categories and 50 categories. The broad scope of build error types indicates the prevalence and the diversity of build failures. We next discuss categories by groups according to their belonging inner-categories. Due to space limit, we only address the frequent and non-trivial categories, and the complete explanations for each category are on our website [3].

Parse (B). It is a build stage where the build system validates the build script prior to executing build tasks, and 15.48% of the build issues occur in the parsing phase. All the previous studies [22, 26] lump parsing issues together as one group, while our taxonomy further classifies them into 7 categories regarding to their error symptoms. *Illegal symbol (B.1)* errors cover 59.72% parsing issues and they are triggered when the build system fails to resolve characters in the build script (e.g., undefined keywords, properties or tasks). Besides such grammar check, the build system also validates the presence of necessary artifacts such as embedded properties and project modules, and interrupts when the property is not initialized (i.e., *missing property (B.4)*), modules are not correctly recognized (i.e., *module resolution error (B.5)*) , or the value of the property is invalid (i.e., *property value resolution error (B.7)*).

Dependency resolution (C). It is an essential step in the build process and responsible for preparing necessary dependencies directly or transitively specified by developers, which covers 19.57% build issues. All the previous studies [27, 34] classify dependency-related issues into one large group and regard them as a big concern in build activities; on this basis, our taxonomy further derives 4 finer categories within dependency-related failures. *Dependency findability error (C.3)*, the most frequent category in the dependency resolution phase, occurs when the required dependencies cannot be found in the specified location (e.g., remote, central, or local repository). *Conflict error (C.2)* is triggered when a project relies on different versions of the same library. Usually developers would not simultaneously use multiple versions of one dependency, but it is difficult for them to be aware of the hidden transitively-dependent relationship among dependencies, which actually causes the conflict issue. These two common error types cover 91.43% dependency resolution issues while the rest are caused by *connection error (C.1.1)* and *license error (C.1.2)* triggered in the unsuccessful dependency downloading process.

Resource processing (E). It covers the build failures triggered in processing resources. Half of the resource processing issues are *acccessibility error (E.1)*, which is triggered when the build system is unable to find the specified resources. Besides, some resources are not originally embedded within the project and need to be generated or merged. Therefore, we classify these issues into categories *generation error (E.2)* and *merge error (E.3)*, respectively. Furthermore, when the build system parses resource files and finds any violation, *format error (E.4)* is triggered.

Compilation (F). It covers a non-negligible part (12.04%) in build issues. A small ratio (16.07%) of compilation issues are reported by the build system. For example, when the build system is unable to load compilers (i.e., *compiler loading error (F.1)*) or unable to load specified files for compilation (i.e., *source file loading error (F.2)*). The rest issues (83.93%) in the compilation phase are errors thrown by the compiler (i.e., *compiler error (F.3)*). For example, *unsupported operator (F.3.3)* (i.e., the current Java version is not compatible with the used operators) and *source code encoding error (F.3.2)* are the errors detected by a compiler when it is parsing the source code. We observe that most errors (82.98%) reported by a compiler are *symbol resolution error (F.3.1)*, which refers to the cases that symbols (e.g., variable, class, method or package) in source code cannot be resolved by the compiler, usually with error messages as `cant.resolve` or `doesnt.exist`. This observation is consistent with the finding in the previous work [32].

Assemble (G). It is a collection of tasks that process the compiled files and package them into distributable or executable formats

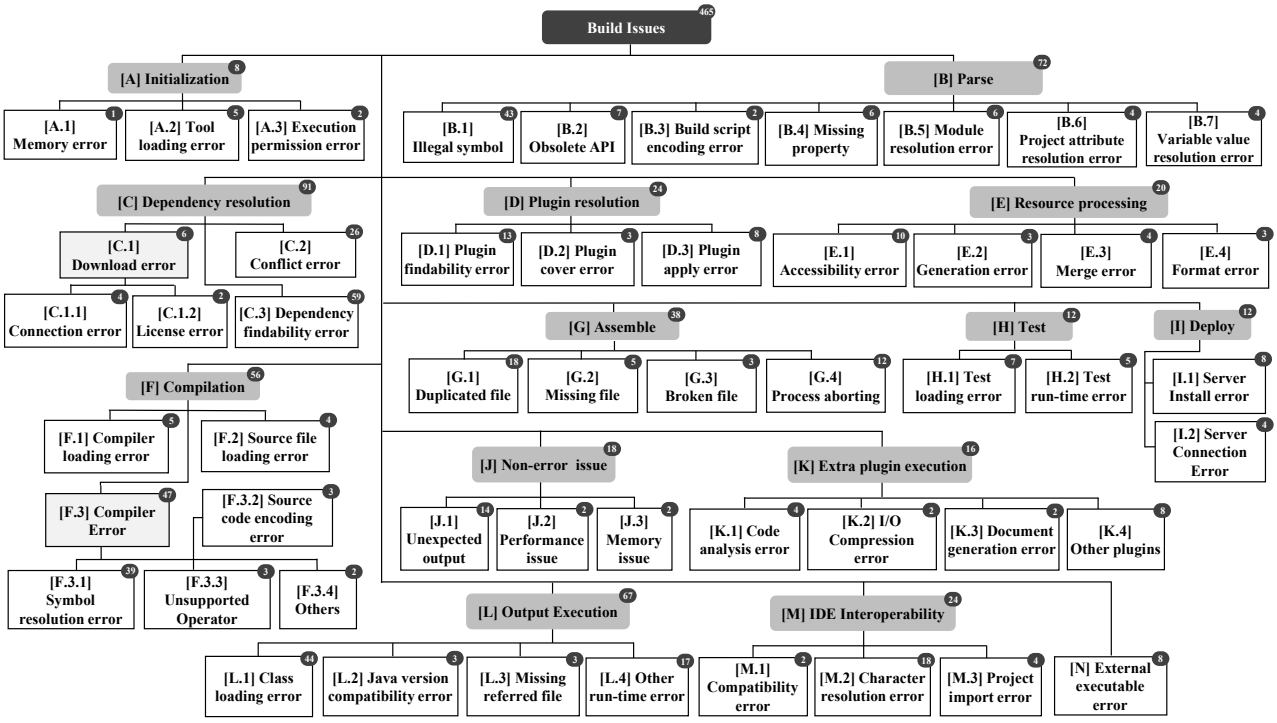

Figure 4: Taxonomy of symptoms in build issues

(e.g., Jar, War, Ear, or Apk). The assemble stage breaks down when there are duplicated, missing, or broken files (*G.1*, *G.2*, and *G.3*, respectively). Besides, 31.58 % assemble issues result in an unexpected exit (i.e., *process aborting (G.4)*), which means that the packaging process interrupts abnormally. For example, the assemble process of Android applications often breaks down when there are excess .dex files.

Non-error issue (J). It is concerned about the build issues with implicit symptoms rather than explicit error message (i.e., "build failure"). For example, *performance issue (J.2)* and *memory issue (J.3)* contain the builds that abnormally occupy too much time or too large memory space. *Unexpected output (J.1)* refers to the cases where the output of certain build task is not as expected although no error is triggered during that build task. For example, the developer assigns Jacoco plugin to collect and report dynamic test coverage during the build process; although the build finishes successfully, the generated report is incomplete with missing coverage results for some tests.

Extra plugin execution (K). Considering the large number of third-party plugins, except the plugins for common build tasks (e.g., resource processing (E), compilation (F), test (H), deploy (I), and assemble (G) as listed in our taxonomy), we group the build failures triggered during the execution of the other plugins into this inner-category. These build failures are further classified into finer categories regarding to their functions. For example, *code analysis (K.1)* covers the build failures triggered by plugins for conducting code analysis (e.g., static code analysis or dynamic coverage collection); *document generation error (K.3)* covers the errors thrown by plugins for generating documents (e.g., Javadoc). Given the diversity of plugins in the wild, instead of enumeration, an extendable category *other plugins (K.4)* is used to contain the rest issues.

Output execution (L). Some build-related issues do not trigger build failures explicitly during the build process, but generate problematic outputs (e.g., compiled class file, assembled jar, or executable application) that fail to work properly. We group such issues as output execution (L). In other words, these issues are actually build failures with delayed exposure. Note that such build failures are often neglected by the previous studies on build failures [26, 27], since their analysis only keeps the build issues with explicit error symptoms during the build process. After inspecting the various run-time exceptions and errors thrown during the output execution, we find that 67.69% of them are related to *class loading error (L.1)*, which consists of two symptoms: `ClassNotFoundException` and `NoClassDefFoundError`. They occur when the Java Virtual Machine is unable to find a particular class at run-time and we group these two symptoms together, because in many cases they are nested with each other and essentially they are both caused by missing classes during run-time. *Class loading error (L.1)* often stems from the missing or incorrect dependencies in the assemble or compilation phases, and more details and examples of their root causes will be discussed in Section 6. Furthermore, when the Java versions in run-time and compilation phases are inconsistent, *Java*

version compatibility error (L.2) is triggered; and inaccessible referred files during run-time induce *missing referred file (L.3)*. Due to their delayed exposure, output execution issues increase developers' efforts in solving build problems.

IDE interoperability (M). The interoperability between IDE and build frameworks may also induce build issues. In this study, we observe such symptoms on most mainstream IDEs, such as Android Studio, Eclipse, and IntelliJ. Most interoperability issues (75.00%) are *character resolution errors (M.2)* when the build script cannot be resolved properly in IDE (e.g., an unresolved character is often addressed by a red underline). *Compatibility error (M.1)* is triggered when IDE is incompatible with the current version of the build system. When IDE fails to import an existing project that is already built with a build framework, IDE reports a *project import error (M.3)*. Interoperability build issues are often occurring and resolved in developers' local work space and therefore they are seldom included in previous studies [22, 29] that are based on historical or reproduced build data.

For RQ2, see Findings F.3 F.4 and Implications I.3 I.4 in Table 1.

5.2 Discussion

5.2.1 Comparison to Previous Studies. Several studies have investigated error types of build failures driven by different research goals. For example, Kerzazi *et al.* [17] interviewed developers in a company to investigate circumstances, under which the build process was broken in the lense of human factors. Given the prevalence of continuous integration (CI), there are emerging studies investigating build breakages in the CI scenario: Vassallo *et al.* [29] compared different error types of CI in open-source and closed-source projects; Zolfagharinia *et al.* [34] investigated environmental impacts on CI error types of CPAN projects. Besides CI, there are studies [13, 26, 27] investigating build issues under the traditional build scenario. For example, Hassan *et al.* [13] inspected 91 build issues to explore the feasibility of automatic build.

Table 2 lists all the studies related to general build issue classification. The column "Dataset attributes" shows project types (closed-source or open-source), build scenarios, and build systems involved in their datasets; the column "Label process" shows how these studies label each build failure in their datasets to derive the categorization. Specifically, "Automated label" means that the study utilizes text processing techniques to automatically extract labels from build failure logs to label each build failure; "Manual label" means that each build failure is labelled via manual inspection. The column "Size" counts the number of categories in the taxonomy and the last column presents the frequent error types in their findings.

Overall, we observe that our taxonomy differs from previous studies in terms of *granularity*, *diversity*, and *distribution*. (i) Our taxonomy is derived at a finer granularity so that the corresponding fix patterns within each symptom category would be more uniform. It helps understand developers' resolution behaviours on different build failures. Note that a fine-granularity taxonomy is not always necessarily practicable in all studies, which is dependent on the research goal and the categorization methodology (automated or manual label). (ii) We observe that our taxonomy covers extra categories that are absent in previous studies. The reason might be that our dataset is mined from build issues posted by developers on SO while previous studies are based on build history data in CI or reproduced data in traditional build scenario. The latter inherently filters out the build issues without explicit failure symptoms during the build process. (iii) Another difference lies in the frequency distribution of the categories in our taxonomy, which is not identical with previous ones. Actually as shown in the column "Frequent categories", almost every study derives different frequent categories. The biggest reason for the inconsistence may stem from the underlying datasets. Different sources, build scenarios and build tools all attribute to the different distribution of frequent failure types. As shown in the table, analyzing build issues from SO, our taxonomy inherently is not limited to open or closed source projects, CI or traditional build scenario. Furthermore, the frequent error types identified by previous studies may be those that occur most frequently in build activities, while the frequent error types identified by our study tend to be the issues hard to solve, since SO posts are often composed by the problems developers cannot resolve at first glance.

5.2.2 Evaluation of the Taxonomy. In the previous section, we illustrate the difference of our taxonomy compared to the previous work. Note that we are not to judge which taxonomy is better, since as aforementioned, all the taxonomies are driven by different research goals. In this work, our goal is to understand build issue resolution in practice. Therefore, we evaluate our taxonomy against the latest taxonomy proposed by Hassan et al. [13] in terms of this goal. We further construct another sample of 150 SO posts from the remaining dataset, categorize them according to our taxonomy and Hassan's, respectively, and then calculate the following measurements as suggested in previous work [21]. The numbers before and after "/" refer to the results on our taxonomy and Hassan's respectively. We find that (i) 10%/49% of issues cannot fit in any category; (ii) 14%/36% categories have no issues; (iii) there are 3.1/7.2 fixing patterns on average in each category. The results demonstrate that our taxonomy has less categorization errors and the smaller number of fix patterns in each category indicates the better performance in facilitating failure resolution.

5.2.3 Different Views of the Taxonomy. Our taxonomy is primarily derived based on the issue-triggering phases and the manifested symptoms. Actually there can be various aspects to analyze build issues (e.g., incremental/full, local/remote, and system/user), and thus we further discuss our taxonomy with different views. For example, build issues can be generally categorized into system errors or user errors. The former refers to the issues associated with incorrect functioning of the build tool itself (e.g., *Ant*), and the latter refers to the issues caused by artifacts outside the build tool (e.g., dependencies or user configuration). Within our taxonomy, we can observe that some categories are purely related to system errors (e.g., all issues in *Tool loading error* can be resolved by the build tool itself), while some categories are the mixture of both errors (e.g., issues in *Illegal symbol* are probably caused by changing the build tool or syntax errors from users). Overall, most build issues are resulting from developers. Moreover, we also find that different

[1]The company adopts one central build server acting as build controller and four build Agents in the build process.

Table 2: Summary of previous studies related to build issue categories

Study	Dataset attributes			Category attributes		Frequent categories
	Source	Build scenario	Build system	Label process	Size	
Kerzazi et al. [17]	Closed	Continuous integration	Industrial system [1]	Interview	3	Missing files; mistaken check-in
Tufano et al. [27]	Open	Traditional scenario	Maven	Automated label	3	Dependency
Sulír et al. [26]	Open	Traditional scenario	Maven; Ant; Gradle	Automated label	12	Dependency; compilation
Vassallo et al. [29]	Open; Closed	Continuous integration	Maven	Automated label	20	Test; release preparation
Rausch et al. [22]	Open	Continuous integration	Maven; Gradle	Automated label	14	Test; code quality; compilation
Zolfagharinia et al. [34]	Open	Continuous integration	CPAN	Manual label	13	Dependency; undefined variable
Hassan et al. [13]	Open	Traditional scenario	Maven; Ant; Gradle	Manual label	14	JDK incompatibility; build tool
This work	Open; Closed	Traditional scenario; continuous integration	Maven; Ant; Gradle	Manual label	50	Dependency; parse; compilation; output execution; assemble

operating platforms may cause different fix patterns. Although we observe only a small number of such cases (i.e., less than five), it is interesting to further investigate the impact from different platforms (e.g., operating systems) on build issue resolution.

6 FIX PATTERNS OF BUILD ISSUES (RQ3)

To capture how developers fix different types of build issues, in this section, for each symptom category, we summarize their fix patterns in Table 3. The columns "Inner-category" and "Category" are consistent with our taxonomy in Figure 4 and the number in parentheses is the number of build issues in that category. The column "Category description" briefly describes the symptom category, the column "Fix pattern description" presents the fix pattern and the last column "#Issues" is the number of issues fixed by that pattern. Take the first row in the table as an example, out of the 8 issues in inner-category *initialization (A)*, 5 issues belong to *tool loading error (A.2)* and all of them are fixed by changing the version of the build tool. Due to space limit, we do not list the patterns with low frequency (i.e., #Issues < 3) and the complete list of all patterns can be found in our website[3].

Overall, we observe complicated crossovers between different fix patterns and symptom categories, confirming that build failure resolution is a challenging problem. Meanwhile, the average number of fix patterns in each category (i.e., 3.08) is much less than in each inner-category (i.e., 7.36), indicating that our taxonomy potentially divides build failures with different causes into different small groups, which supports the necessity and validity of our fine-granularity taxonomy derived in RQ2. We next discuss the main findings and implications as following.

6.1 Prevalent Fix Patterns Across Symptoms

Although the number of symptom categories and fix patterns is large, there are prevalent patterns that can fix multiple symptom categories. We find that 67.96% of the build issues are fixed by repairing the build script code related to the components *plugin* and *dependency*, and these build issues cover 37 out of 50 (74.00%) symptom categories in our taxonomy.

In addition, we use Figure 5 to show the frequency of each fix pattern on each inner-category. X axis represents each inner-category and the letter identifier is consistent with our taxonomy in Figure 4; Y axis shows fix patterns[1] following with their total frequency.

[1]Due to space limit, patterns with total frequency < 5 are not shown in the figure.

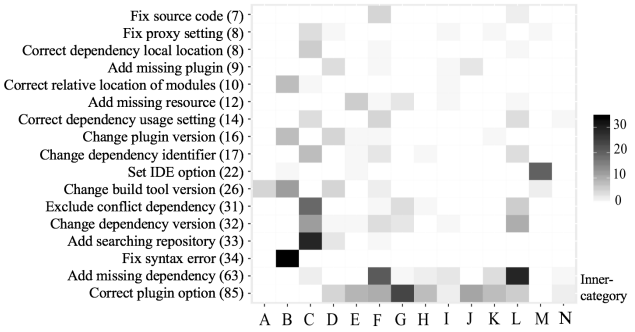

Figure 5: Distribution of fix patterns on inner-categories

The figure shows the overall trend more intuitively but at a price of detailed information loss compared to Table 3. In Figure 5, we can also observe that the patterns related to plugins and dependencies cover a broad scope of inner-categories. In particular, fix patterns related to plugins (e.g., *correcting plugin option* or *adding missing plugin*) cover build issues in 31 categories. For example, *correcting plugin option* fixes 28 categories and 18.27% build issues, which is the most frequently-used fix pattern. Patterns on script code related to dependencies cover 21 symptom categories: *adding missing dependency* covers 12 categories and fixes 13.54% build issues; *correcting dependency version*, *adding searching repository*, and *excluding conflict dependency* fix 7.10%, 7.10%, and 6.67% of build issues, respectively. It implies that defective script code in plugin setting and dependency declaration is a major cause for build issues.

The analysis of prevalent fix patterns across symptoms is summarized by **Finding F.5 and Implication I.5 in Table 1**.

6.2 Frequent Pairs of Fix Patterns and Symptoms

From Table 3, we observe that there exist *frequent pairs* of patterns and symptom categories, which indicates that given a symptom category, most issues in that category can be fixed by a specific pattern.

In Table 3, we find that for 20 symptom categories, more than half of its build issues can be fixed by a specific pattern, and we regard such combination of symptom category and fix pattern as a frequent pair. For example, 30 out of 43 *illegal symbol (B.1)* issues are resolved by *fixing syntax error*; 19 out of 26 *conflict errors (C.2)*

Table 3: Frequent fix patterns of each category

Inner-category	Category	Category description	Fix pattern description	#Issues
[A] Initialization (8)	[A.2] Tool loading error (5)	Unable to launch build tool	Change build tool version	5
[B] Parse (72)	[B.1] Illegal symbol (43)	Keywords or variables (e.g., property, task, or method) cannot be resolved	Fix syntax error	30
			Change build tool version	11
	[B.2] Obsolete API (7)	Deprecated API is in build script	Change plugin version	6
	[B.4] Missing property (6)	Property without default value is not initialized	Correct relative location of modules	3
	[B.5]Module resolution error (6)	Resolution error during configuring modules	Correct relative location of modules	5
	[B.6] Project attribute resolution error (4)	Project attribute (e.g., build type) is invalid	Correct project build type	4
[C] Dependency resolution (91)	[C.1.1] Connection error(4)	Connection timeout due in dependency download	Fix build tool proxy setting	3
	[C.2] Conflict error (26)	Multiple versions of the same dependency are required simultaneously	Exclude conflict dependency	19
			Change dependency version	4
	[C.3] Dependency findability error (59)	Unable to find the required dependency in the specified remote/central/local repository or local location	Add searching repository	29
			Change dependency version	9
			Change dependency identifier	8
			Correct dependency local location	7
			Correct dependency usage setting	3
[D] Plugin resolution (24)	[D.1] Plugin findability error (13)	Unable to find the required plugin in remote/local /central repository	Add plugin declaration	4
			Change plugin version	3
			Add searching repository	3
	[D.2] Plugin cover error (3)	Plugin execution is not specified in build lifecycle	Specify plugin execution phase	3
	[D.3] Plugin apply error (8)	Unable to apply plugin	Change build tool version	5
[E] Resource processing (20)	[E.1] Accessibility error (10)	Unable to find the required resource	Correct plugin$_{assemble}$ option$_{version}$	5
			Add missing resource in project	4
[F] Compilation (56)	[F.3.1] Symbol resolution error (39)	Cannot resolve symbol (e.g., variable, class method or package) that does not exist	Add missing dependency	19
			Fix source code	5
			Change dependency version	4
			Correct dependency usage setting	3
	[F.3.2] Source code encoding error (3)	Encoding format error in parsing source code	Set plugin$_{compile}$ option$_{encoding}$	3
	[F.3.3] Unsupported operator (3)	Compilation compatibility error	Set plugin$_{compile}$ option$_{source/target}$	3
[G] Assemble (38)	[G.1] Duplicated file (18)	Multiple or conflict files are found	Exclude redundant files	9
			Exclude conflict dependency	3
	[G.2] Missing file (5)	Cannot find required files	Set plugin$_{assemble}$ option$_{path}$	3
	[G.4] Process aborting (12)	Package process is interrupted abnormally	Set plugin$_{assemble}$ option$_{enable}$	7
[H] Test (12)	[H.1] Test loading error (7)	Unable to load tests	Set plugin$_{test}$ option$_{classpath}$	4
[L] Output execution (67)	[L.1] Class loading error (44)	Run-time exceptions (e.g., `ClassNotFound` or `NoClassDef`) are triggered during class loading	Add missing dependency	22
			Change dependency version	7
			Exclude conflict dependency	4
			Correct dependency usage setting	3

are fixed by *excluding conflict dependency*; and 22 out of 44 *class loading errors (L.1)* are fixed by *adding missing dependency*. Besides, as shown in Figure 5, the darker cells also confirm that specific fix patterns are clustering in different inner-categories. Therefore, heuristic strategies for build failure resolution can be derived from these frequent pairs.

The analysis of frequent pairs of fix patterns and symptoms is summarized by **Finding F.6 and Implication I.6 in Table 1**.

6.3 Case-by-case Fix Patterns for Symptoms

In spite of the existence of prevalent fix patterns and frequent pairs, many symptom categories (30 out of 50) do not have common fixing strategies and are fixed by case-by-case patterns. Specifically, the number of fix patterns in each category varies from 1 to 10 and the average number is 3.08. For example, there are 10 fix patterns for the symptom *symbol resolution error (F.3.1)*, 10 patterns for *dependency findability error (C.3)* and 8 patterns for *class loading error (L.1)*[2]. Furthermore, 115 build issues are fixed by a pattern different from the other issues that are in its same category. To a certain extent, this observation explains why manually build issue resolution is challenging, which requires developers' comprehensive knowledge and experience in various cases of failure resolution. Additionally, it may also explain why the state-of-the-art automated build resolution techniques are not that effective. Compared to the overall cases, their fixing strategies only cover a tip of the iceberg, and more fixing strategies may be included according to the massive corner fixing cases derived from our results.

The analysis of case-by-case fix patterns is summarized by **Finding F.7 and Implication I.7 in Table 1**.

6.4 Challenges in Simplistic Fix Patterns

According to our statistics, 79.78% build issues are fixed by a few lines of modification on build script code (i.e., within 10 lines). Particularly, the pattern *changing dependency version*, *correcting plugin option* or *correcting plugin version* often involve only one-line modification. Notwithstanding that, there is a knowledge gap for developers to transform fix patterns (template) into concrete fix patches[3].

For example, in *process aborting (G.4)* category, 7 build issues can be fixed by enabling a specific option in the plugin for assemble (i.e.,

[2]Recall that we do not list less frequent patterns due to space limit and the complete pattern list is on our website [3].

[3]A pattern is an abstract template while a patch is concrete and ready-for-use modification.

correcting plugin$_{assemble}$ option$_{enable}$). As shown in the Example (a) (i.e., the #35890257 post on SO), the assemble phase is interrupted due to the abnormal process abortion. According to the developers' discussion in this post, the root cause is the large number of .dex files; and setting `multiDexEnabled` to `True` can prevent the build process from abortion. `multiDexEnabled` is an option in `Android` plugin while most questioners of this failure symptom are unaware of the function of this option. To sum up, a third-party plugin usually has many options, e.g., `Android` plugin has 82 options, which require massive open knowledge for developers to solve plugin-related build issues.

Question Description:
`com.android.build.api.transform.TransformException`
`Process command finished with non-zero exit value 1.`
Symptom: process aborting (G.4)
Fix Pattern: correct plugin option
Patch:
`+ multiDexEnabled true`

Example (a) - 35890257

Therefore, although most patterns are simplistic, they are associated with different third-party libraries and the libraries always keep evolving and updating their features, which is challenging for a manual follow-up. Although build fixing techniques may relieve such manual efforts by embedding as much domain knowledge as possible in advance, considering the timeliness, the fixing strategies have to be dynamically updated to keep stable effectiveness since the external resources (dependencies/plugins) are evolving all the time. For example, the history-based failure fixing technique Hirebuild [14] is less half effective when it is evaluated one year later with the original training set [18] and the timeliness may partly attribute to it.

The challenge in simplistic fix patterns is summarized by **Finding F.8 and Implication I.8 in Table 1.**

6.5 Challenges in Non-intuitive Fix Patterns

We observe that many fix patterns are non-intuitive for the given symptom category and developers often face difficulties in realizing the root causes behind these build issues.

For example, for 7 build issues in *obsolete API (B.2)*, surprisingly they are not caused by the direct usage of the deprecated API; instead, some plugin transitively uses that obsolete API, which is often neglected by developers. In other words, developers apply a third-party plugin, which uses an obsolete API indirectly. In Example (b), a developer encounters an obsolete API warning and confidently claims that the obsolete API is never used by herself in the build script. Actually, the older version of the plugin `com.google.gms:google-services` uses this deprecated API; and upgrading the plugin to a new version can resolve this issue.

Besides, half of the issues in *missing property (B.4)* are not caused by the absence of property declaration but caused by the incorrect relative path among modules. In Example (c), although the error message is reporting *missing property* and the developer is pretty sure that the property is already written in the script code, the root cause is the incorrect relative location of modules. Because there are inherited properties between the parent module and the child

Question Description:
`Configuration 'compile' is obsolete'`
"I tried to look for "compile " in the whole project but no match was found"
Symptom: obsolete API (B.2)
Fix Pattern: change plugin version
Patch:
`- Classpath 'com.google.gms:google-services:3.1.1'`
`+ Classpath 'com.google.gms:google-services:3.2.0'`

Example (b) - 48709870

module, an incorrect relative location declaration can prevent the build system from resolving the value of the inherited property.

Question Description:
`dependencies.dependency.version is missing`
"I check pom but there are the version written"
Symptom: obsolete API (B.2)
Fix Pattern: correct relative location of modules
Patch:
`+ <parent>`
`+ … <relativePath> … </relativePath> …`
`+ </parent>`

Example (c) - 20424245

Furthermore, most issues in *class loading error (L.1)* are also fixed by non-intuitive patterns. In the Example (d), the developer complains that the build process is successful but the generated application does not run successfully by reporting the *class loading error (L.1)*. It is difficult for the developer to learn that this error comes from the building process because *class loading error (L.1)* is a type of build errors with "fake" successful build status but exposed after build (discussed in Section 5). In this example, the root cause is that the dependency library `json` is not included in the compilation phase and thus adding it can resolves the `ClassNotFoundException` issue.

Question Description:
`Java.lang.ClassNotFoundException`
"mvn clean package and everything builds successfully, but when I try to run it, I get error…"
Symptom: class loading error (L.1)
Fix Pattern: add missing dependency
Patch:
`+ <dependency><groupId>org.json</groupId>`
`+ <artifactId>json</artifactId>`
`+ <version>20090211</version></dependency>`

Example (d) - 15951032

The challenge in non-intuitive fix patterns is summarized by **Finding F.9 and Implication I.9 in Table 1.**

7 THREATS TO VALIDITY

A major threat to validity is that we only use Stack Overflow as the data source to study how developers resolve build issues. Although the study is based on a representative sample of SO posts and SO posts have been widely used in previous work [5, 7, 8, 10, 23, 30, 33], there could be build issues that are never discussed on SO. In other words, we cannot guarantee the generalizability of our observations due to the bias induced by single data source. In the further, we plan to extend our study in more data sources (e.g., Github) and build systems to further validate our findings.

Another threat lies in the construction of the tag set which we utilize to extract build-related SO posts. We cannot guarantee the tag set is complete since the thresholds chosen for the metrics *significance* and *relevance* may overlook some tags related to build

Diff platforms have diff patterns.

issues. To mitigate this threat, we adopt the smallest threshold values used in previous work [8] to firstly include as many related tags as possible, and then we refine the tag set by further manual inspection to ensure the precision. In addition, to eliminate the false positive resulting from automatically extracting posts with selected SO tags, the first two authors of this paper manually label these posts.

In addition, the possible subjectiveness introduced during the manual analysis might induce bias in the results. To mitigate this threat, we ensure each data item is labelled by at least two authors with a third arbitrator resolving the conflicts and inspecting all final results.

8 RELATED WORK

As this paper targets at the understanding of build-issue resolution, it is very related to both the work on build failures and build fixing.

8.1 Build Failures

Build failures have been extensively studied in recent years. Kerzazi *et al.* [17] interviewed developers to study why build breakages are introduced in an industrial case. Tufano *et al.* [27] investigated build errors in *Maven* projects and Sulír *et al.* [26] studied error categories in open source Java projects built with *Maven*, *Ant* and *Gradle* . Hassan *et al.* [13] inspected 91 build issues to explore the feasibility of automatic build. With the prevalence of continuous integration (CI), there are emerging studies investigating build breakages in CI scenario. Rausch *et al.* [22] studied the factors affecting CI build errors in open-source Java projects. Vassallo *et al.* [29] compared the build errors in open-source CI and closed-source CI. Zolfagharinia *et al.* [34] investigated the environmental impacts (REs and OSes) on CI build failures in CPAN projects. Besides general build failures, several studies focused on specific type of build failures. Seo *et al.* [25] performed a large scale study on compiler errors in build process at Google; Zhang *et al.* [32] further investigated compiler errors on open-source Java projects in continuous integration; Beller *et al.* [9] investigated build breakages related to testing on TRAVIS CI; Ghaleb *et al.* [12] studied the build issues with long duration.

Different from existing studies, we conduct the first comprehensive study on build failures by mainly focusing on fix patterns; meanwhile, to characterize patterns precisely, a fine-granularity taxonomy of build failure symptom is derived in our study and the difference with previous studies is discussed detailedly in Section 5.2.1. Furthermore, in this work, we also summarize frequent topics developers ask frequently about build issues, indicating the challenges developers encounter in build activities.

8.2 Build Fixing

The prevalence of build failures has inspired emerging studies on automated build fixing techniques. Macho *et al.* [19] designed three fixing strategies for only dependency-related *Maven* build failures. Zhang *et al.* [32] studied compiler failures in CI and summarized fix patterns particularly for frequent compiler errors. Different from these studies on the specific type of build failures, our work investigates fix patterns across general types of build failures.

Al-Kofahi *et al.* [6] proposed a fault localization approach for Makefile by calculating statement suspiciousness based on their dynamic execution trace. Vassallo *et al.* [28] proposed a supporting tool, summarizing *Maven* build logs and providing relevant online links to reduce developers' resolution efforts. Hassan and Wang [14] fixed general build failures via analysis on build history data. Recently, Lou *et al.* [18] proposed a history-oblivious fixing technique for general build failures via code analysis and search-based patch generation. Although promising, the effectiveness of the state-of-the-art fixing techniques is still far from satisfactory (i.e., the latest technique successfully fixed only 18% build failures [18]), indicating a gap between automated techniques and practical build failure resolution. To bridge such a gap, in this work, we manually inspect a large number of practical human fixing cases and summarize fix patterns for general-type build failures.

9 CONCLUSION

In this work, we present a comprehensive study of build-issue resolution by manually inspecting 1,080 build issues from Stack Overflow. We distill frequent topics in developers' how-to questions and find that most (80.84%) of them are about script code programming. Through the study, we also construct a fine-granularity taxonomy of 50 failure symptom categories and summarize fix patterns for different failure types. We find that build issues stretch over a wide spectrum of symptoms; and prevalent fix patterns and frequent combinations of failure symptoms and fix patterns can be adopted to facilitate build failure resolution. Furthermore, we discuss the challenges in both simplistic and non-intuitive fix patterns.

ACKNOWLEDGEMENTS

This work was partially supported by the National Key Research and Development Program of China under Grant No. 2017YFB1001803 and the National Natural Science Foundation of China under Grant Nos. 61872008 and 61861130363.

REFERENCES

[1] [n.d.]. "How do I execute a program using Maven?". https://stackoverflow.com/questions/2472376/how-do-i-execute-a-program-using-maven Accessed Jan-2020.
[2] [n.d.]. "Stack Exchange Data Dump". https://archive.org/details/stackexchange Accessed Dec-2019.
[3] [n.d.]. 'Supplementary material'. https://sites.google.com/view/buildissue2020/.
[4] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software documentation issues unveiled. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1199–1210. https://doi.org/10.1109/ICSE.2019.00122
[5] Syed Ahmed and Mehdi Bagherzadeh. 2018. What do concurrency developers ask about?: a large-scale study using stack overflow. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*, Markku Oivo, Daniel Méndez Fernández, and Audris Mockus (Eds.). ACM, 30:1–30:10. https://doi.org/10.1145/3239235.3239524
[6] Jafar M. Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2014. Fault localization for build code errors in makefiles. In *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 600–601. https://doi.org/10.1145/2591062.2591135
[7] Moayad Alshangiti, Hitesh Sapkota, Pradeep K. Murukannaiah, Xumin Liu, and Qi Yu. 2019. Why is Developing Machine Learning Applications Challenging? A Study on Stack Overflow Posts. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2019, Porto de Galinhas, Recife, Brazil, September 19-20, 2019*. IEEE, 1–11. https://doi.org/10.1109/ESEM.2019.8870187

[8] Mehdi Bagherzadeh and Raffi Khatchadourian. 2019. Going big: a large-scale study on what big data developers ask. 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*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 432–442. https://doi.org/10.1145/3338906.3338939

[9] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: an explorative analysis of Travis CI with GitHub. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, Jesús M. González-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, 356–367. https://doi.org/10.1109/MSR.2017.62

[10] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, Tao Xie, and Xuanzhe Liu. 2020. A comprehensive study on challenges in deploying deep learning based software. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2020*.

[11] Li Dong and Mirella Lapata. 2016. Language to Logical Form with Neural Attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics. https://doi.org/10.18653/v1/p16-1004

[12] Taher Ahmed Ghaleb, Daniel Alencar da Costa, and Ying Zou. 2019. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering* 24, 4 (2019), 2102–2139. https://doi.org/10.1007/s10664-019-09695-9

[13] Foyzul Hassan, Shaikh Mostafa, Edmund S. L. Lam, and Xiaoyin Wang. 2017. Automatic Building of Java Projects in Software Repositories: A Study on Feasibility and Challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, Ayse Bener, Burak Turhan, and Stefan Biffl (Eds.). IEEE Computer Society, 38–47. https://doi.org/10.1109/ESEM.2017.11

[14] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: an automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1078–1089. https://doi.org/10.1145/3180155.3180181

[15] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-Based Neural Code Generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, 925–930. https://doi.org/10.18653/v1/d18-1111

[16] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *In Proceedings of the 41st International Conference on Software Engineering, ICSE 2020*.

[17] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. 2014. 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*. IEEE Computer Society, 41–50. https://doi.org/10.1109/ICSME.2014.26

[18] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-driven build failure fixing: how far are we?. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 43–54. https://doi.org/10.1145/3293882.3330578

[19] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 106–117. https://doi.org/10.1109/SANER.2018.8330201

[20] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. 2012. The evolution of Java build systems. *Empirical Software Engineering* 17, 4-5 (2012), 578–608. https://doi.org/10.1007/s10664-011-9169-5

[21] Paul Ralph. 2019. Toward Methodological Guidelines for Process Theories and Taxonomies in Software Engineering. *IEEE Trans. Software Eng.* 45, 7 (2019), 712–735. https://doi.org/10.1109/TSE.2018.2796554

[22] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, Jesús M. González-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, 345–355. https://doi.org/10.1109/MSR.2017.54

[23] Christoffer Rosen and Emad Shihab. 2016. What are mobile developers asking about? A large scale study using stack overflow. *Empirical Software Engineering* 21, 3 (2016), 1192–1223. https://doi.org/10.1007/s10664-015-9379-3

[24] Carolyn B. Seaman. 1999. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Trans. Software Eng.* 25, 4 (1999), 557–572. https://doi.org/10.1109/32.799955

[25] Hyunmin Seo, Caitlin Sadowski, Sebastian G. Elbaum, Edward Aftandilian, and Robert W. Bowdidge. 2014. Programmers' build errors: a case study (at google). In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 724–734. https://doi.org/10.1145/2568225.2568255

[26] Matúš Sulír and Jaroslav Porubän. 2016. A quantitative study of Java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. 17–25.

[27] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process* 29, 4 (2017). https://doi.org/10.1002/smr.1838

[28] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. 2018. Un-break my build: assisting developers with build repair hints. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, Foutse Khomh, Chanchal K. Roy, and Janet Siegmund (Eds.). ACM, 41–51. https://doi.org/10.1145/3196321.3196350

[29] 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 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 183–193. https://doi.org/10.1109/ICSME.2017.67

[30] Xinli Yang, David Lo, Xin Xia, Zhiyuan Wan, and Jian-Ling Sun. 2016. What Security Questions Do Developers Ask? A Large-Scale Study of Stack Overflow Posts. *J. Comput. Sci. Technol.* 31, 5 (2016), 910–924. https://doi.org/10.1007/s11390-016-1672-0

[31] Luke S. Zettlemoyer and Michael Collins. 2007. Online Learning of Relaxed CCG Grammars for Parsing to Logical Form. In *EMNLP-CoNLL 2007, Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, June 28-30, 2007, Prague, Czech Republic*, Jason Eisner (Ed.). ACL, 678–687. https://www.aclweb.org/anthology/D07-1071/

[32] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. 2019. A large-scale empirical study of compiler errors in continuous integration. In *Proceedings of the 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*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 176–187. https://doi.org/10.1145/3338906.3338917

[33] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019*, Katinka Wolter, Ina Schieferdecker, Barbara Gallina, Michel Cukier, Roberto Natella, Naghmeh Ivaki, and Nuno Laranjeiro (Eds.). IEEE, 104–115. https://doi.org/10.1109/ISSRE.2019.00020

[34] Mahdis Zolfagharinia, Bram Adams, and Yann-Gaël Guéhéneuc. 2019. A study of build inflation in 30 million CPAN builds on 13 Perl versions and 10 operating systems. *Empirical Software Engineering* 24, 6 (2019), 3933–3971. https://doi.org/10.1007/s10664-019-09709-6