# ADDRESSING INCONSISTENCIES ACROSS SOFTWARE ENVIRONMENTS

by

SUNZHOU HUANG, M.Sc.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

COMMITTEE MEMBERS:
Xiaoyin Wang, Ph.D., Chair
Jianwei Niu, Ph.D.
Wei Wang, Ph.D.
Mitra Bokaei Hosseini, Ph.D.
Na Meng, Ph.D.

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
May 2025

Copyright 2025 Sunzhou Huang
All rights reserved.

# DEDICATION

*To my wife, my daughter, and my family*

## ACKNOWLEDGEMENTS

I would like to express my deepest appreciation and sincere gratitude to my advisor, Dr. Xiaoyin Wang, for his invaluable advice and continuous support throughout my PhD studies and research. His patience, motivation, and enthusiasm have been a constant source of inspiration, and his insights and knowledge have significantly enriched my work.

I also extend my heartfelt thanks to my committee members, Dr. Jianwei Niu, Dr. Wei Wang, Dr. Mitra Bokaei Hosseini, Dr. Na Meng, and Dr. Rocky Slavin, for their insightful comments and helpful suggestions on my research.

I am grateful to my lab mates and friends for their various forms of help and support.

My deepest thanks go to my wife, my daughter, and my family for their unwavering support and love throughout my PhD journey.

This research was supported in part by the National Science Foundation under grants CCF-1846467, CCF-2007718, and CSPECC-1736209, and the Google Cloud Research Credits program with the award GCP19980904.

*This Doctoral Dissertation was produced in accordance with guidelines which permit the inclusion as part of the Doctoral Dissertation the text of an original paper, or papers, submitted for publication. The Doctoral Dissertation must still conform to all other requirements explained in the Guide for the Preparation of a Doctoral Dissertation at The University of Texas at San Antonio. It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.*

*It is acceptable for this Doctoral Dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts, which provide logical bridges between different manuscripts, are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the students contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the Doctoral Dissertation attest to the accuracy of this statement.*

May 2025

**ADDRESSING INCONSISTENCIES ACROSS SOFTWARE ENVIRONMENTS**

Sunzhou Huang, Ph.D.
The University of Texas at San Antonio, 2025

Supervising Professor: Xiaoyin Wang, Ph.D.

Software environments are vital to modern software development, offering tools and frameworks that enhance productivity, reduce costs, and improve software quality. Consistency across different environments is crucial for reliable and efficient builds, tests, and executions. Variations in hardware, operating systems, software dependencies, and configurations can lead to significant challenges and inefficiencies. Although approaches like containerization, virtualization, continuous integration/continuous deployment (CI/CD), and infrastructure as code (IaC) strive to make environments more reproducible and consistent, issues related to environments are still common, particularly for those with additional restrictions or limited domain knowledge.

This dissertation addresses two significant research problems related to inconsistencies across software environments. Firstly, we introduce PExReport, a novel framework designed to extract cross-project failures (CPFs) along with their execution environment. PExReport automates the generation of stand-alone executable CPF reports by leveraging build systems to prune source code and dependencies, creating a pruned environment that accurately reproduces CPFs. Demonstrated within the Maven build system as PExReport-Maven, our framework achieved a high reproduction rate of 92.93%, successfully reproducing 184 out of 198 CPFs with exact failure types and messages. Additionally, PExReport-Maven significantly reduced the required source classes and dependencies, enhancing the efficiency of CPF reproduction. Secondly, we explore the challenges faced by non-contributors, specifically Computer Science students, in handling build issues in open-source software (OSS). Our pilot study collected data from command history logs, environment variables, network information, and virtual machine snapshots, revealing that non-contributors often struggle with certain build issues. Furthermore, a dual-phase study involving 330 build tasks among 55 students characterized non-contributors' build issues and resolution at-

tempts. An intervention in Phase II successfully improved the build success rate by proactively addressing difficult “trap” issues.

Overall, this dissertation introduces the PExReport, which addresses environmental inconsistencies in CPF reproduction. Additionally, it presents comprehensive studies aimed at resolving build environment inconsistencies from the perspective of non-contributors. This dissertation offers tools and methodologies to mitigate these inconsistencies across various software environments and identifies potential directions for future research.

**TABLE OF CONTENTS**

**Acknowledgements** . . . . . . . . . . iv

**Abstract** . . . . . . . . . . v

**List of Tables** . . . . . . . . . . xii

**List of Figures** . . . . . . . . . . xiii

**Chapter 1: Introduction** . . . . . . . . . . 1

1.1 Motivation . . . . . . . . . . 1

1.2 Problem Statement . . . . . . . . . . 2

1.2.1 Extracting Failures Along with Their Execution Environment . . . . . . . 2

1.2.2 Addressing Inconsistencies in the Build Environment from the Perspective of Non-Contributors . . . . . . . . . . 3

1.3 Contributions . . . . . . . . . . 3

1.3.1 To address the challenge of extracting failures along with their execution environment . . . . . . . . . . 3

1.3.2 To address inconsistencies in the build environment from the perspective of non-contributors . . . . . . . . . . 4

1.4 Organization . . . . . . . . . . 5

1.5 List of Publications . . . . . . . . . . 5

**Chapter 2: PExReport: Automatic Creation of Pruned Executable Cross-Project Failure Reports** . . . . . . . . . . 7

2.1 Introduction . . . . . . . . . . 7

2.2 Motivation . . . . . . . . . . 9

2.2.1 Characteristics of Ideal CPF Reports . . . . . . . . . . 9

2.2.2 Cross-Project Failure Report Trilemma . . . . . . . . . . . . . . . . . . . . 11
2.3 PExReport . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.3.1 Principles of Design . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.3.2 Prevalent Tasks for the Lifecycle of CPFs . . . . . . . . . . . . . . . . . . 15
2.3.3 Base Approach: Hybrid Backward Failure Tracing . . . . . . . . . . . . . . 16
2.3.4 Three Enhancements . . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.3.5 Automatic Creation of CPF Reports . . . . . . . . . . . . . . . . . . . . 23
2.3.6 Failure Report Validation . . . . . . . . . . . . . . . . . . . . . . . . 24
2.4 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.4.1 Experiment Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.4.2 Metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.4.3 Dataset Construction . . . . . . . . . . . . . . . . . . . . . . . . . . 26
2.4.4 Evaluation Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
2.4.5 Threats to Validity . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
2.5 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
2.6 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
2.7 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
2.8 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

**Chapter 3: PExReport-Maven: Creating Pruned Executable Cross-Project Failure Reports in the Maven Build System . . . . . . . . . . . . . . . . . . . . . . . . . . 37**
3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
3.2 PExReport-Maven Description . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.2.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.2.2 Core Maven Phases for the Lifecycle of Test Failures . . . . . . . . . . . . 40
3.2.3 Hybrid Backward Failure Tracing . . . . . . . . . . . . . . . . . . . . . 41
3.2.4 Enhancement Components . . . . . . . . . . . . . . . . . . . . . . . . 43
3.3 Envisioned Users . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

3.4 Using the Tool . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
3.4.1 Download . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
3.4.2 Create a Working Environment . . . . . . . . . . . . . . . . . . . . 45
3.4.3 Command Line Usage . . . . . . . . . . . . . . . . . . . . . . . 46
3.5 Evaluation and Studies . . . . . . . . . . . . . . . . . . . . . . . . . 46
3.6 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

**Chapter 4: Build Issue Resolution from the Perspective of Non-Contributors . . . . . . 48**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.2 Related Works . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.3 Study Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
4.3.1 Participants and Tasks . . . . . . . . . . . . . . . . . . . . . . . 51
4.3.2 Data Collection . . . . . . . . . . . . . . . . . . . . . . . . . . 53
4.4 Results and Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.5 Discussion and Future Work . . . . . . . . . . . . . . . . . . . . . . . 58
4.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

**Chapter 5: An Exploratory Study on Build Issue Resolution Among Computer Science Students . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.2 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
5.2.1 Resolution Strategy Study . . . . . . . . . . . . . . . . . . . . . . 61
5.2.2 Intervention Study . . . . . . . . . . . . . . . . . . . . . . . . . 62
5.2.3 Participants and Tasks . . . . . . . . . . . . . . . . . . . . . . . 62
5.2.4 Data Collection . . . . . . . . . . . . . . . . . . . . . . . . . . 64
5.2.5 Qualitative and Quantitative Data Analysis . . . . . . . . . . . . . . . 67
5.3 Resolution Strategies Study . . . . . . . . . . . . . . . . . . . . . . . 68

5.3.1 As learners of OSS, are students able to accurately interpret the build issues that block build tasks? (RQ1) . . . 69
5.3.2 How often do build tasks end with unresolved issues? (RQ2) . . . 70
5.3.3 What are the common symptoms of unresolved build issues, and to what degree can they be mitigated? (RQ3) . . . 71
5.3.4 What strategies have students employed to successfully address these unresolved build issues, and are these strategies effective? (RQ4) . . . 73
5.3.5 Are there any prior experiences that may correlate with students' ability to resolve build issues? (RQ5) . . . 74
5.4 Intervention Study . . . 75
5.4.1 Given the resolution strategies employed by students, what proactive measures can we introduce to effectively enhance their performance in resolving build issues? (RQ6) . . . 77
5.5 LESSONS LEARNED . . . 80
5.5.1 Educators . . . 80
5.5.2 Researchers . . . 81
5.5.3 Students . . . 81
5.5.4 OSS contributors . . . 82
5.5.5 Generative AI . . . 83
5.6 Threats to Validity . . . 83
5.6.1 Internal Validity . . . 83
5.6.2 External Validity . . . 84
5.7 Related Work . . . 84
5.8 Conclusion . . . 86
5.9 Data Availability . . . 87
**Chapter 6: Conclusion and Future Directions . . . 88**
6.1 Dissertation Summary . . . 88

6.2 Future Research Directions . . . . . . . . . . 89

**Bibliography** . . . . . . . . . . **91**

**Vita**

## LIST OF TABLES


2.1 Statistics of the collected issues. . . . . 28

2.2 Reproduced CPFs . . . . . 29

2.3 Statistics of reduction rate . . . . . 30

4.1 Selected OSS projects . . . . . 52

5.1 OSS projects in resolution strategies study . . . . . 68

5.2 Success rate of verified build outcomes . . . . . 70

5.3 Errors, symptoms and resolutions . . . . . 72

5.4 Build outcomes with intervention . . . . . 78

## LIST OF FIGURES


2.1 Cross-project failure report trilemma . . . . . . . . . . . . . . . . . . . . . . 12

2.2 Overview of the PExReport workflow . . . . . . . . . . . . . . . . . . . . . . 13

2.3 Exemplar three tasks build process of Java project . . . . . . . . . . . . . . . 16

2.4 Exemplar resource directory structure . . . . . . . . . . . . . . . . . . . . . . 21

2.5 Top 10 failure types in representative CPFs . . . . . . . . . . . . . . . . . . . 28

2.6 Cumulative step graph of reduction rates . . . . . . . . . . . . . . . . . . . . . 31

2.7 Comparison of different PExReport techniques . . . . . . . . . . . . . . . . . 32

3.1 Cross-project failure report trilemma . . . . . . . . . . . . . . . . . . . . . . 38

3.2 PExReport-Maven workflow . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

3.3 Maven test failure example . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

4.1 Symptoms experienced by non-contributors. . . . . . . . . . . . . . . . . . . . 55

4.2 Distribution of categories on PLs. . . . . . . . . . . . . . . . . . . . . . . . . 57

5.1 Data collection framework for build tasks . . . . . . . . . . . . . . . . . . . . 64

5.2 Root symptoms from resolution strategies study . . . . . . . . . . . . . . . . . 71

5.3 Distribution of prior experiences . . . . . . . . . . . . . . . . . . . . . . . . . 75

5.4 Spearman's rank correlation of prior experiences on build success . . . . . 76

5.5 Helpfulness rating for intervention (diamond markers indicates means) . . . 78

# CHAPTER 1: INTRODUCTION

## 1.1 Motivation

Software environments are essential to modern software development, offering the necessary tools and frameworks to boost productivity, reduce costs, and improve software quality. These environments range from comprehensive engineering platforms to specialized programming settings, each tailored to specific development needs and challenges. For instance, effective development environments are crucial in agile and distributed settings, where prompt feedback and efficient collaboration are necessary to maintain productivity and software quality [23]. Additionally, modern Integrated Development Environments like Microsoft Visual Studio and Eclipse have significantly advanced from legacy environments, enhancing programmer productivity despite the complexity they introduce [95]. Furthermore, the evolution of Integrated Development Environment into Intelligent Development Environments is transforming the role of human programmers, allowing them to manage AI agents and automated tools to implement software features more efficiently [57].

Ensuring consistency across various environments is crucial for maintaining reliable and efficient builds, tests, and executions. Differences in hardware, operating systems, software dependencies, and configurations can introduce significant challenges and inefficiencies. These inconsistencies often result in complex, error-prone activities that are not only time-consuming but also difficult to troubleshoot and resolve. The Vitruvius approach highlights the issues of fragmentation and redundancy in system descriptions when multiple modeling languages are used, which can lead to inconsistencies that negatively impact system quality and are costly to fix [48]. Similarly, the concept of megamodel consistency in software build systems addresses the need for sound and optimal building processes that maintain consistency across various models and transformations, minimizing unnecessary recomputation [79].

Managing consistency across software environments is a well-recognized challenge. To address this, developers employ several approaches: Containerization with tools like Docker to create containers that encapsulate the application and its dependencies, ensuring it runs consistently

across different environments [18]; Virtualization, where virtual machines replicate the production environment on different hardware, providing a consistent platform for development and testing [72]; Configuration Management using tools like Ansible, Puppet, and Chef to automate the setup and maintenance of environments, ensuring configurations remain consistent [41]; Continuous Integration/Continuous Deployment (CI/CD) pipelines to automatically test and deploy code changes in a consistent manner, reducing the risk of environment-related issues [76]; Version Control to track changes in code, dependencies, and configurations through systems like Git, maintaining consistency and allowing easy rollback if issues arise [20]; and Environment as Code, where defining environments using code (e.g., Terraform for infrastructure as code) ensures environments can be recreated reliably and consistently [21]. By implementing these approaches, developers can minimize the risks associated with environmental inconsistencies, leading to more reliable and efficient development processes.

While existing approaches aim to make environments more reproducible and consistent, there is no silver bullet. Environment-related issues are still frequently encountered, especially by those who face additional restrictions or lack domain knowledge. Constraints such as limited resources, time pressures, or organizational policies can complicate the process, making it challenging to achieve the desired level of consistency. Additionally, developers who are new to certain tools or technologies may struggle to implement these approaches effectively, leading to inconsistencies and unexpected behavior.

## 1.2 Problem Statement

This dissertation addresses two key research problems related to inconsistencies across software environments:

### 1.2.1 Extracting Failures Along with Their Execution Environment

Modern software development heavily relies on libraries created by various developer teams, whether within the same organization or across different ones. When software is executed, its

execution trace can traverse multiple software products, leading to cross-project failures (CPFs). The distributed nature of modern software development and the constraints in sharing software environments make reporting CPFs particularly challenging. A previous study [19] indicates that the most effective way to communicate these issues is through a stand-alone executable failure report. However, generating such a report is often difficult due to the intricate interactions between files and dependencies within software ecosystems.

### 1.2.2 Addressing Inconsistencies in the Build Environment from the Perspective of Non-Contributors

Open-source software (OSS) often needs to be built by individuals who are not contributors. For example, Computer Science (CS) students frequently attempt to use or extend OSS projects, only to face the common challenge of build failures on their local machines. Although OSS typically offers ready-to-build packages, subtle differences in local environment setups can lead to build issues, requiring students to spend significant time and effort debugging. Despite the widespread occurrence of these build issues among non-contributors, there is a notable lack of studies addressing this topic.

## 1.3 Contributions

This dissertation makes the following contributions to addressing inconsistencies across software environments.

### 1.3.1 To address the challenge of extracting failures along with their execution environment

- We present a novel framework, PExReport, designed to extract both code and build dependencies and create pruned executable CPF reports.
- We developed three technical enhancements for PExReport to handle build configurations, resource files, and generated source code during automatic creation.

- We demonstrate PExReport within the Maven build system as PExReport-Maven. An evaluation on 198 representative CPFs from 74 software project issues shows that PExReport-Maven successfully reproduced 184 out of 198 CPFs with exact failure types and messages, achieving a high reproduction rate of 92.93%.

- PExReport-Maven achieved an average reduction rate of 55.37% on required source classes. It also outperformed in handling internal classes, source + internal classes, build configurations, and resources, with reduction rates of 75.94%, 72.97%, 82.96%, and 74.18%, respectively.

### 1.3.2 To address inconsistencies in the build environment from the perspective of non-contributors

- We explored how non-contributors handle build issues in OSS by collecting data from command history logs, environment variables, network information, and snapshots of virtual machine states.

- We investigated 303 build issues experienced by 31 non-contributors (CS students) and found that the build issues they struggle with often display only limited symptoms.

- We observed additional symptoms not covered by previous studies: command misuse and remnant conflicts.

- We conducted a novel dual-phase study involving 330 build tasks among 55 CS students, revealing that intuitive resolution strategies often led to "trap" issues, which are especially difficult to resolve.

- Based on our findings, we introduced an intervention method that emphasized key information (e.g., recommended programming language versions) to CS students, which significantly improved the success rate of build outcomes.

## 1.4 Organization

This dissertation is organized as follows:

- Chapter 2 presents PExReport: Automatic Creation of Pruned Executable Cross-Project Failure Reports, a novel framework designed to extract failures along with their execution environment.
- Chapter 3 demonstrates PExReport within the Maven build system as PExReport-Maven, which can create concise cross-project failure reproduction packages by trimming down source code, dependencies, and the Maven build environment.
- Chapter 4 presents a pilot study exploring how non-contributors handle build issues in open-source software.
- Chapter 5 introduces a dual-phase exploratory study investigating the build issues faced and resolved by computer science students (non-contributors). It also introduces an intervention to help students build open-source software.
- Chapter 6 concludes this dissertation and outlines future research directions.

## 1.5 List of Publications

1. An Exploratory Study on Build Issue Resolution Among Computer Science Students
   **Sunzhou Huang**, Na Meng, Xueqing Liu, Xiaoyin Wang
   In Proceedings of the 37th IEEE Conference on Software Engineering Education and Training (CSEE&T), 2025
   **ACM SIGSOFT Distinguished Paper Award**
   In 2025, CSEE&T replaced ICSE's traditional education track.
2. Build Issue Resolution from the Perspective of Non-Contributors
   **Sunzhou Huang**, Xiaoyin Wang

In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE NIER), 2024

3. PExReport-Maven: Creating Pruned Executable Cross-Project Failure Reports in Maven Build System

   **Sunzhou Huang**, Xiaoyin Wang

   In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA DEMO), 2023.

4. PExReport: Automatic Creation of Pruned Executable Cross-Project Failure Reports

   **Sunzhou Huang**, Xiaoyin Wang

   In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023.

# CHAPTER 2: PEXREPORT: AUTOMATIC CREATION OF PRUNED EXECUTABLE CROSS-PROJECT FAILURE REPORTS

## 2.1 Introduction

With the assistance of third-party infrastructures and functional components, software ecosystems have expanded tremendously over the past several decades. The prevalent usage of third-party software libraries has significantly reduced the cost of software development and improved the quality of software. As the size and complexity of software products increase, developers introduce more and more software dependencies via software libraries. However, the intricate dependencies among different software projects raise new maintenance challenges. When a developer executes the software, the execution trace often goes across the boundaries of multiple software products. So, for some execution failures, it may not be easy to determine which software project should take responsibility for and fix its code. In this chapter, we refer to such failures as *cross-project failures (CPFs)*. For example, when a developer upgrades a software library and encounters a software failure, it may be caused by either a backward incompatibility bug in the software library or a non-robust usage of the library API.

Resolving CPFs usually requires the cooperation and negotiation of more than one project as opposed to intra-project failures (IPFs), so the failure needs to be reported from its observing software project to other projects. However, creating a good failure report is not trivial. A previous study on desirable bug reports [19] shows that a stand-alone executable test case is the most desirable information to be included in reports. A textual failure report from client developers may not be very helpful for the library developers to investigate or reproduce the failure. Since CPFs are typically triggered at the interface between the client code and the third-party software libraries, a thorough explanation of such issues in a failure report at least needs to involve some client-side software artifacts. However, in some situations, even if the client developers provide some code, the failure can also be hard to reproduce. For example, in the comment section of an Apache

Issue (Issue 2497 in Apache TIKA project) [14], an Apache TIKA developer complained that "I'm not able to reproduce it w/in Solr in the unit tests with your file.". If the failure report contains a test case that can be compiled and executed in a stand-alone way, it would greatly simplify the diagnosis process.

An ideal failure report should satisfy three essential requirements: Executability, Readability, and Conciseness. Our pilot research reveals the existence of a trilemma when client developers try to create CPF reports using existing techniques. In particular, static and dynamic slicing techniques [13, 90], can both prune source code based on a given *seed code*, but they do not take into account the build process or the execution dependencies, so the generated slices may not be compiled or executed as a stand-alone project. Software debloating [22] is a practical technique for pruning software releases, but it focuses on destination code and execution-time dependencies instead of source code and compilation-time dependencies, so applying it to source code will cause the pruned code to be uncompilable. Finally, packaging the entire client software and sending it together as the CPF report is typically not a realistic solution, considering the numerous redundant dependencies that may (1) significantly increase the size of the report; (2) bring in lots of noise to the debugging process; (3) unintentionally leak internal information and proprietary code from other parties.

In this chapter, to achieve all three requirements, we developed the PExReport, a framework for generating executable pruned CPF reports. Given a CPF, PExReport performs a three-step analysis to trace the source code and object code (e.g., Java bytecode), which were loaded at compilation and run time. In the build process, it gradually identifies all the required source code and dependencies for compiling the tests and reproduces the CPFs. Besides source code and dependencies, PExReport further identifies the required portion of build configuration, resource files, and generated source code. Finally, PExReport extracts all identified required files from the original build environment and reconstructs a stand-alone project that can compile and reproduce the CPF.

We implemented PExReport for Maven [1] and Java combination, and performed an evaluation on 198 CPFs in 74 issues. Our results show that PExReport can reproduce 184 out of the 198 CPFs

and achieve a pruning rate of 72.97% on source classes and the classes from internal JAR files.

To sum up, this chapter makes the following contributions.

- A novel framework, PExReport, to extract both code and build dependencies and create pruned executable CPF reports.
- Three technical enhancements of PExReport to handle build configuration, resource files, and generated source code when performing automatic creation.
- An evaluation of PExReport on 198 cross-project failures from 74 software project issues, showing the effectiveness of PExReport on CPF reproduction and project pruning, as well as impacts of each enhancement.

## 2.2 Motivation

In this section, we conducted a pilot study in Java software ecosystems to provide real-world insights into good CPF reports and to illustrate the trilemma faced by current techniques, which motivates the design of our techniques.

### 2.2.1 Characteristics of Ideal CPF Reports

We conducted our pilot study using the JIRA issue tracker from Apache [31]. Upgrade incompatibility failures are strongly related to CPFs; therefore, we used three keywords (upgrade, incompatible, and Java) to retrieve 147 CPF reports from 32 Apache projects and tried to reproduce them.

Two researchers with more than five years of experience in software development were involved in this study. The first researcher recorded the CPF report and tried to reproduce the same failure in our build environment with the information from the report. If failure was not reproducible, the first researcher wrote down the reason, and the second researcher validated the reason. In cases where there was a conflict between the two researchers, a discussion was raised until all issues reached an agreement. Eventually, two researchers were only able to reproduce two of the

147 real-world CPF reports. We summarized the following reasons why these 145 CPF reports are difficult to reproduce; if a CPF report has more than one failed reason, we select the most direct one; the attached number is the frequency of reasons:

- **Never Reproduced (18)**. The client reporter provided an inaccurate test case, so the library developer never reproduced it either.
- **Environment Specific (43)**. The failure is environment specific, so researchers cannot reproduce it without knowing the environment settings.
- **Misuse (16)**. The client reporter misused the library, and the developer explained the reason.
- **No Test Code (41)**. The CPF report is pure textual, and researchers failed to create a test to reproduce the same behavior.
- **Fixed without Record (27)**. The failure was fixed without the specified fix commit, so researchers could not find the code version to reproduce it.

We believe these CPF reports do not deliver enough information for reproduction, especially for new developers who are not familiar with the software project's historical status. A previous study on desirable bug reports [19] also suggests a mismatch between what developers consider most helpful and what users provide. Based on these findings, we summarize the following characteristics of ideal CPF reports:

- **Executability.** A ideal CPF report should be easily reproduced by developers. An executable test case is the most desirable information to be included in reports. Considering the difference in build environments of client and library software projects, required code dependencies, configuration settings, and resource files should also be integrated into the test case.
- **Readability.** Keeping source code accessible is essential for developers to understand the issue. With a helpful code snippet, developers could accurately identify potential misuse

to address CPFs for reporters. Due to the difficulty in locating the CPFs, a good report should retain all the related source code. Destination code is not equivalent to the source code for compiled languages. For example, Java bytecode is more readable than most other destination code, but its decompiled source code with the highest ranking decompiler retains only 78% of its semantics, according to a recent study [34].

- **Conciseness.** Considering a failure is typically triggered by a single execution, low redundancy is preferred in reporting scenarios, which could bring many benefits. A smaller report will reduce the time for network transmission and the space needed for storage (especially when there are many reports). Removing unnecessary code can significantly reduce the noise and accelerate the diagnosis progress. Furthermore, pruning redundant dependencies may also help avoid unintentionally sharing sensitive information or proprietary software artifacts.

### 2.2.2 Cross-Project Failure Report Trilemma

To further understand the obstacles in creating CPF reports, we explored existing automatic build tools and code pruning techniques. Figure 2.1 shows the CPF report trilemma in creating ideal CPF reports with current techniques, which may partially explain why most reporters cannot create CPF reports effectively.

**Modern automatic build tools.** Maven [1] and Gradle [5] are popular tools for building and testing software in Java software ecosystems. Although both tools are designed to be platform-independent, we noticed that they lack features to extract a stand-alone test case. Using these tools can only provide an entire code base with public dependencies. Since these tools cannot identify unrelated dependencies, reporters only could attach the entire private dependencies to the test case. In brief, modern automatic tools only achieve executability and readability but do little to reduce redundant dependencies.

**Program slicing.** Program slicing [90] is a dataflow-based technique for pruning source code based on a specified *seed*, which is known to have limits owing to not working well with dynamic

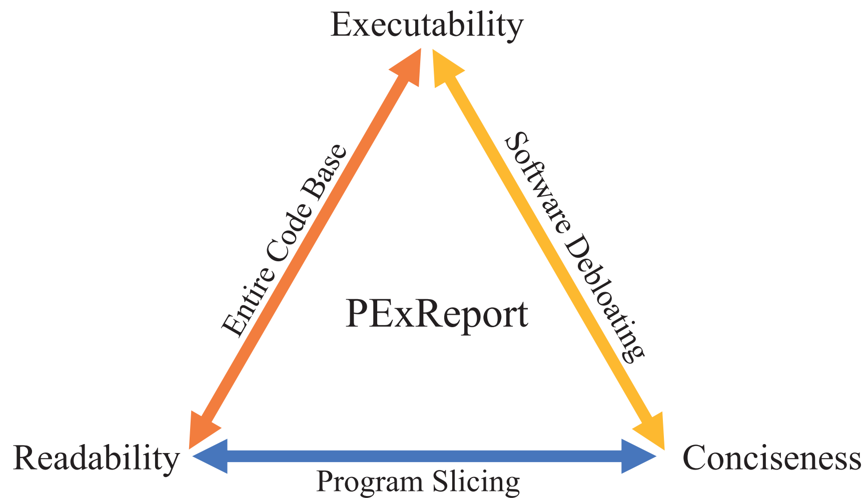



**Figure 2.1**: Cross-project failure report trilemma

features. In addition, program slicing does not consider the execution environment or the build process. Therefore, the created slices of failures cannot be compiled or executed by themselves, which means it lacks executability.

**Software debloating.** In real-world ecosystems, debloating the size of applications is vital in embedded systems and application distribution. There are many practical tools and research studies [22] in this area. For example, ProGuard [11] is integrated into the Android build system, which only runs when building the application in release mode. It detects and removes unused classes, fields, methods, and attributes. In Java applications, released software does not contain source code. Because the output of compiled code is not equivalent to the source code, the developers will receive the CPF report with low readability. In short, the software debloat technique focuses on the software release stage, which does not consider the readability in the test stage.

None of the current techniques can perfectly solve this trilemma, which motivates the design of our approach to create a practical framework–PExReport, to focus on providing CPF reports with Executability, Readability, and Conciseness.

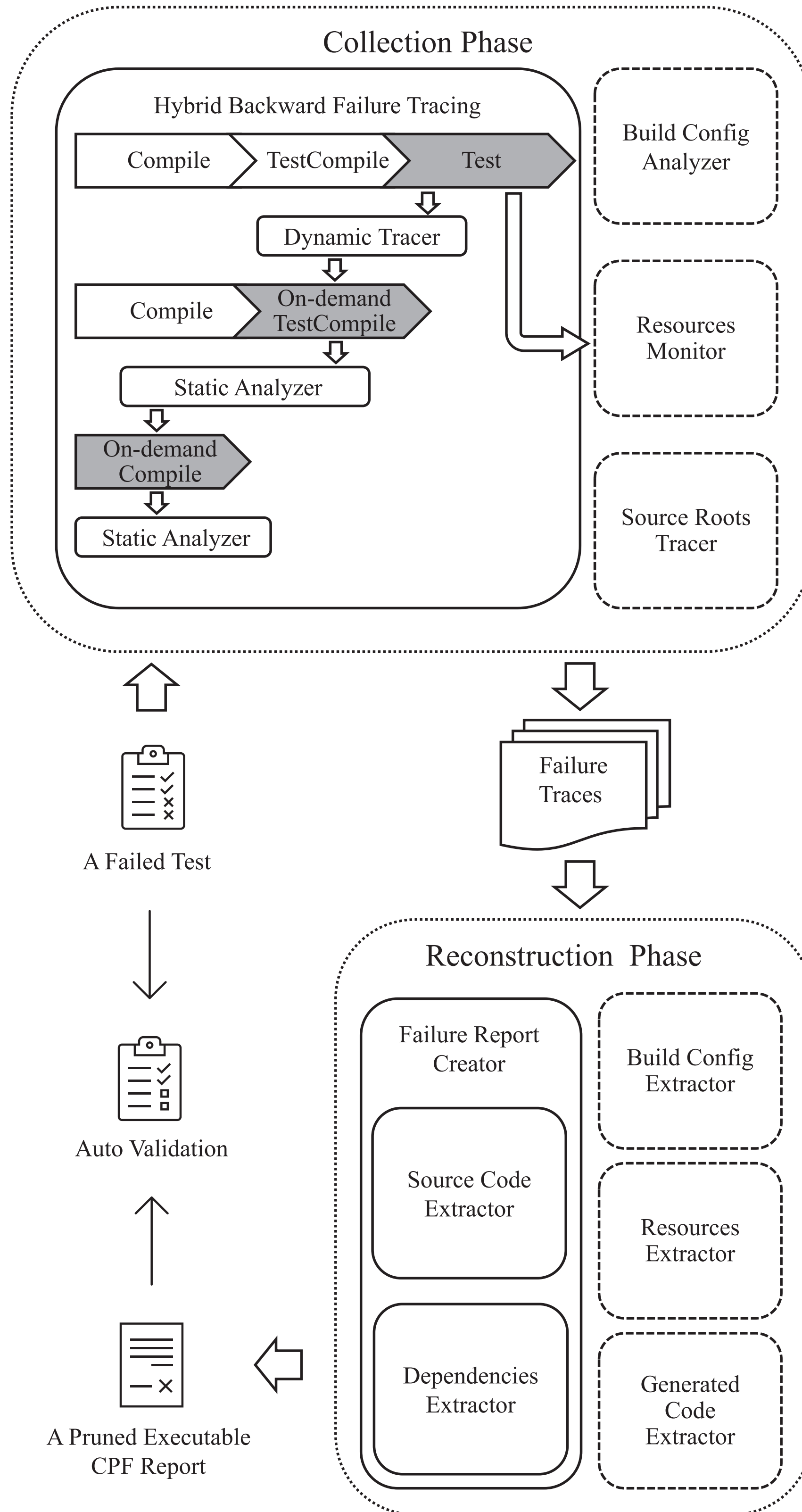



**Figure 2.2**: Overview of the PExReport workflow

## 2.3 PExReport

In this section, we present PExReport, a framework designed to create pruned executable cross-project failure reports automatically. Figure 2.2 shows an overview of PExReport's workflow. PExReport is designed based on prevalent tasks of modern build tools involved in the lifecycle of CPFs. The input to PExReport is a failed test from an existing build environment, and the output is a pruned stand-alone executable CPF project. PExReport contains two major phases: (1) the collection phase to collect information about necessary source code, dependencies, and build environment by monitoring the underlying platforms: OS, the build tool (e.g., Maven), the compiler (e.g., Javac), and runtime (e.g., JVM), in the essential tasks of the modern build process; (2) the reconstruction phase to reconstruct a stand-alone build environment for the failed test based on the collected information. We name the collected information of a failed test from the first phase as failure traces. The arrows show the information flow between each component of PExReport. The reconstructed project for failure reporting will be automatically validated by checking whether the same error messages are triggered as in the original failure. Once the project is validated, it can serve as a reproduction package of the original failed test and will be reported as a pruned executable CPF report to the developers.

The *fundamental insight* of PExReport is that it first identified and addressed the problem of pruning the build and execution environment of a test failure on top of code dependencies. This is crucial for cross-project-failure reproduction because the environment is essential for a high reproduction rate and cannot be easily shared. To overcome this problem, we develop Hybrid Backward Failure Tracing (Section 2.3.3) which extracts and prunes the failure by monitoring the underlying platforms in essential build tasks (i.e., `Compile`, `TestCompile`, and `Test`) of the modern build process. It takes advantage of the fact that the underlying platforms already resolve all necessary dependencies in an on-demand build process and their resolutions are the most trustworthy for reproduction. Following the same insight, we further developed three novel enhancements (Section 2.3.4) to support more heterogeneous build environment.

### 2.3.1 Principles of Design

PExReport should be easy to use and compatible with real-world projects and build ecosystems. We introduce the following principles to help reach the goals.

**Source code preservation.** When clients misuse the library, a snippet of source code can be extremely helpful. Developers could also easily trace the source code to identify the fault location. Considering that even the highest ranking decompiler does not always create semantically equivalent source code [34], source code preservation becomes more critical in CPF reports.

**Build environment adaptation.** A real-world project contains not only source code and dependencies; the build environment is also crucial. Modern build tools, such as Maven and Gradle, have been widely used by library developers. Given that PExReport requires CPF reports to be executable, it should be highly adaptable to modern build ecosystems.

**Incremental creation of build environment.** As the size and complexity of projects increase, modern build tools become complex and highly customizable. Identifying the redundant build environment could be inexhaustible and hard to scale. Therefore, PExReport opts for an incremental approach that creates the build environment by identifying necessary build dependencies, as opposed to a pruning method.

### 2.3.2 Prevalent Tasks for the Lifecycle of CPFs

The following tasks of modern build tools form the typical lifecycle of CPFs. These tasks are generally executed in the order below, but some tasks may be omitted if not required for the current project.

- **Generate sources (optional).** Generate additional source code for inclusion in compilation.
- **Process resources (optional).** Copy and process the resource files into the destination directory.
- **Compile (mandatory).** Compile the application source code of the project.

- **TestCompile (mandatory).** Compile the test source code, which is not coupled with the application source code.

- **Test (mandatory).** Run tests using a suitable testing framework; execute object code (e.g., bytecode) compiled from application and test source code.

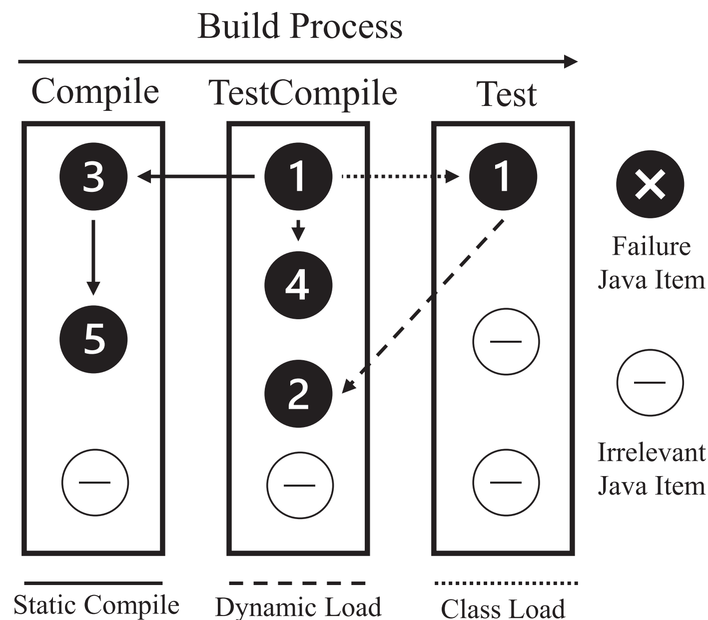



**Figure 2.3**: Exemplar three tasks build process of Java project

The three mandatory tasks, namely `Compile`, `TestCompile` and `Test`, are the basic steps used by build tools to reproduce a test failure. The main reason to use both `Compile` and `TestCompile` to compile source code is to make the application source code independent of the test source code. Code generation and resource management are quite popularly used in complex real-world applications. Although they are optional tasks, we should not underestimate them in reproducing real-world CPFs.

### 2.3.3 Base Approach: Hybrid Backward Failure Tracing

In the collection phase, our base component is hybrid backward failure tracing, a three-step analysis that traces failure-related source code and library dependencies. This component compiles and

executes the failed test in its original build environment, and tracks items at `Test`, `TestCompile` and `Compile` tasks, which are essential for any failure source code to be compiled and executed.

Figure 2.3 shows the dependency tree in a build process for a failed test of Java project with three basic tasks. We refer to source code and object code (bytecode) as *items*, which are used to compile and execute the failed test, respectively. All the circles are the items of the Java project, a black circle means a failure item, and a white circle means an irrelevant item. The dotted line arrow is for class loading, so the two *Item 1* in the two tasks are equivalent but in a different form (source code in `TestCompile` task, bytecode in `Test` task). The solid line arrows show the static dependencies, and the dashed line arrows show the dynamic dependencies. Following the build process order, the compiled application source code in `Compile` task is provided to the subsequent `TestCompile` task as dependencies, after that, the compiled test source code of `TestCompile` is loaded to `Test` task for execution. For example, the `Compile` task compiles *Item 3* (application source code), which is provided to `TestCompile` task as a reference later; the *Item 1* (bytecode) of `TestCompile` task is loaded to `Test` task. The build process must be irreversible to ensure that the previous item will not depend on the later item. Given that the build process cannot predict the required items for the subsequent task, PExReport must perform the build process in three rounds to track all failure items. For example, in Figure 2.3, the reference $1 \rightarrow 2$ can only be discovered in `Test` task, reference $1 \rightarrow 3$ can only be discovered in `TestCompile` task, and the reference $3 \rightarrow 5$ can only be discovered in `Compile` task. If a CPF occurs, as failures happen at the end of the build process (`Test` task), a backward tracing will be performed to obtain the failure traces. The details of Hybrid Backward Failure Tracing are described in Algorithm 1.

**Algorithm 1** Hybrid backward failure tracing Algorithm

**Input:** Failed test $t$, all test source code $C_t$, all application source code $C_s$, all library object code $O_l$

**Output:** $Traces$

1: Initialize trace of $C_t$: Set $T \leftarrow \emptyset$
2: Initialize trace of $C_s$: Set $S \leftarrow \emptyset$
3: Initialize trace of $O_l$: Set $L \leftarrow \emptyset$
4: **Round 1:**
5: $Compile$: Object code $O_s \leftarrow$ Compile $C_s$ by referencing $O_l$;
6: $TestCompile$: Object code $O_t \leftarrow$ Compile $C_t$ by referencing $O_s$ and $O_l$;
7: $Test(t)$: Record $R_1 \leftarrow$ Execute $t$ by dynamic loading $O_s$, $O_t$ and $O_l$;
8: $T, S, L \leftarrow DynamicTracer(R_1)$;
9: **Round 2:**
10: $Compile$: reuse $O_s$;
11: $TestCompile(T)$: Record $R_2 \leftarrow$ On-demand compile test source code in $T$ by referencing $O_s$ and $O_l$;
12: $T, S, L \leftarrow StaticAnalyzer(R_2)$;
13: **Round 3:**
14: $Compile(S)$: Record $R_3 \leftarrow$ On-demand compile application source code in $S$ by referencing $O_l$;
15: $S, L \leftarrow StaticAnalyzer(R_3)$;
16: **return** $Traces\{T, S, L\}$

In **Round 1**, PExReport runs the whole build process to the last task–`Test` with entire code base and library dependencies for both `Compile` and `TestCompile` tasks, and executes only the target failed tests in the `Test` task. PExReport uses the log of build tool to record all dynamically loaded items in the `Test` task. These items are required for the execution of the failed test in runtime, so we refer to them as $R_1$. For example, in a Java project, $R_1$ are the bytecode (object code) loaded into JVM, also referred to as Java class. The $DynamicTracer$ analyzes $R_1$, and collects the corresponding traces (usage of items).

In **Round 2**, PExReport runs the build process up to the `TestCompile` task, reusing the object code compiled from all application source code in `Compile` task. For the `TestCompile` task, PExReport on-demand compiles test source code collected from *Round 1* to avoid compiling irrelevant items. In `TestCompile` task, PExReport records all referred items as $R_2$. These items are required for the compilation of test source code, so $StaticAnalyzer$ collects and combines the corresponding traces.

In **Round 3**, PExReport runs a single `Compile` task of the build process, trying to compile only the application source code collected from *Round 1* and *Round 2*. In the `Compile` task, PExReport records all referred items as $R_3$. These items are required for the compilation of failure related application source code, so $StaticAnalyzer$ collects the corresponding traces and combines them with previous traces.

**Traces.** PExReport returns the collected failure traces of test source code, application source code and library object code in Algorithm 1, and sends the traces to the failure report creation component to be included in the executable CPF reports.

PExReport uses the build tools to resolve dependencies to increase robustness and non-invasiveness. Modern build tools can fetch information directly from compilers or virtual machines, meaning the resolved dependencies are the most trustworthy. The inferred dependencies from the analysis tools may be incomplete due to the quality of the tools, and some invasive tools can change the behavior of the analyzed projects. However, the build tools only provide the resolved dependencies without the dependency tree, which means that PExReport cannot use all the items in each task that may contain irrelevant items during the build process. As shown in Algorithm 1, to solve this problem, PExReport uses on-demand compilation[1] to compile only the failure related source code and then use the references of compilation as failure traces.

The dynamic loading and static reference are handled by the run-time environment and the compiler, respectively. PExReport uses the dynamic tracer to monitor dynamic item loading in the `Test` task and the static analyzer to monitor the reference in the `Compile` and `TestCompile` tasks due to the fact that the information provided by the build tools differs between compilation and execution. The current PExReport supports the default Java compiler and multiple customized compilers, such as *javac-with-errorprone*; if any compiler is not supported, PExReport only needs an adjustment to the static analyzer. As the build tools sort different types of items into different directories, the PExReport can accurately categorize items with the location information and further allow the Failure Report Creator to place them in corresponding directories.

[1] On-demand compilation, such as the implicit compilation in Java, which allows the compiler to search the required source code and dependencies to compile the designated source code.

### 2.3.4 Three Enhancements

As discussed in Section 2.2, besides source code and library dependencies, the build environment also plays an important role in the reproduction of failed tests. Although the base approach can still reproduce the failure with source code and library dependencies in the minimal standard build environment, in order to enhance the reproduction rate for real-world complex projects, we developed the following enhancements over the base components to further reconstruct a reliable build environment:

- **Handling of the build configuration.** In the collection phase, the build configuration analyzer fetches the resolved build configuration and determines the necessary values of them, which will be then inserted into the template project by the build configuration extractor in the reconstruction phase.
- **Handling of resource files.** For the collection phase, we developed the resource monitor to watch all file accesses in the project directory during the test execution at the operating system level. Then, in the reconstruction phase, our resource extractor will copy and insert pruned resource files into the proper locations of the reconstructed project.
- **Handling of source code generation.** Our source roots tracer tracks source code generation in the collection phase, and the generated code extractor locates the generated source code and extracts the needed source code to skip unnecessary code generation and avoid code generation conflicts.

**Handling build configuration.** Build configuration values are necessary parts of an executable failure report. Including unnecessary configuration values will lead to more noise in the debugging phase (the debugging developer needs to consider more factors), more information leaks (e.g., internal emails and file paths), as well as potential build failures, for example, the configuration values may refer to pruned source code and dependencies which no longer exist. Therefore, we enhance PExReport to further identify and extract only build configuration values necessary for the failed test reproduction.

In the Collection Phase, the Build Configuration Analyzer fetches the *effective build configuration* (e.g., Maven provides *help:effective-pom* [10]), which the build tools use to build the failed test at run time. The *effective build configuration* gathers all build configuration values scattered in all build configuration files, and resolves configuration value overwriting among multiple configuration files based on the build configuration hierarchy and resolution rules, for example, Maven picks the "nearest definition" for resolving dependencies. However, the *effective build configuration* is still redundant for reproducing the failed test, compared with the required configuration values of the three main build phases (compile, test-compile, and test). Since tasks in the build process are performed by various plug-ins and the plug-ins can be attached to different build phases (e.g., the Java compiler plug-in can be attached to the compile and test-compile phases), PExReport further leverages the attachment relationship to identify all the plug-ins that are attached to the three basic tasks. It also excludes code-style checking and analysis plug-ins because they do not directly affect the compilation or testing. After all necessary plug-ins are identified, the Build Configuration Extractor extracts the configuration of the plug-ins from the *effective build configuration* by querying with XPath. After modifying the static information, such as the absolute project path, the extractor feeds all the extracted information to the build configuration file of the reconstructed project.

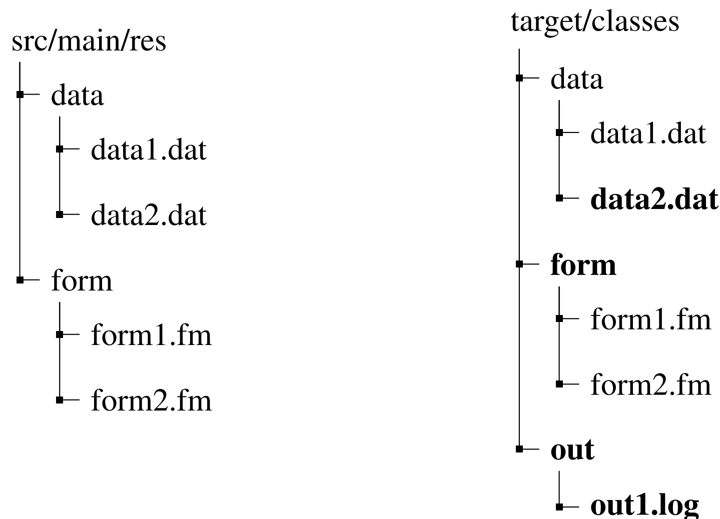



**Figure 2.4**: Exemplar resource directory structure

**Handling resource files.** The Resource Monitor identifies necessary resource files for the failed test reproduction by using file system monitoring API (e.g., *inotify* [4]–a Linux kernel sub-

system that monitors filesystem events) to watch the resource file access during the `Test` phase. We use an example in Figure 2.4 to illustrate how we monitor and extract resource files, the left side is an original resource folder existing before the build, and the right side is a target resource folder generated after the build. The files/folders accessed during testing are highlighted in bold font.

As shown in Figure 2.2, the resource monitor only outputs files accessed at `Test` task because the `Process resources` task often copies all resource files to the target folder (e.g., the whole directory structure on the left of Figure 2.4). For necessary resource files accessed during the compilation process (e.g., templates of generated source code), we specially handled them by the Generated Code Extractor. Our resource monitor also ignores all files/directories generated during the build/test process (e.g., folder `out` and file `out1.log` in Figure 2.4, *.class* files) because these files should not be included in the reproduction package (unnecessary and causing path conflicts). For the files copied to the target location from source locations (e.g., folders `data` and `form` and all their files in Figure 2.4), we do not consider them as generated files, because the Resource Monitor can trace back to their source copies in the original project.

During the reconstruction phase, our Resource Extractor extracts the files and directory structure from the original project based on information collected by the Resource Monitor. For copied resource files, the Resource Extractor uses their source copies and paths tracked by the Resource Monitor (e.g., extracting `data2.dat` from the original resource folder because its copy in the target folder is accessed). Note that maintaining the structure of an accessed directory is also crucial for test reproduction. The failed test may access the directory but never access the files under it (e.g., checking the existence of a file), and some tests require a special directory structure to reproduce successfully. The Resources Extractor creates empty dummy files for the un-accessed files under the accessed directory to retain the directory structure and cover this situation. For example, in Figure 2.4, folder `form` is accessed and can be traced back to the original resource folder, but none of its files is, so the folder will be extracted, and all its files will be replaced with empty files with same names.

**Handling source roots and generated code.** The software build process may generate new source code in various ways, such as creating code from template files, generating parsing code from syntax/XML files, and even directly fetching source code from remote locations. Furthermore, code generation is often implemented in third-party tools and plugins. To handle such high flexibility of code generation in a general way, we enhance PExReport by omitting the code generation process and directly including the generated source code in the test reproduction package.

At `Generate sources` task, the build tools use source code root paths (Source Roots) to locate all (original and generated) source code. In the Collection Phase, our Source Roots Tracer tracks all the accessed source code root paths from the debug information of compilation. Next, the Generated Code Extractor utilizes the paths to identify the generated source code and excludes all original source code. In addition, the build tools may also generate some source code from code annotation processing. Our Generated Code Extractor excludes such code because it will cause compilation conflicts.

### 2.3.5 Automatic Creation of CPF Reports

In the reconstruction phase, our base component is failure report creation. This component creates a template project with a standard build configuration for the failure report and adds the required source code and dependencies into the project.

The Failure Report Creator uses a customizable template to generate a standard project structure for the reproduction package. It uses different extractors to fetch required source code, dependencies, resource files, and build configuration. As two basic components shown in Figure 2.2, the Source Code Extractor converts the required item name to code file paths and copies these files to the generated reproduce project while maintaining the original structure; the Dependencies Extractor duplicates the local and remote dependencies and reduces the unnecessary portion based on failure traces. The generated project organizes and links the required source code and dependencies in a standard build configuration file for reproduction. As mentioned earlier, in many cases, we also need to extract the build environment accordingly so that the replication package can success-

fully reproduce the failure. Therefore, the Failure Report Creator can further extract the required build configuration, resource files, and generated code by incorporating the Build Configuration Extractor, Resources Extractor, and Generated Code Extractor, respectively.

### 2.3.6 Failure Report Validation

After the final executable failure report (i.e., reproduction package) is constructed, the Report Validator uses a conservative strategy to ensure the report can reproduce the original test failure. In particular, a report is validated only if it generates the identical failure type and message after executing the test. Note that this strategy may reject some successfully reproduced test failures (e.g., when failure messages change with date, time, or absolute path), but it allows developers to trust the pruned failure report (and our evaluation to be conservative). Note that in practice the reporter could still manually validate the report when the automatic validation fails.

## 2.4 Evaluation

This section presents our experimental results by answering the following research questions:

- **RQ1: How effective is PExReport in creating executable CPF reports? (Executability)** To answer this research question, we counted the number of subjects that PExReport can exactly reproduce with the same failure type and message, and calculated the reproduction rate.

- **RQ2: How does PExReport perform on project pruning? (Conciseness)** To answer this research question, we calculate the reduction rates of different items from subjects and present cumulative graphs to show the performance of pruning. If a reproduction fails, we use the entire project as the CPF report (0% reduction rate).

- **RQ3: How effective are our techniques in PExReport on solving the CPF report trilemma?** To answer this research question, we performed an ablation analysis using reproduction and reduction rates.

### 2.4.1 Experiment Setup

We implemented PExReport in Python, based on Apache Maven [1] build tool. Since Maven is used primarily for Java, we chose Java projects as our target projects. The Archetype [2], a Maven project template toolkit, is used for generating a standard Maven project for reproduction.

We performed our experiment on a Linux server running Ubuntu 16.04.5 LTS with two 8-core 2.6 GHz CPUs and 512 GB RAM. Apache Maven 3.6.3 was used to build and run tests. To avoid race conditions, we solely executed each build and test on the server. An automatic Failure Validator examined the failure types and messages to ensure the failures had been successfully reproduced. If a failed reproduction was reported, the entire project was provided as a failure report for evaluation.

### 2.4.2 Metrics

To answer **RQ2**, we need to use some general metrics to compare pruning performance among all subjects. Well-defined metrics could also help us to understand the results correctly. PExReport prunes the entire building environment of the failed test, including source code, dependencies, build configuration, and resource files. So, our metrics should measure the pruning on all of them and we define the following metrics:

- **# Internal classes**: The number of classes from the JAR dependencies within the same organization of the subject. In the Maven convention, the same organization typically shares the same domain name in their group ID of libraries. So, we compare the group IDs to identify internal libraries and count the internal classes inside libraries.
- **# Source classes**: the number of classes compiled from the Java source code.
- **# Source+Internal classes**: The total number of source and internal classes. Our subjects have vastly different distribution strategies for main classes and internal classes. So, we combine these classes to provide a more generic metric and reduce the interference of different code distribution strategies. Since we measured the total of source and internal classes,

and classes could be compressed in JARs, we chose the number of classes instead of the size of classes.

- **Size of build configuration**: The total number of characters from the loaded POMs (Maven build configuration files), excluding the whitespace and *dependencies* section for a fair comparison. We uses *# internal classes* as a metric for dependencies, which is more precise.

- **# Resources**: The number of resource files within the resource directories of the project.

- **Percentage of reduction**: The percentage of reduced items, defines as the ratio of removed items to original items. For example, the percentage of reduction of internal classes is the ratio of the number of pruned internal classes to the total number of internal classes in original projects.

### 2.4.3 Dataset Construction

Our experiment dataset is constructed from the benchmark dataset of a research project–Sensor [87], which triggers failures by changing dependency versions of open-source Java projects. The Sensor ground truth dataset contains 316 semantic conflicts confirmed by researchers. In our dataset, we refer to one unique failed project-library pair as one cross-project issue and one failed test case as one CPF. Although PExReport can handle CPFs raised by multiple test cases (treat multiple tests as one test), clustering tests based on dependencies is beyond the scope. Given that each issue may contain multiple CPFs, our experiment dataset consists of 74 issues with 198 CPFs.Â

We do not use every CPF in Sensor dataset as subjects to evaluate PExReport because (1) some issues are irreproducible due to environmental change; (2) some CPFs are irreproducible solely since they are dependent on other CPFs (clustering non-independent tests is beyond the scope of this chapter); (3) some CPFs have random factors or are flaky, so they may cause uncertainties during validation; and (4) many CPFs in one issue are identical, so using all of these repetitive failures will dominate the results. Therefore, we performed the selection for representative CPFs in the following steps:

- **Step 1: Verifying failed issues.** After removing duplicated project-library pairs, we downloaded projects into our server and executed them without applying any changes. For those successfully executed projects, we re-executed them with conflicting libraries based on the Sensor dataset. Then, we selected all the issues that only failed with changed dependencies in our experiment environment. We set a 900-second time limit for each execution and removed all projects with timeout and compilation failures. Only three issues were discarded due to timeout; 119 issues were collected throughout this step.

- **Step 2: Clustering CPFs.** We executed all CPFs three times and removed all CPFs with inconsistent output over three executions, after which five issues were discarded. For each of the remaining issues, we clustered CPFs with the identical failure type and message (failure group) into one cluster. As a result, we clustered 6,415 CPFs into 1,020 failure groups from 114 issues. The maximal number of failure groups within an issue is 367.

- **Step 3: Selecting representative CPFs as subjects.** We observed that an issue may have significantly more failure groups (i.e., 367) than others because test failures whose messages contain unique numbers (e.g., Java Object ID) are clustered into distinct failure groups. In addition, ninety percent of failure types contain no more than four failure groups. To avoid over-representation, we further considered failure types (e.g., Assertion Failure, No Such Method) by selecting up to four failure groups at random for each failure type and issue, and then randomly selecting one representative CPF for each selected failure group. In this step, we only selected CPFs that are solely reproducible; 105Â out of 395 (26.5%) failure groups and 45 out of 238 (18.9%) failure types have been discarded due to none of their CPFs being solely reproducible. We observed that 33 of the 107 selected issues lack internal libraries. For a more comprehensive and consistent evaluation, we only use the 74 issues with internal libraries.[2]

[2]The data used in this chapter and PExReport implementation are available at `https://doi.org/10.5281/zenodo.7578677`; additional evaluation results for the 33 issues and concrete examples can be found on our website: `https://sites.google.com/view/PExReport/home`

**Table 2.1**: Statistics of the collected issues.

| Item | Min. | Max. | Mean | Med. | *≠0* |
|---|---|---|---|---|---|
| # Internal classes | 32 | 36,389 | 3,848 | 955 | 74 |
| # Source classes | 8 | 2,457 | 265 | 144 | 74 |
| # Source+Internal cls | 90 | 36,417 | 4,113 | 1,331 | 74 |
| Size of Config | 8,428 | 77,735 | 25,349 | 21,746 | 74 |
| # Resources | 0 | 655 | 55 | 8 | 68 |
| # Total issues | | | | | 74 |

Eventually, we collected 198 representative CPFs from 74 issues. Table 2.1 shows the statistics of the issues in our dataset. The symbol *≠0* means not equal to zero. The *≠0* column shows the count of issues that have at least one corresponding item. For instance, an issue that does not have any resource files will not be counted.. The statistics show that the issues in our final dataset have a large number of classes (4,113 Mean and 1,331 Median of *# Source+Internal classes*).

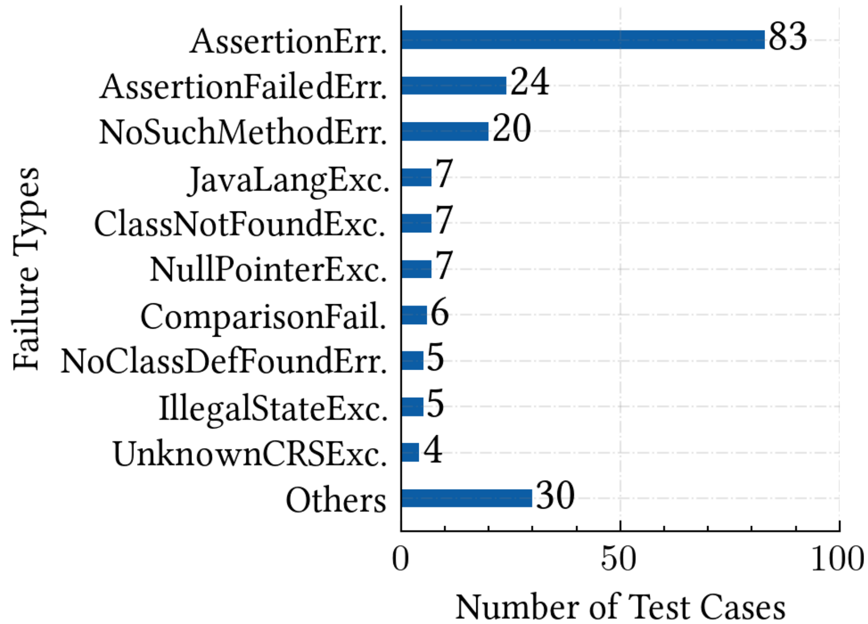



**Figure 2.5**: Top 10 failure types in representative CPFs

We further categorized the 198 CPFs based on their failure type to show the diversity of failures in our dataset. The total number of different failure types is 31, and the top 10 failure types are shown in Figure 2.5. The top two failure types are assertion-related, representing more than half

of CPFs. The assertion-related errors indicate that the actual variable values are not equal to the expected values. Another common failure type is “not found”. If the dependency change modified class/method signatures, the program could still fail in the `Test` task in the case of reflection.

### 2.4.4 Evaluation Results

#### The answer to RQ1

To answer the RQ1, we executed PExReport on 198 representative CPFs and summarized the results in Table 2.2. PExReport successfully reproduced 184 out of 198 CPFs with exact failure types and messages, and achieved a high reproduction rate of 92.93%.

**Table 2.2**: Reproduced CPFs

| Implementation | # Repro. CPFs | Repro. Rate | Perf.(sec) |
|---|---|---|---|
| **PExReport** | **184** | **92.93%** | **73.03** |
| w/o Dynamic | 48 | 24.24% | 66.51 |
| Source & Deps. | 46 | 23.23% | 51.22 |
| w/o Build Config | 101 | 51.01% | 65.79 |
| w/o Resources | 104 | 52.53% | 68.61 |
| w/o Generated Code | 146 | 73.74% | 55.08 |
| # Total CPFs | 198 | | |

We investigated the 14 unreproduced CPFs to understand why PExReport cannot reproduce them. 9 of the 14 CPFs failed because of the unsupported compilers (i.e., Groovy [3] compiler), which could not provide the class reference information for Hybrid Backward Failure Tracing. The rest of the 5 unreproduced test cases are missing required classes at run time. We believe the Java Runtime failed to provide all the information on required classes during the test execution. The mistracking could happen when a program checks the existence of a class but never access it.

#### The answer to RQ2

We further calculated the percentage of reduction for each metric and presented Table 2.3 to show the statistics of the percentage of reduction. PExReport performed a 55.37% of average reduction

**Table 2.3**: Statistics of reduction rate

| Percent Reduction | Min. | Max. | Mean | Med. |
|---|---|---|---|---|
| # Internal classes | 0.00% | 100.00% | 75.94% | 84.67% |
| # Source classes | 0.00% | 99.05% | 55.37% | 64.39% |
| # Source+Internal cls | 0.00% | 99.65% | 72.97% | 79.95% |
| Size of Config | 0.00% | 96.76% | 82.96% | 89.02% |
| # Resources | 0.00% | 100.00% | 74.18% | 90.91% |

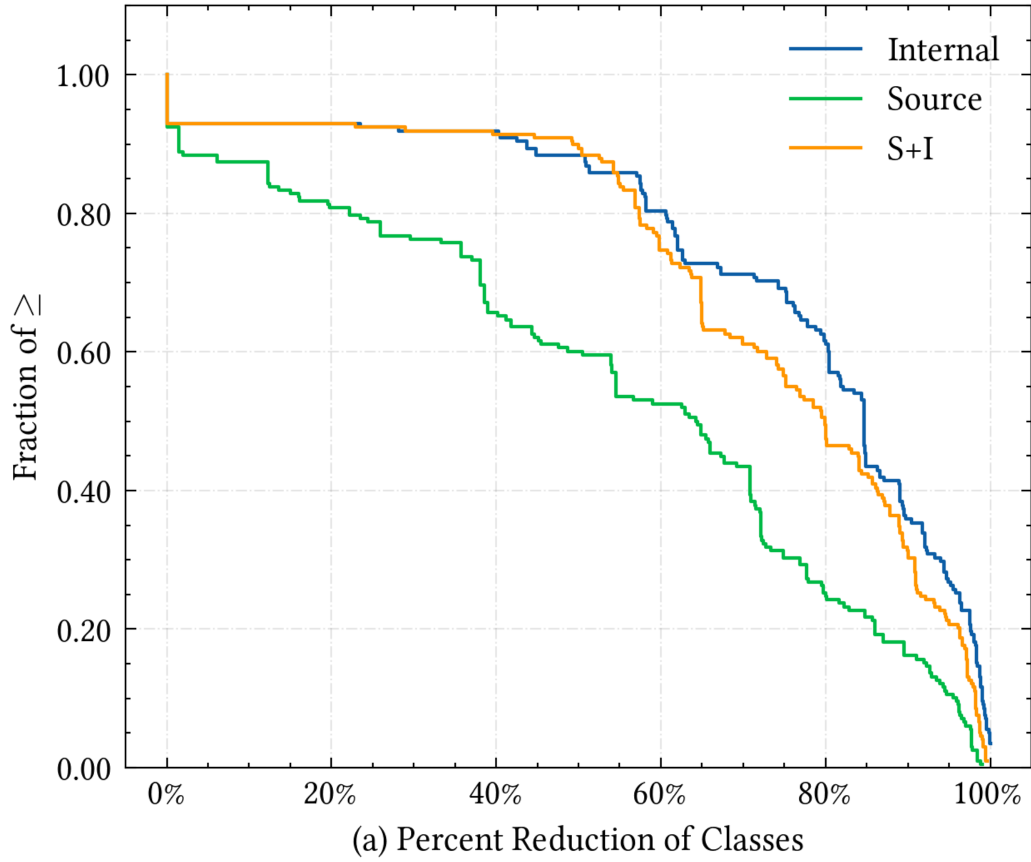
rate on required source classes and outperformed on internal classes, source+internal classes, build configuration, and resources with 75.94%, 72.97%, 82.96%, and 74.18%, respectively.

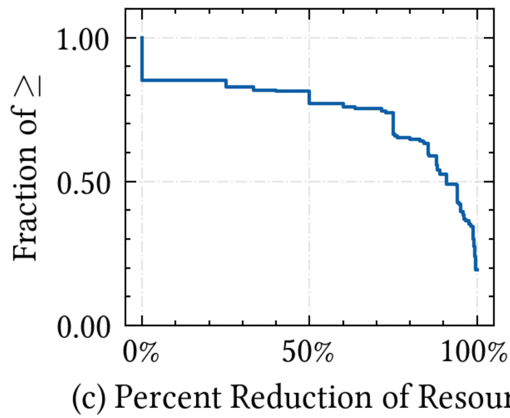
We assume the reporter still wants to use the entire project as a failure report if PExReport fails to reproduce the failure. Therefore, the percentage of reduction is 0% for unreproduced subjects. We presented the cumulative plot of results in Figure 2.6. The S+I represents Source+Internal, shown as the orange curve. The y-axis is the fraction of x that has satisfied the $\geq$ condition, which means that the fraction of results have at least a certain percentage of reduction. For example, a point ($x = 60\%$, $y = 0.79$) on the blue curve (Internal) shows that 79% of CPFs have a reduction rate over 60% on internal classes.

PExReport performed impressively on almost all the metrics. As the source code can provide excellent readability to developers, we believe 55.37% of the average reduction rate is acceptable in the CPF report. Our investigation shows that the high reduction rate for build configuration is based on non-test-related configuration, such as the deployment settings and unused plugin settings. Figure 2.6 shows that PExReport could provide a stable reduction rate for the redundant dependencies. In conclusion, the high reduction rates indicate that PExReport could provide conciseness to CPF reports.

#### The answer to RQ3

Besides the Hybrid Backward Failure Tracing, PExReport applies three enhancements to handle build configurations, resource files and generated code. We performed an ablation analysis on PExReport and calculated reproduction and reduction rates.



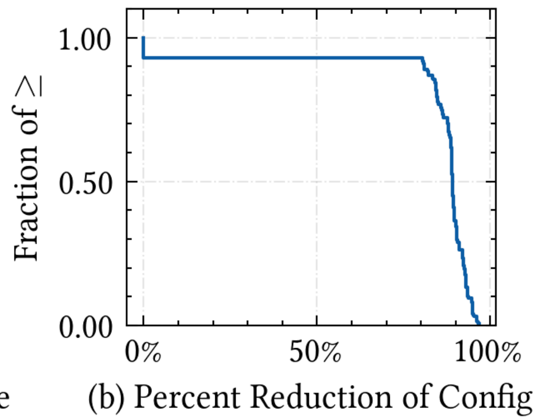




**Figure 2.6**: Cumulative step graph of reduction rates

*w/o Dynamic* represents PExReport without the dynamic tracer (JVM monitoring), which still retains the static analyzer and three enhancements. As shown in Table 2.2, *w/o Dynamic* only achieved a low reproduction rate of 24.24% by reproducing 48 of 198 CPFs. Compared with the high reproduction rate of 92.93% from PExReport, it states that pure static analysis could perform terribly in real-world Java applications as large Java applications use various dynamic features. *Source & Deps.* represents PExReport without three enhancements, which provides source code and dependency packages with the standard Maven build configuration. The *Source & Deps.* only reached 23.23% (46 of 198 CPFs), a low reproduction rate. Although source code and dependencies contain the most information that developers expect the reporter to provide, it is still highly likely that developers cannot use them with the standard build setting to reproduce the CPF, as exemplified in Section 2.2. To better understand the effectiveness of the three enhancements, we turned off each enhancement one by one and compared the results with the integrated PExReport. As shown in Table 2.2, the impacts of these enhancements are large. The reproduction rate reduces from 92.93% to 51.01% and 52.53% without enhancements on build configuration and resource files, respectively. Without the generated code enhancement, 73.74% of CPFs can still be reproduced because code generation is not a must for many Java projects. In contrast, developers need to deal with build configuration in every project, so the corresponding enhancement has the largest impact.

We also presented the reduction results of ablation analysis in Figure 2.7, using cumulative plots to intuitively compare our techniques.

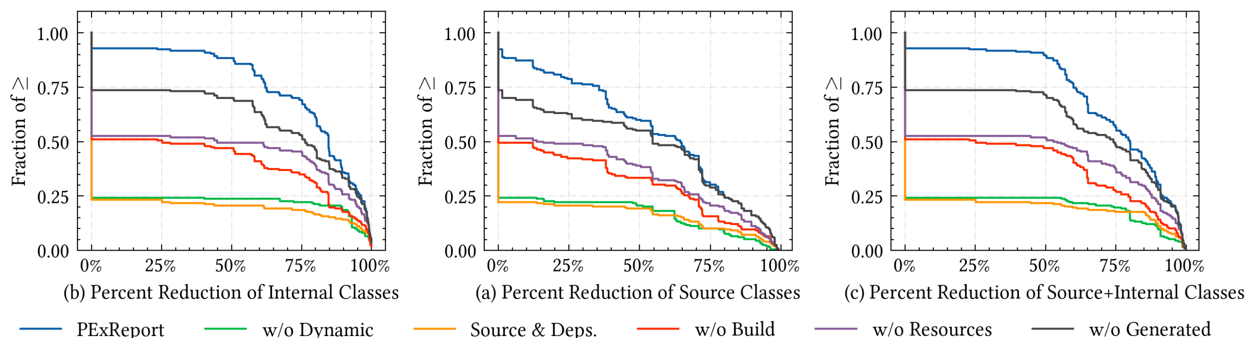



**Figure 2.7**: Comparison of different PExReport techniques

As shown in the performance column of Table 2.2, PExReport's execution time is not heavily affected by these enhancements. As creating CPFs is not a repeated activity like the standard build process, the average reproduction finished in a minute or so should be acceptable.

Overall, PExReport reproduced the majority (92.93%) CPFs and provided executable cross-project failure reports with an average of 72.97%, 82.96%, and 74.18% reduction rate for source and internal classes, build configuration, and resources, respectively.

### 2.4.5 Threats to Validity

The major internal threat to validity is the potential errors in our scripts and tool building. To reduce this threat, we carefully checked the implementation and shared them for peer review. The major external threat to validity is the variance of projects and test failures that may not be covered by our evaluation. To reduce this threat, we constructed our dataset based on the ground truth research dataset–Sensor [87] and carefully chose CPFs to avoid bias. Although PExReport is designed to work on all reproducible failures, our evaluation does not consider flaky tests and non-independent tests to avoid uncertainties and inconsistencies. Consistently reproducing and clustering failures are beyond the scope of this chapter, but may also require future research.

## 2.5 Discussion

**Generalization to Other Languages and Building Tools.** PExReport is designed based on Maven and implemented for Java programming language. However, the idea of our approach is general and may be applied to other programming languages and build tools. The dependency analysis and enhancements for build configuration, resource files, and generated code are generalizable to most JVM languages, but compiler integration may be different from language to language, so the Hybrid Backward Failure Tracing component needs to be adjusted to support new languages. Other build tools such as Gradle / Make may allow more complicated file operations so more advanced analysis of the build process may be required.

**Duplicate Failure Reports.** PExReport takes a single CPF as its input, but one issue may

generate more than one CPFs. PExReport does not resolve the potential duplication of CPFs but it can be resolved using test failure clustering and selection of representative CPF from each cluster. In our experiment, such clustering largely reduced the client CPFs to be considered. In practice, the developer may manually determine which CPF should be reported and which one should not, and select a representative CPF. Furthermore, even if the developer chose two CPFs with a similar root cause, after the pruning, representative CPFs sharing the same root cause may be reduced to similar executable CPF reports. In such cases, the client developer may need to double-check the similarity between executable CPF reports before submitting them.

## 2.6 Related Work

Though we are not aware of any research efforts creating pruned executable CPF reports to solve the CPF report trilemma, our techniques are related to existing efforts on bug reproduction, program slicing, software de-bloating, and fault localization.

**Bug and Crash Reproduction.** Recently, JCHARMING [63] uses crash traces and model checking to identify program statements needed to reproduce a crash. Liu et al. proposed Double-Take [54], which uses evidence-based analysis to largely reduce the cost of recording erroneous states. Huang and Zhang proposed LEAN [42] to reduce the complexity of the replayed trace and the length of the replay time without losing the determinism in reproducing concurrency bugs. Weeratunge et al. [89] propose a novel approach that performs a lightweight analysis of a failing execution in a multi-core environment and reproduces the bug in a single-core system, under the control of a deterministic scheduler. Herbold et al. [39] proposed a generic and non-intrusive GUI usage monitoring mechanism to record and replay GUI bugs. Moran et al. [60] proposed a technique that records user action steps when reproducing an Android bug, and automatically fills them into a bug report. Some other research efforts [36] [35] try to automatically construct build or execution environments but they have not been applied to bug reproduction. Compared with existing approaches in this area, our approach focuses on the pruning and reconstruction of the build environment for buggy execution, including build configuration, resource files, and generated code.

**Program Slicing.** Program slicing [90] is a technique to carve from a large program a smaller program that implements one or multiple features of the larger program. Program slicing often relies on code dependency graph [28] [86] and has been used to prune a lot of targets from source code [78] to pre/post conditions [33], paths [43], and databases [91]. Dynamic slicing [13] [99] identifies the statements that the buggy output depends on. Executable union slicing preserves the meaning of the original program using conditioned slicing. [27] Compared with program slicing, PExReport focuses more on the build environment. The low reproduction rate of PExReport's variant without enhancements shows that fetching only code dependency is not sufficient. On the other hand, PExReport's enhancements can also be viewed as a slicing of the build environment, including the build configuration and the resource files.

**Software De-bloating.** Software de-bloating techniques prune a released software package to remove unnecessary dependencies and thus reduce its size, loading time, and attack surface. Pure static de-bloating techniques such as ProGuard [11] and Jax [81] rely on static program slicing and more recent work such as JShrink [22] further consider runtime dependencies. Although both pruning code, compared with PExReport, the goal of de-bloating is to retain software features instead of reproducing a single execution. Also, although de-bloating may generate executable pruned code by tracking execution dependencies, it does not work on source code and does not retain the build environment.

**Fault Localization.** Statistical fault localization [45] [44] gives a suspicious score to each statement (or other types of code structures such as sub-control-flow-graphs [25]) according to the number of successful and failed test cases that cover the statement. IR-based fault-localization [85] evaluate suspiciousness scores of statements by their textual similarity to the descriptions in a given bug report. Change-aware fault localization, such as Delta-debugging [97] [96] [94] considers the scenario of localizing regression faults when the history between the successful version and failed version is available. Other approaches [98] [71] perform impact analysis on code changes to localize faults. Hassan and Wang [37] studied localization and repair of bugs in build scripts. On cross-project bugs, Mostafa et al. [61] and Chen et al. [24] studied behavior backward incompatibil-

ities and their detection. Compared with all these techniques, PExReport focuses on reproduction instead of localization of bugs, so it needs to further identify and package all dependencies of a bug in its build and execution process.

## 2.7 Future Work

In the future, we plan to enhance our research in the following directions. First of all, we plan to enlarge our dataset to evaluate PExReport on more CPFs in more projects. Second, we plan to further prune the created executable CPF reports with finer-grained analysis and perform source code detailed reduction. Third, we plan to perform user studies to understand how much our tool can help developers in reproducing real-world bugs. Fourth, we plan to expand PExReport with other build tools by developing more advanced features for different software ecosystems.

## 2.8 Conclusions

An executable test case is one of the most desirable features of failure reports. When reporting *cross-project failures (CPFs)* to library developers, a test case is even more helpful because code is a natural way to describe interactions between library code and client code. In this chapter, we developed PExReport, a framework to automatically create pruned executable CPF reports for developers, and solve the CPF report trilemma. PExReport uses Hybrid Backward Failure Tracing to identify the necessary source and dependencies, and has further enhancements to handle build configurations, resource files, and generated code. Our evaluation shows that PExReport can produce pruned executable CPF reports for 184 of 198 CPFs with an average reduction rate of 72.97%.

# CHAPTER 3: PEXREPORT-MAVEN: CREATING PRUNED EXECUTABLE CROSS-PROJECT FAILURE REPORTS IN THE MAVEN BUILD SYSTEM

## 3.1 Introduction

Modern Java software development extensively depends on existing libraries written by other developer teams. When a developer executes the test, the execution trace may go across the boundaries of multiple dependencies and create cross-project failures (CPFs). Typically, the client and third-party developers do not share dependencies or the test environment, which makes reporting CPFs more challenging. None of the existing techniques is able to produce an ideal CPF report that is simultaneously executable, readable, and concise. In particular, code portions generated from program slicing techniques [13, 90] are typically not compilable or executable because they do not consider environment dependencies beyond source code. Software debloating [22,38] techniques are applied directly to binary code, so the corresponding source code is difficult to acquire. Packaging the whole project will guarantee failure reproduction but lead to high storage and network transmission costs, as well as noise during debugging. This creates the trilemma of the cross-project failure report, as shown in Figure 3.1.

To solve the CPF trilemma, we designed a framework called PExReport in Chapter 2. We implemented PExReport in Maven [1] build system to create a tool called PExReport-Maven for solving the trilemma in the Java ecosystem. In this chapter, we reveal additional implementation details with a concrete example, describe the tool's anticipated users, and propose a plan for future study based on user feedback.

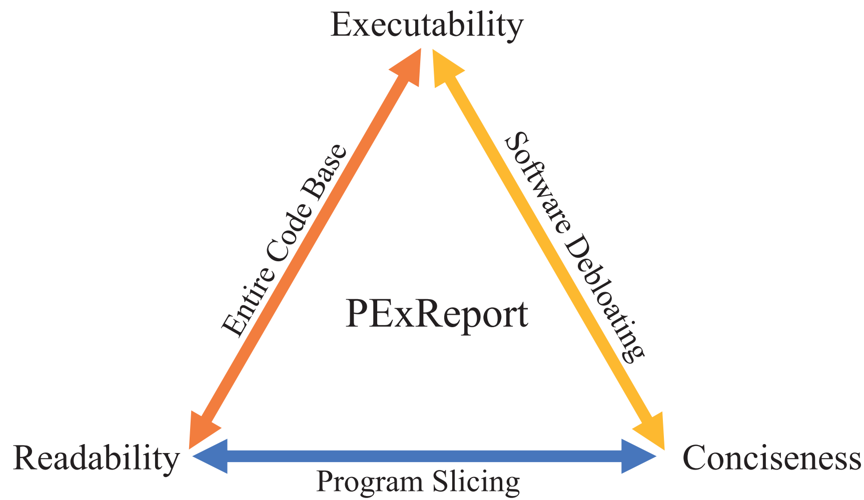



**Figure 3.1**: Cross-project failure report trilemma

## 3.2 PExReport-Maven Description

### 3.2.1 Overview

Figure 3.2 shows an overview of PExReport-Maven's workflow. The input to PExReport-Maven is a failed test case from an existing Maven build environment, and the output is a pruned stand-alone executable failure report. PExReport contains two major phases: the collection phase to collect information about necessary Java source code, JAR dependencies, and the Maven build environment for re-executing the failed test case, and the reconstruction phase to reconstruct a stand-alone Maven project for the failed test case based on the collected information. The failure traces are the collected information from a failed test case in the first phase. The arrows show the information flow between each component of PExReport. The reconstructed Maven project for failure reporting will be automatically validated by checking whether exactly the same error messages are triggered as in the original failure. Once the project is validated, it can serve as a reproduction package of the original failed test case and will be reported as a pruned executable

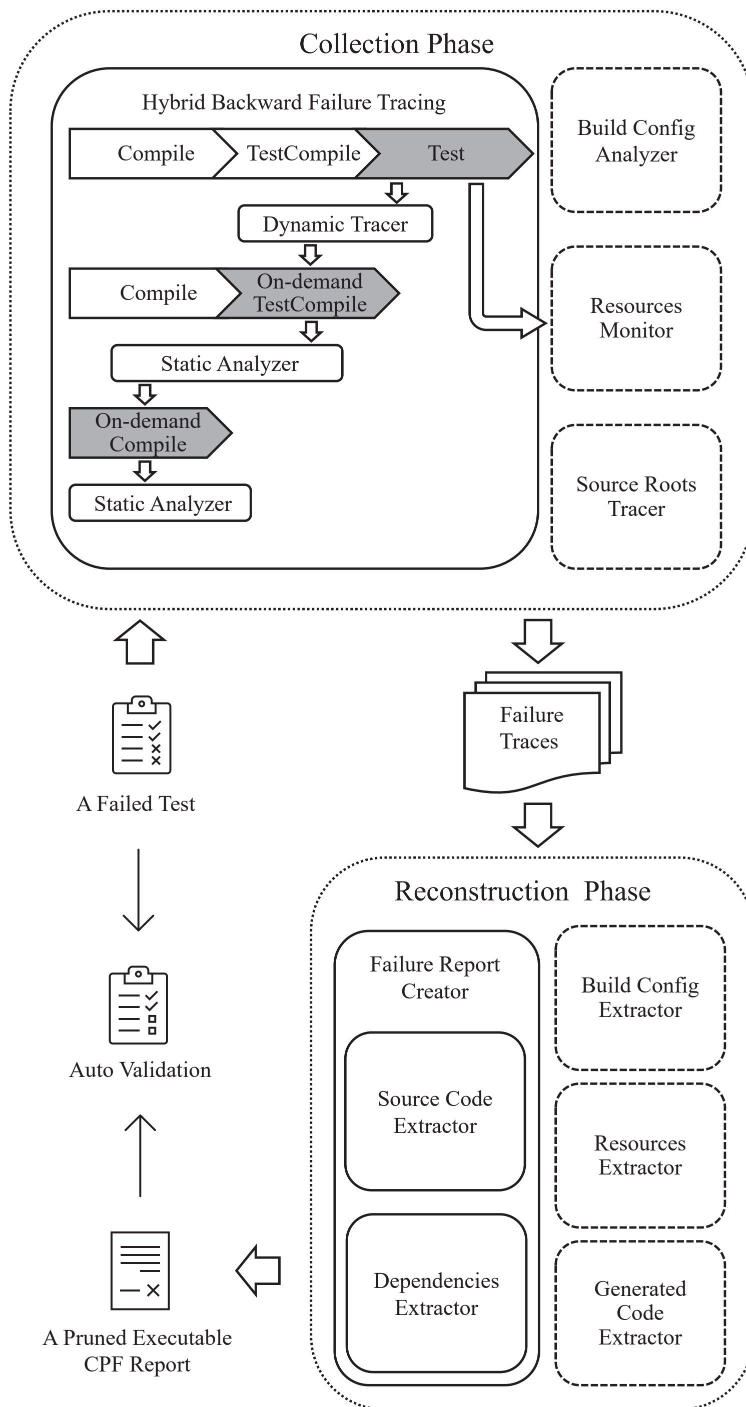



**Figure 3.2**: PExReport-Maven workflow

failure report to the developers of dependency code.

### 3.2.2 Core Maven Phases for the Lifecycle of Test Failures

PExReport-Maven is designed based on the core phases of the Maven build system involved in the lifecycle of test failures. [12] These phases are generally executed in the order they are presented, but some phases may be omitted if not required for the current project.

- **GenerateSources** (optional): generate any source code for inclusion in compilation.
- **ProcessResources** (optional): copy and process the resources into the destination directory.
- **Compile** (mandatory): compile the source code of the project.
- **TestCompile** (mandatory): compile the test source code into the test destination directory.
- **Test** (mandatory): run tests using a suitable unit testing framework.

The three mandatory phases, namely `Compile`, `TestCompile` and `Test`, are the basic steps to reproduce a test failure used by Maven. We noticed that `compile` and `test-compile` are both used to compile source code. The main reason is to make the software source code independent of the test source code. Code generation and resource management are quite popular in complex real-world Java applications. Although they are optional tasks, we should not underestimate their role in reproducing real-world failures.

Figure 3.3 is a concrete example that shows the lifecycle of a Java test failure in the Maven build system. This example only involves sample Java source code and JAR dependencies. The GenerateSources and ProcessResources are excluded for simplicity.

- **Compile**: Java compiler compiles all source code in the left rounded rectangle (red) by referencing JARs.
- **TestCompile**: Java compile compiles all test source code in the middle rounded rectangle (blue) by referencing classes of *Compile* task and test JARs.

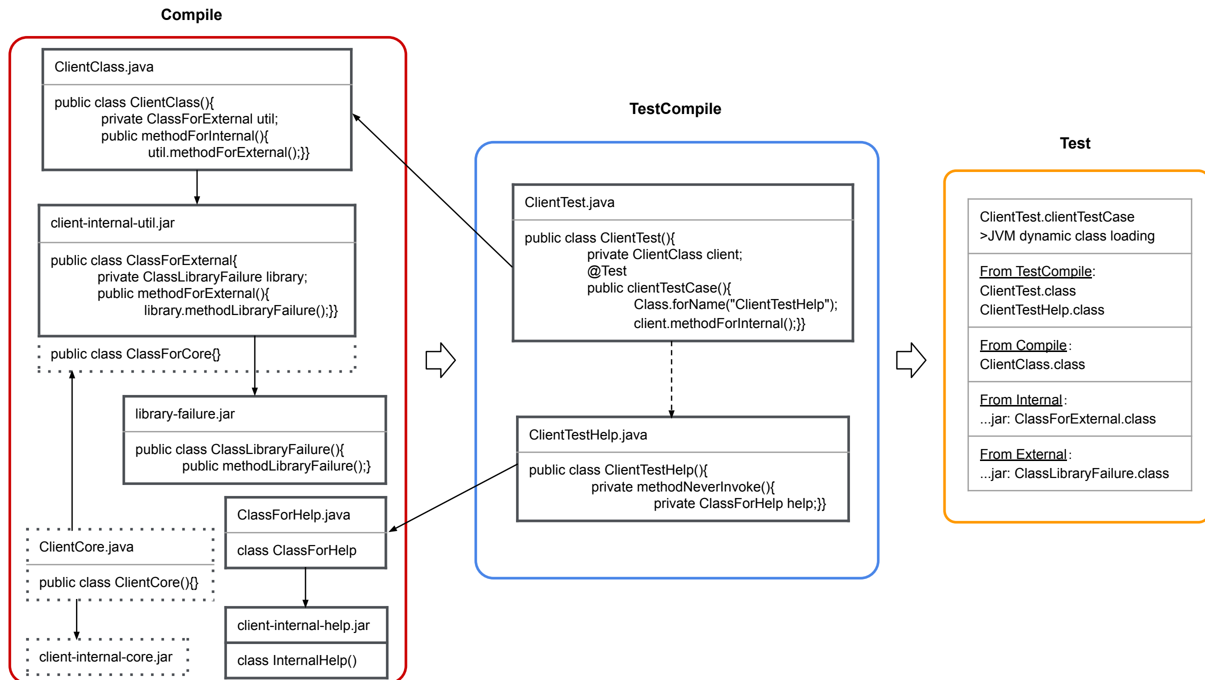



**Figure 3.3**: Maven test failure example

- **Test**: In the right rounded rectangle (orange), the JVM runs *ClientTest.clientTestCase* test by loading classes from *Compile* and *test* tasks.

### 3.2.3 Hybrid Backward Failure Tracing

In the collection phase, the base component is hybrid backward failure tracing, a three-step analysis that traces failure-related Java source code and JAR dependencies (Java classes or Bytecode). This component compiles and executes the failed test case in its original Maven build environment, and tracks Java class usage at *Test*, *TestCompile* and *Compile* core Maven phases. These three phases are essential for any test code to be compiled and executed, as shown in Figure 3.3.

The workflow of hybrid backward failure tracing is shown in the top part of Figure 3.2. In build process order, the compiled Java classes in *Compile* task are provided to the succeeding *testCompile* task as dependencies; after that, the compiled Java classes of *testCompile* are loaded to *test* task for execution. The build process must be irreversible to ensure that the Java classes that compile before them never depend on those that compile after them. If a test failure occurs,

all the failure-related Java classes can be tracked in the reversed build process order. Since failure happens at the end of the build process (*test* task), backward tracing can be performed to obtain the failure dependency tree.

The detailed tracing of the failure example in Figure 3.3 is shown in the following:

- **Round 1**: executes *Compile* (all source) and *TestCompile* (all test source) tasks, then run *ClientTest.clientTestCase* in *Test* task. The dynamic tracer component records all dynamically loaded classes in the *Test* task, accordingly.
  - Input: *ClientTest.clientTestCase*
  - Output Traces:
    * From TestCompile: *ClientTest*, *ClientTestHelp* (dynamic)
    * From Compile: *ClientClass*
    * From Internal: *ClassForExternal* (client-internal-util.jar)
    * From External: *ClassLibraryFailure* (library-failure.jar)
- **Round 2**: executes *Compile* task, and *TestCompile* (*ClientTest.java*, *ClientTestHelp.java*) task. The static analyzer component records all referred classes in *TestCompile* task, accordingly.
  - Input: Traces from *Round 1*
  - Output Traces:
    * From TestCompile: *ClientTest*, *ClientTestHelp* (dynamic)
    * From Compile: *ClientClass*, *ClassForHelp* (static)
    * From Internal: *ClassForExternal* (client-internal-util.jar)
    * From External *ClassLibraryFailure* (library-failure.jar)
- **Round 3**: executes *Compile* task (*ClientClass.java*,

*ClassForHelp.java*) task. The static analyzer component records all referred classes in *Compile* task, accordingly.

- Input: Traces from *Round 2*
- Output Traces:
  * From TestCompile: *ClientTest*, *ClientTestHelp* (dynamic)
  * From Compile: *ClientClass*, *ClassForHelp* (static)
  * From Internal: *ClassForExternal* (client-internal-util.jar ), *InternalHelp* (client-internal-help.jar)
  * From External: *ClassLibraryFailure* (library-failure.jar)

### 3.2.4 Enhancement Components

In addition to the base component (hybrid backward failure tracing), our tool also consists of three enhancement components to further reconstruct a reliable build environment.

- **Handling of the build configuration.** In Maven, all work is done by plugins, but most plugins are irrelevant for reproducing failure. The Build Configuration Analyzer fetches the *effective build configuration* (*help:effective-pom* [10]), which Maven uses to build the failed test at run time. The effective POM gathers all build configuration values scattered in all build configuration files and resolves configuration value overwriting among multiple configuration files based on the *nearest definition* for resolving dependencies. However, the *effective POM* is still redundant for reproducing the failed test compared with the required configuration values of the three core phases. Since phases in the Maven build process are performed by various plugins and the plugins can be attached to different build phases (e.g., the Java compiler plug-in can be attached to the Compile and TestCompile phases), we further leverage the attachment relationship to identify all the plugins that are attached to the three core build phases. It also excludes code-style checking and analysis plugins because they do not directly affect compilation or testing. The build configuration extractor collects the configuration information from these plugins and updates the failure reproduction package.

- **Handling of resource files.** In the Linux kernel, the inotify API provides a mechanism for monitoring filesystem events. We use Pyinotify [7], a Python module that wraps the inotify API, to monitor all resource activities under the project scope. As shown in Figure 3.2, the resource monitor only outputs files accessed at *Test* phase because the *Process resources* phase copies all resource files to the target folder. For necessary resource files accessed during the compilation process (e.g., templates of generated source code), we specially handled them by the Generated Code Extractor. Our resource monitor also ignores all files and directories generated during the build or test process because these files should not be included in the reproduction package (unnecessary and causing path conflicts). For the files copied to the target location from source locations, we do not consider them generated files because the Resource Monitor can trace back to their source copies in the original project.

- **Handling of source code generation.** The Maven build process may generate new source code in various ways, such as by creating code from template files, generating parsing code from syntax or XML files, or even directly fetching source code from remote locations. Furthermore, code generation is often implemented in third-party tools and plugins. To handle such high flexibility in code generation in a general way, we omit the code generation process and directly include the generated source code in the failure reproduction package. At *GenerateSources* phase, the build tools use source code root paths (Source Roots) to locate all original and generated source code. In the Collection Phase, our Source Roots Tracer tracks all the accessed source code root paths from the debug information of compilation. Next, the Generated Code Extractor utilizes the paths to identify the generated source code and excludes all original source code. In addition, the build tools may also generate some source code from code annotation processing. Our Generated Code Extractor excludes such code because it will cause compilation conflicts.

## 3.3 Envisioned Users

The following users can make use of PExReport-Maven:

- Developers of Java applications who use the Maven build system and third-party libraries (open-source or proprietary) can use our tool to report test failures.
- All software defect datasets suffer from software breakages that are mostly related to software dependencies. Dependency caching can effectively prevent software breakages and ensure long-term reproducibility. [101] Because PExReport-Maven can preserve Maven build environments (e.g., required dependency caching) and necessary portions of Java projects for reproducing CPFs, our tool is useful for researchers who want to share their CPF datasets with pruned build environments. For example, our tool has been validated on the Sensor dataset, which is a dependency conflict dataset. [87]

## 3.4 Using the Tool

### 3.4.1 Download

PExReport have been uploaded to a open source repository on GitHub. It can be download use following `git` command.

```
git clone https://github.com/wereHuang/PExReport-Maven
```

### 3.4.2 Create a Working Environment

1. Use a Linux machine; the tool is verified in Ubuntu 22.04 LTS. This tool also supports containerized environments with Linux host machines.
2. Install Maven 3 and pip for Python3.

   ```
   sudo apt update
   && sudo apt install maven python3-pip -y
   ```

3. Install Python packages using `requirements.txt` under the root folder of PExReport-Maven.

```
sudo pip install -r requirements.txt
```

4. Install the customized Maven Archetype for PExReport-Maven.

   Change the working directory to pexreport-archetype, then execute:

   ```
   mvn clean install
   ```

5. Install Java 8 (required for test)

   ```
   sudo apt install openjdk-8-jdk -y
   ```

### 3.4.3 Command Line Usage

1. A CPF can be triggered by the following Maven command:

   ```
   mvn clean test -Dtest=TEST_NAME
   ```

2. Pass the test name, path of the source project, group ID for internal dependencies, and report name of output to `per.py`:

   ```
   per.py -n TEST_NAME -s SOURCE -g GROUPID -t TARGET
   ```

3. The CPF report named TARGET will be created, which is a reproduction package for the original CPF. In the directory of the generated CPF report, use the same Maven command that triggered the original CPF to reproduce.

## 3.5 Evaluation and Studies

Our tool has been evaluated on executability and conciseness in Section 2.4. The evaluation of 74 software project issues with 198 CPFs achieved a high reproduction rate for 184 out of 198 CPFs, with an average reduction rate of 72.97% on Java classes.

We plan to further collect user feedback to understand how much our tool can help developers report real-world test failures. This study could also assist us in answering the question, "How could the community enhance this tool?"

## 3.6 Conclusions

An executable test case is one of the most desirable features of failure reports. When reporting *cross-project failures (CPFs)* to library developers, a test case is even more helpful because code is a natural way to describe interactions between library code and client code. In this chapter, we present PExReport-Maven, a tool to automatically create pruned executable CPF reports for reporters using the Maven build system, and solve the CPF report trilemma in the Java ecosystem. The future study will be conducted based on user feedback from using this tool for real-world test failure reporting.

# CHAPTER 4: BUILD ISSUE RESOLUTION FROM THE PERSPECTIVE OF NON-CONTRIBUTORS

## 4.1 Introduction

Open-source software (OSS) frequently requires building from source code. This "build" process often encompasses several steps, including compiling the source code, resolving dependencies, packaging, testing, performing static analysis, generating documentation, and preparing for deployment. The primary roles of those engaged in these building activities are those of project contributors, who manage repositories and commit code. However, there are also numerous non-contributors who need to build the project for their own goals. Examples of such non-contributors are as follows:

**Users:** Not all OSS projects provide pre-built installation packages. Even when available, these packages may not be compatible with a user's operating system (OS) or local environment. So some users may need to build projects from source code in their local environments. Advanced users might also prefer non-default configurations, utilize new features not yet included in a stable release, or compile directly from the source code for security reasons, such as in blockchain scenarios [8].

**Learners:** Individuals who are new to OSS development and are in the learning phase also engage in building OSS. They may start by compiling and experimenting with the source code to understand the project structure and functionality better. This hands-on experience is essential for their growth as OSS contributors.

**Potential contributors:** Individuals who wish to propose feature improvements or bug fixes for OSS must also build the project initially. This step is necessary before they can validate their revisions and submit a pull request. However, because OSS often has diverse build environments, these potential contributors frequently lack the required local build setups. Setting up these environments themselves can pose its own set of challenges.

In this chapter, within the context of the OSS build process, we use the term "non-contributor" to refer to roles that lack familiarity with the build environment of the target OSS and need to set up their local environment. These roles may have varying degrees of background knowledge about the target OSS, but they all share the common objective of building the OSS from its source code.

Since non-contributors are often unfamiliar with the build environments of the OSS projects they aim to work on, many encounter challenges in this process. On Stack Overflow (SO), filtering by the keywords "not build" yielded 6,760 out of 43,140 (15.7%), 7,374 out of 95,275 (7.73%), and 10,336 out of 73,559 (14.1%) relevant SO questions for three well-known open-source projects: Tomcat, Hibernate, and OpenCV, respectively [66]. Since OSS contributors are unlikely to address internal build issues on Stack Overflow (preferring internal issue tracking systems like GitHub Issues or JIRA), it is reasonable to infer that most of these SO questions are from non-contributors experiencing build issues in their local environments. These numbers highlight the prevalence and difficulty of build issues faced by non-contributors. Another illustrative example is evident in computer science (CS) classes, where students frequently encounter initial project hurdles. These difficulties typically arise from their inability to build or install the frameworks or tools on their own systems.

Despite the prevalence of build issues from non-contributors, existing research has largely overlooked these challenges, focusing primarily on the perspective of contributors, such as the developers of the project being built. For instance, extensive studies have been conducted on the effort required to maintain build scripts [58,59], the categorization and distribution of bugs in build scripts [16, 77, 93], the patterns of fixing build scripts [55, 100], and how build failures occur in new integration scenarios such as Continuous Integration (CI) chains [102] or Docker environments [92]. It is important to note that build issues experienced by non-contributors may have different root causes compared to those experienced by contributors. The latter are often caused by flaws in the build scripts or code of the project being built, while the former are likely related to the local environments of the non-contributors, assuming that most OSS projects are released with validated build scripts. One potential reason why build issues experienced by non-contributors are not

studied may be due to data collection challenges. Most existing studies gather data from version histories and build logs, while non-contributors' experienced build issues are never systematically tracked or recorded.

In this chapter, we present a study to investigate build issues experienced by non-contributors, aiming to answer the prevalent question, **Why does the OSS project not build on my machine?**. Our study tracks the behaviors of senior-year and graduate computer science students as they undertake build tasks for various OSS projects. We believe these students are representative subjects for our study, as they currently occupy learner roles for the target OSS projects and could potentially transition to user or potential contributor roles with appropriate guidance. All these students have a certain level of background in information technology (most have some internship and working experience), which positions them for a more advanced user role in utilizing the target OSS projects. We have designed these build tasks as a course project for a cross-listed software engineering course, where we instructed 31 students to build 12 different OSS projects in 6 programming languages (PLs) in order to explore the symptoms and causes of build issues experienced by non-contributors.

## 4.2 Related Works

The research efforts most related to our research are user studies on software build tasks. Kwan et al. [50] studied developers from IBM to find out whether team composition and coordination may have an impact on software build success. Dawns et al. [29] studied the operation of a build team to evaluate an automatic build management tool that enforces the build failure handling process. Philips et al. [67] studied software building teams at Microsoft and found that most challenges are on the social aspects of the team. Kerzazi et al. [47] studied 3,214 software builds and found that 17.9% of builds failed and more than 300 man-hours were cost to fix them. Hilton et al. [40] performed interviews with 16 developers to find out their opinion on whether the CI process may enhance software productivity. Vassallo et al. [83] performed a study with 17 participants to find out how well developers can take advantage of BART, a tool for summarizing build failure reasons.

Different from our research, all these studies are from the perspective of project managers or senior developers instead of non-contributors.

Besides user studies, there have been a lot of empirical studies on software build history and build failures. Mcintosh et al. studied the version histories of proprietary and open source software projects to estimate the effort required to repair build scripts [58] and to correlate effort with type of bulid systems [59]. Xia et al. [93] performed studies to summarize bugs in software build systems. Zhao et al. [100] studied build failure reports in five OSS and found that build failures take much more time to fix than others. Barrak et al. [16] studied how build failures are correlated with code smells in build scripts and code. Licker and Rice [53] investigated the incorrect rules in build scripts by using a mutation testing approach. Wu et al. [92] studied the characteristics of build failures in the context of docker environments. These existing studies focus on build failure logs and build failure fixes in the commit history. In contrast to previous studies, our research collects and analyzes the entire process of completing multiple OSS build tasks in a local environment, taking into account the system environment factors that influence the build process.

## 4.3 Study Design

We conducted a study that gathered data on build issues from 31 participants involved in 12 OSS projects. This provided us with a comprehensive understanding, enabling us to identify key symptoms and examine their resolution process. We aim to answer the following research question:

**What are the common symptoms of build issues experienced by non-contributors during the build process, and to what extent can these symptoms be mitigated?**

### 4.3.1 Participants and Tasks

Our participants were 31 students enrolled in the same software testing course. This course is a cross-listed elective for senior-year undergraduates and graduate students, with software engineering as a prerequisite. The course, taught by two of the authors at a university, focuses on software testing approaches, test planning, test case design, and build systems with CI/CD concepts. In our

Table 4.1: Selected OSS projects

| PL | Project | No. Participants | No. Issues |
|---|---|---|---|
| C++ | opencv/opencv | 3 | 30 |
| | tensorflow/tensorflow | 3 | 38 |
| Go | gohugoio/hugo | 4 | 32 |
| | kubernetes/kubernetes | 2 | 18 |
| Java | elastic/elasticsearch | 2 | 18 |
| | spring-projects/spring-boot | 3 | 46 |
| JavaScript | electron/electron | 1 | 4 |
| | vuejs/vue | 4 | 36 |
| PHP | fzaninotto/Faker | 2 | 9 |
| | guzzle/guzzle | 1 | 15 |
| Python | pallets/flask | 3 | 26 |
| | scikit-learn/scikit-learn | 3 | 31 |
| **Total** | | **31** | **303** |

study, students shared many characteristics with non-contributors who only needed to build the released OSS without modifying the source code. None of the students have the experience to set up the specific build environments required by the OSS projects in our study. Almost all of the students will work or are already working part-time or full-time as developers, so studying their behavior could further help us understand the build issues when newcomers are onboarding.

As shown in Table 4.1, we selected 6 popular PLs from the top 15 used on GitHub. For each PL, we chose 2 of the top 10 most popular OSS projects on GitHub, ranked by the number of stars. We made our best effort to choose PLs and OSS projects with distinct real-world application scenarios. Participants picked a project from the 12 OSS projects on a first-come, first-served basis. We limited each project to three slots to ensure that each project had at least one participant. Participants were instructed to write a report documenting at least 10 non-trivial build issues. They were encouraged to resolve the issues if they could.

Students were required to complete the study as one of their major course projects. We designed a three-stage task list to help participants become familiar with the study process: 1) Setup. Participants were instructed to use the Google Cloud Platform (GCP) online console to set up a

GCP project with $50 education credits and enabled APIs for creating virtual machines (VMs). 2) Warm-up. Participants were instructed to create a warm-up VM instance. They then performed warm-up build activities to familiarize themselves with the logging and snapshot-taking processes. 3) Build OSS projects. Each of the participants picked one OSS project for their build tasks. They utilized log scripts, snapshots, and textual issue reports to document build issues encountered during the build process of the target OSS project.

The anticipated outcomes of the study included successfully compiling the OSS projects and passing all tests using the commands specified in the build tasks. Beyond the build outcome, participants' grades would be evaluated based on their effort invested in completing the tasks, as evidenced by snapshots and issue reports. This grading approach was designed to accommodate participants who might not be able to successfully build the OSS projects. As the conductors of the study, our role was to provide clarity on the objectives of the build tasks. Our study protocol underwent assessment and received approval from our local Institutional Review Board (IRB). All students were informed that they may withdraw their data after grading, ensuring their data would not be included in our dataset for this study or any future research activities.

### 4.3.2 Data Collection

We designed protocols to collect data throughout the build process. As participants performed build tasks on a virtual machine (VM), we used the following methods to capture all relevant data.

**VM state logging.** When encountering build issues, participants used a script to log command history, environment variables, and network information. This logging helped correlate system data with reported issues. Participants also logged the final VM state upon completing tasks, with all logs saved for later analysis.

**VM snapshots.** Snapshots captured the entire VM state at specific moments, including files, settings, and configurations. Participants manually took snapshots upon encountering build issues and upon task completion, though temporary environment variables could be lost in the process.

**Build issue report.** Participants documented build issues, including inputs, outputs, and solu-

tions, along with details about the build environment (e.g., language versions, tools). They summarized their process and provided feedback, offering insights into their problem-solving strategies.

## 4.4 Results and Analysis

We collected a total of 303 build issues from 31 build issue reports, along with 380 snapshots containing environment logs. One student changed from the "electron/electron" project to "vuejs/vue" due to low GCP credit. One student worked on the wrong project in Go. Eventually, for each OSS, there was at least one student and a maximum of four students. For each PL, there were at least three students and a maximum of six students. In the collected 303 build issues, the minimum number of issues reported was 4 for the "electron/electron" project, with only 1 student participant. The maximum number of issues reported was 46 for the sprint-boot project, with 3 participants. The number of reported issues was not correlated with only the number of participants but was also project-specific.

We implemented a qualitative analysis on these 303 build issues, utilizing the open coding procedure [75]. This procedure was executed by two of the authors. The first author coded all build issues, drawing from error messages and solutions reported within issue reports, and identified any borderline cases. Subsequently, the second author validated this coding and engaged in discussions about any borderline cases. In instances where conflicts arose, the first author leveraged corresponding snapshots and collected environment information to attempt to reproduce the build issue. Decisions were made based on the reproduced build results or the history available in the snapshot, with the two authors discussing and reaching consensus. Additionally, two authors were responsible for labeling the resolution state of the issues. If no solution was provided, the issue was labeled as *Unsettled*. However, if either of the two authors believed that a solution or workaround was provided, the issue was labeled as *Settled*.

Figure 4.1 illustrates the 14 distinct symptoms of build issues identified in our study. The light-colored bars represent unsettled build issues, while the remaining dark-colored bars denote settled issues. Out of the total 303 issues analyzed, 182 (60.1%) remained unsettled, highlighting the

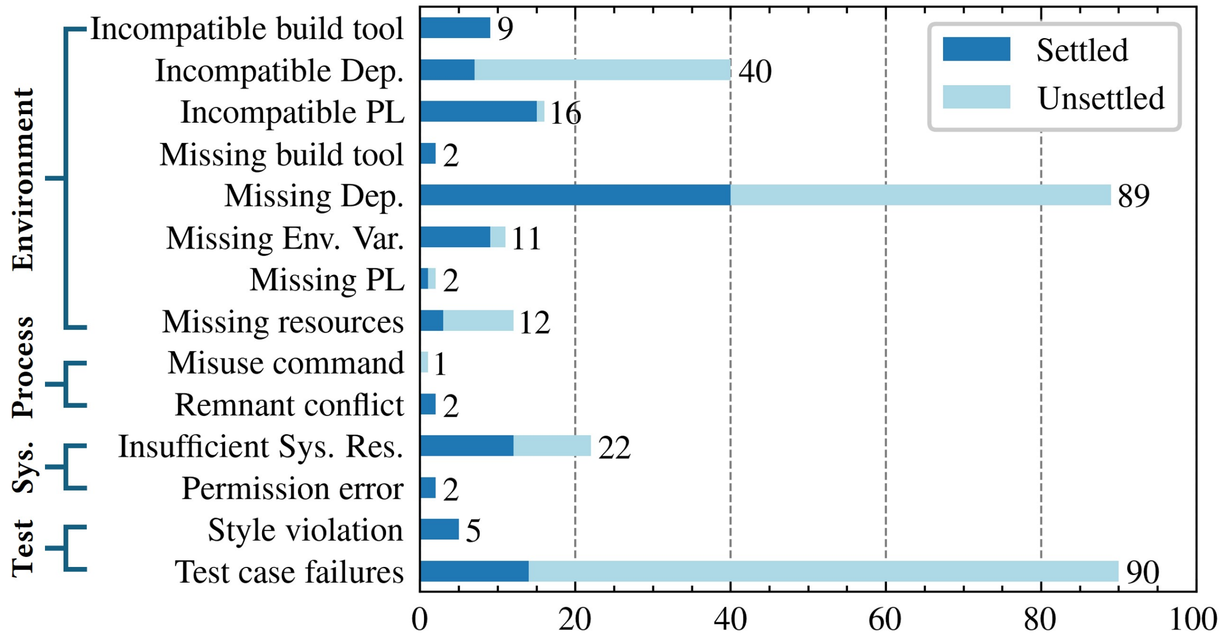



**Figure 4.1**: Symptoms experienced by non-contributors.

persistent challenges faced in addressing build issues experienced by non-contributors. To better understand the characteristics of these symptoms, we further categorized them into four broad categories: environment, process, system, and test-related issues. By grouping the 14 symptoms into these categories, we aim to provide a structured analysis of the underlying factors contributing to build issues. In the following sections, we examine each category, analyzing the symptoms and their implications for build issue resolution.

**Environment-related symptoms** include those related to incompatibility and missing components, which together account for the majority (181 out of 303) of build issues. Incompatibility-related symptoms, such as those involving build tools, dependencies (Dep.), and PL, occur when OSS projects cannot work with necessary components. Missing-related symptoms arise when build tools, dependencies, environment (Env.) variables (Var.), PLs, and external resources are not available in the current workspace. These two dependency-related symptoms have a high unsettled rate, a finding that aligns with studies of contributors-experienced build issues [55, 80, 82, 103]. Missing resources, common in evolving OSS systems, are challenging for non-contributors to resolve due to the difficulty in fetching external resources. The incompatible PL has a good settlement

rate, primarily because the error messages in selected projects are easy to identify, unlike those for incompatible dependencies. Missing environment variables may mask other symptoms, as those issues require specific environment variables. We observed during the coding process that many incompatibility symptoms can be converted to other symptoms as non-contributors seek solutions.

**Process-related symptoms** refer to build issues caused by participants' mistakes during key steps of the build process. For example, "misuse command" symptom can occur when participants fail to escape special characters in command options, and "remnant conflict" symptom can arise when participants neglect to clean up the build remnants. Resolving these issues can be complex if participants are unable to correctly interpret the error messages, and the difficulty can be compounded by the complexity of the required build process.

**System-related symptoms** refer to "Insufficient system (Sys.) resources (Res.)" and "Permission error". The insufficient system resources symptom ranks among the top four symptoms, as most of the selected OSS projects are real-world projects that require a larger amount of system resources than our participants anticipated. Moreover, non-contributors often lack an effective approach to estimating the system resources required before they perform the build activities. This could explain the high unsettled rate of 10 out of 22. This symptom is also common since VMs created by our participants typically do not have many redundant system resources. For instance, participants experienced system crashes due to disk space exhaustion, followed by failures to reconnect to VMs. They did not realize the issue until the system resources were exhausted. Another symptom is the "permission error", which is caused when underlying software requires higher permissions than it currently has. In our study, we found that *Docker* commands often need superuser permissions. This is a common issue among OSS projects that rely on *Docker* [9].

**Test-related symptoms** refer to "Style violations" and "Test case failures". Style violation test failures typically occur when a project does not adhere to a certain standard. Participants can often find a workaround to bypass this issue. However, it is important to note that the root cause of this symptom may be due to compatibility issues. The most common symptom in our study is the "Test case failures" symptom, which also has a high unsettled rate. This aligns with existing

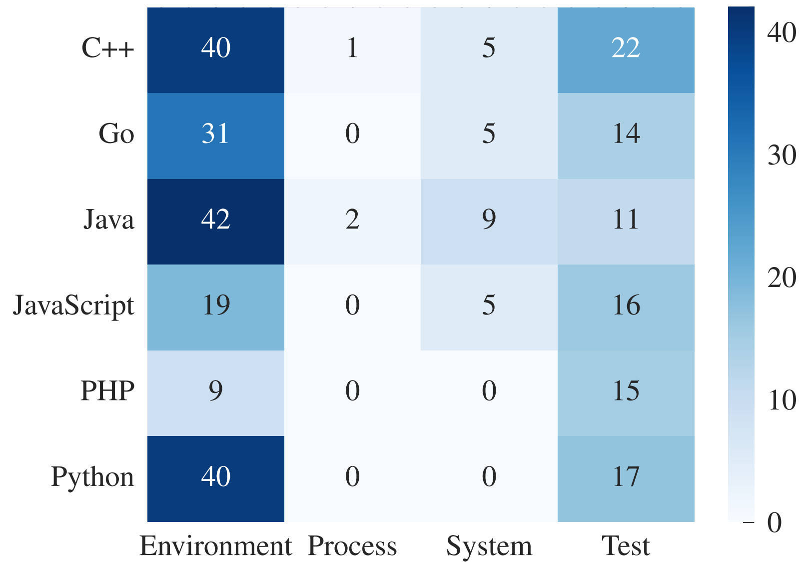



**Figure 4.2**: Distribution of categories on PLs.

studies [69, 84]. Notably, some test failure messages are difficult to interpret, making it hard to identify the root cause. The symptom could be the result of other underlying issues.

The distribution of these categories among the participants is as follows: "Environment" was experienced by 29 participants (93.6%), "Process" by 2 participants (6.5%), "System" by 10 participants (32.3%), and "Test" by 24 participants (77.4%). Figure 4.2 illustrates the distribution of categories across different PLs. Environment and test related symptoms are present in all PLs. In contrast, system-related symptoms are project-specific, which can be attributed to the fact that lightweight projects do not consume significant system resources. Upon further investigation of process-related symptoms, we observed that a participant encountered both "Misuse command" and "Remnant conflict" issues in a Spring Boot project. Despite its infrequent occurrence among the issues observed, the "Remnant conflict" issue was identified in two projects, one written in Java and the other in C++.

Compared to the study by Lou et al. [55], our study on the symptoms of build issues shows that non-contributors still frequently experience the same symptoms as contributors. However, these are mostly limited to issues caused by a lack of installation, version incompatibility errors, and environment variable issues. We also observed additional symptoms not covered by previous studies

that are related to participants' actions, such as "Command misuse" and "remnant conflicts". As the 39.9% settlement rate shows, participants found it difficult to mitigate most of these frequent symptoms as well as identify the root causes of build issues. The strategies for resolving these build issues tend to be more straightforward due to the non-contributors' lack of specific project knowledge and experience. This limitation can make them more susceptible to encountering certain build issues.

## 4.5 Discussion and Future Work

Although containerization holds promise for addressing certain build issues, the lack of accessible container environments in non-contributors' local systems presented a challenge for many OSS projects, including most of those in our study. A threat to the validity of the findings is the subjectivity involved in determining whether a build issue is trivial. This judgment depends on several factors, including the individual's level of knowledge and local systems. Our future research will involve more participants and focus on a specific scope to explore the correlation between various factors and our findings. Additionally, we observed inconsistencies in reporting, such as misinterpreted test results and misaligned snapshots. Therefore, incorporating more monitoring measures into the process could significantly mitigate these potential threats.

## 4.6 Conclusion

This chapter investigates 303 build issues experienced by 31 non-contributors. We found that non-contributors often struggle to resolve issues due to limited symptoms. Our study provides valuable insights into build issue resolution from the perspective of non-contributors, highlighting the importance of understanding their behavior and challenges. This work lays the foundation for further research in this area, with the ultimate goal of improving the experience of non-contributors in dealing build issues.[1]

[1]This work is supported in part by NSF projects 1736209 and 1846467.

# CHAPTER 5: AN EXPLORATORY STUDY ON BUILD ISSUE RESOLUTION AMONG COMPUTER SCIENCE STUDENTS

## 5.1 Introduction

It is highly beneficial to integrate open-source software (OSS) projects into a software engineering (SE) curriculum for three reasons. First, OSS exposes students to real-world software development practices. By inspecting the software evolution and reading developers' discussions on technical issues, students can improve their understanding of industrial challenges and become better prepared for careers in the software industry [32]. Second, OSS offers opportunities for students to gain hands-on experience when they use and modify the software based on local needs and preferences [49], collaborate with a global community of developers, and adopt project management tools like version control as well as issue tracking. Third, the educational usage of OSS bridges the gap between theoretical knowledge and practical application, enabling students to see how theoretical knowledge is applied in real life [68] and to solve real problems with book knowledge.

While the integration of OSS projects into SE curriculum brings great opportunities, it also poses challenges to Computer Science (CS) education. One big challenge is that students often fail to build the provided OSS on their local machines. Build failures can be very detrimental to students' learning outcomes. This is because as students encountered build failures at the beginning of OSS-based class activities, they might spend tremendous time and effort trying to resolve those issues and neglect teachers' further instructions on designated tasks. As a result, they might lose the valuable opportunities to read, modify, or execute the same software as other students do and become unavailable to collaborate with other students.

In the research area of CS education, educators and researchers proposed various ways of integrating OSS into CS curriculum [26, 32, 49, 62, 68], discussed the benefits and challenges brought by the OSS-integration into courses [74], and measured students' contributions to OSS [30]. In particular, Salerno et al. [74] conducted a large-scale study on the impact of OSS-based courses,

revealing that 83% of the students faced technical challenges while contributing to OSS projects. While only 2% of students anticipated hurdles in setting up the local environment, a notable 9% of students eventually reported facing such issues. This discrepancy suggests that students tend to underestimate the challenge of addressing build issues. None of the existing work characterizes the build issues CS students frequently encounter in OSS-based courses, let alone suggest intervention to help address those issues.

In the research area of build issues, some researchers conducted user studies to explore the influencing factors (e.g., tool usage) for software build results [29, 40, 47, 51, 67, 83]. Other researchers examined relevant artifacts (e.g., build logs and bug fixes) to characterize various aspects of software build, including developers' efforts in resolving build issues [58, 59], root causes for build failures [16, 92, 93], and fixes [55, 100]. However, these studies are only vaguely related to the build issues faced by CS students for two reasons. First, prior work studied build issues from the perspective of developers or project managers who owned the software-to-build, rather than from the perspective of CS students who are new to the software-to-build. Second, prior studies were based on either self-retrospection or artifacts of experienced software practitioners, instead of based on the experience of CS students in class. Thus, many of the research findings by prior work are unlikely to find evidence in the build issues faced by students.

In Chapter 4, we conducted a pilot study to investigate the symptoms and causes of build issues experienced by non-contributors (e.g., users, learners, and potential contributors). The findings highlight specific build issues that are challenging to resolve and emphasize the need for further study to understand non-contributors' behavior.

To overcome the limitations of prior work, in this chapter, we introduce an exploratory study to investigate the build issues faced and resolved by CS students. The study has two phases, which tracked the behaviors of 55 CS students as they undertook 330 OSS build tasks in an advanced SE course. In Phase I, each student of the course in 2022 was assigned six build tasks. Phase I collected and analyzed the 198 build results from 33 students to characterize the symptoms of build issues, the students' resolution attempts, and the effectiveness of those attempts. Phase II

introduced an intervention to help students build software. Namely, the intervention emphasized providing additional key information sources, so that students can refer to them initially when completing build tasks. As with Phase I, Phase II assigned each student of the course in 2023 with six build tasks. By gathering and analyzing the 132 build results of 22 students, Phase II compared students' results with those from Phase I, to assess the intervention effectiveness.

This work makes the following major contributions:

- We presented a comprehensive analysis of the build issues encountered by 55 CS students, offering insights into the common challenges faced during students' OSS builds.
- Our study revealed the resolution strategies frequently applied by students, although some of these strategies often led to build issues that are challenging to resolve.
- We introduced an intervention method (i.e., providing key information sources initially to bridge the knowledge gap), which demonstrated effectiveness in helping students better build OSS.
- Our study offers practical recommendations for various stakeholders in the OSS and CS education communities, aiming to enhance the overall experience of integrating OSS into the SE curriculum.

## 5.2 Methodology

Our exploratory study was designed and executed in two distinct phases to comprehensively understand build issue resolution among CS students. Each phase had a unique objective and contributed valuable insights to the overall study.

### 5.2.1 Resolution Strategy Study

The initial phase of our study is the resolution strategies study, focused on understanding the symptoms of unresolved build issues and analyzing strategies that CS students have employed to successfully address them. We collected data from 33 CS students across 4 OSS projects and 2

programming languages (PL) in the 2022 software testing class. We evaluated the effectiveness of these strategies and their limitations. In addition, we explored the impact of prior experience on the build success rate. This phase provided us with a deeper understanding of students' resolution strategies. RQ1-5 will be answered in this phase.

- **RQ1**: As learners of OSS, are students able to accurately interpret the build issues that block build tasks?
- **RQ2**: How often do build tasks end with unresolved issues?
- **RQ3**: What are the common symptoms of unresolved build issues, and to what degree can they be mitigated?
- **RQ4**: What strategies have students employed to successfully address these unresolved build issues, and are these strategies effective?
- **RQ5**: Are there any prior experiences that may correlate with students' ability to resolve build issues?

### 5.2.2 Intervention Study

Phase II is the intervention study. The insights gathered from the preceding phase guided the development of intervention, which were aimed at enhancing the success rate of CS students on build tasks with proactive measures. The intervention was performed on 22 CS students registered for the same course in 2023. We assessed the impact of the intervention on the success rate and collected feedback to identify any potential mismatches. RQ6 will be answered in this phase.

- **RQ6**: Given the resolution strategies employed by students, what proactive measures can we introduce to effectively enhance their performance in resolving build issues?

### 5.2.3 Participants and Tasks

Our participants were all CS students registered in the Software Testing course in 2022 ($n = 33$) and 2023 ($n = 22$). This course is a cross-listed elective for senior-year undergraduates and

graduate students, with Software Engineering as a prerequisite. The course, taught by two of the authors at a U.S. university, focuses on software testing approaches, test planning, test case design, and build systems with CI/CD concepts. In our study, students only need to build the released OSS and do not need a background to modify the software.

Students were required to complete the study as one of their major course projects. We designed a three-stage task list to help participants become familiar with the study process:

1. **Setup:** Participants were instructed to use the Google Cloud Platform (GCP) online console to set up a GCP project with $50 education credits and enabled APIs for creating VMs. Then, custom images were shared with participants on GCP. Additionally, we obtained admin permission to access participants' GCP projects for further data collection.

2. **Warm-up:** Participants were instructed to use the warm-up custom image to create a VM instance. They then performed warm-up build activities in the VM, such as executing a "Hello World" build script. Participants learned to use the preinstalled script tool to log the VM state and then write a textual build report. They were also asked to complete a survey to provide feedback to researchers.

3. **Build OSS projects:** Participants were assigned OSS projects as their build task. They used preinstalled scripts and textual issue reports to report issues encountered while building and executing tests of the assigned project. At the end of this stage, participants completed a survey to share their feedback.

All these OSS projects were in the release version and had been verified by the two authors who taught this course. The anticipated outcomes included compiling the assigned OSS projects and passing all tests using the commands specified in the build tasks. In addition to the build outcome, their grades would be evaluated based on their investment in completing the tasks through monitored VMs and issue reports. As the conductors of the study, our role was to help clarify the objectives of the build tasks. Our study protocol was assessed and approved by our local IRB. All

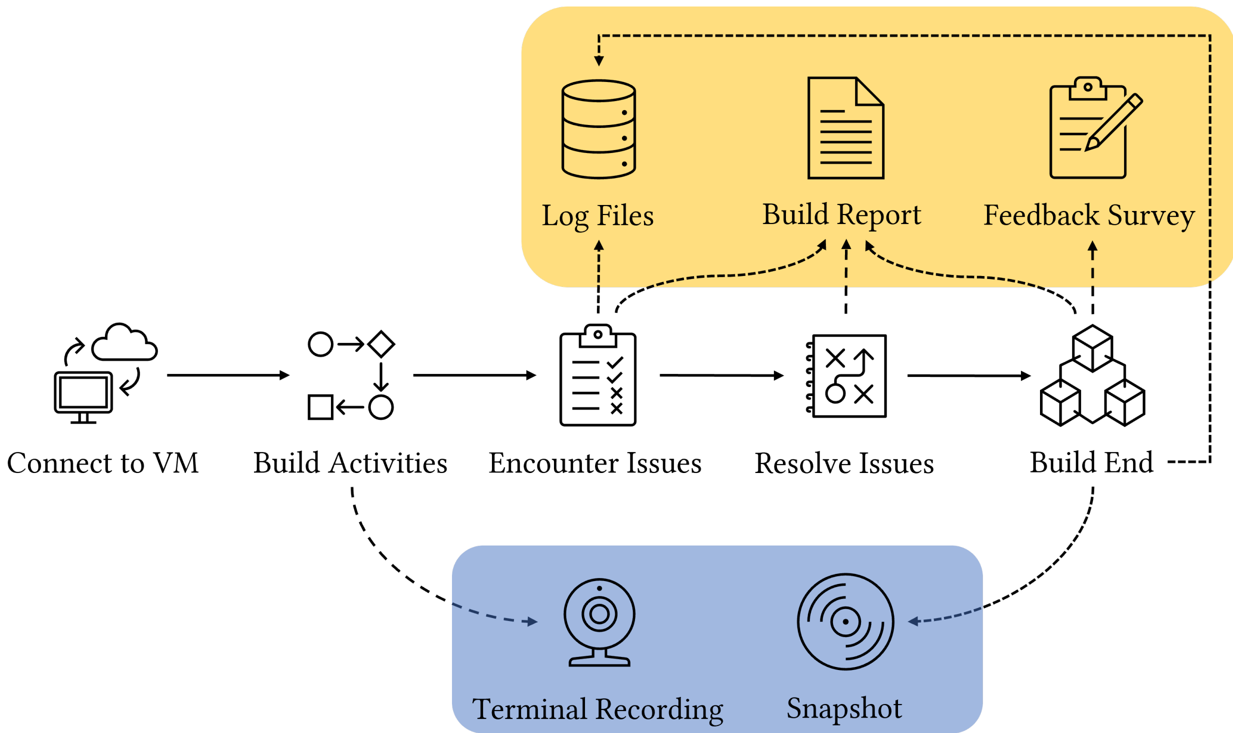



**Figure 5.1**: Data collection framework for build tasks

students were notified that they could withdraw their data after the grading so that their data would not be included in our dataset for our study or any subsequent research activities.

Over a span of two years, our study involved a diverse set of students in a variety of tasks. Specifically, the participation was marked by 39 and 25 students in each phase, respectively. In 2022, we had a total of 33 participants who completed all tasks related to our resolution strategies study. The following year, in 2023, the number stood at 22 for the intervention study. Despite the fluctuation in participant numbers, the prerequisites and design of our course remained consistent.

### 5.2.4 Data Collection

We designed a framework that collects data throughout the entire build process, as illustrated in Figure 5.1. While participants perform build activities on a custom VM and report build issues, all terminal activities are recorded. When a build task ends, the final state of the VM is preserved as a snapshot. The feedback survey is collected after all build tasks are completed. In Figure 5.1, we denote active data collection with the upper yellow rounded rectangle and passive data collection

with the lower blue rounded rectangle.

- **Custom VM:** In this study, we customized the official GCP OS images, which offer basic OS setups. We incorporated a custom script at OS startup to record terminal activities and provided a tool that logs the current state of the VM. These custom images ensure that all participants have consistent base build environments.

- **VM state logging:** After encountering build issues, participants log the current VM state using a script containing Linux commands to record command history, environment variables, and network information. This logging process, integral to the issue reporting, helps match logged data with reported issues. Additionally, participants log the final VM state upon completing the build tasks. All logs are saved in VMs for later retrieval, aiding in issue diagnosis and reproduction.

- **Terminal recording:** We opted for terminal recording as it captures all terminal activities, making the collected data easier to analyze. Using the *script* command in Linux [46], we integrated terminal recording into the VM's startup process. Once participants connect to the VM via SSH, the recording begins, capturing all screen outputs and keyboard inputs, including activities like file editing. While terminal recording lacks the ability to extract executable commands from inputs, it offers automated and comprehensive data collection without requiring manual initiation by participants. Despite this limitation, terminal recording facilitated playback and review of the entire build process, aiding in the analysis and reproduction of encountered build issues.

- **Snapshots of VMs:** In addition to terminal recording, another option for collecting VM data is taking snapshots. A snapshot captures the entire VM at a specific moment, including files, settings, and configurations. While useful for troubleshooting and reproducing issues, snapshots lack playback functionality and erase temporary environment variables. To address cost and practical concerns, we decided to utilize an automated script to capture snapshots only upon the build tasks reaching the final state, whether it be success or failure.

- **Build issue report:** Throughout the completion of all tasks, participants were directed to utilize the build issue report to document encountered issues during the build process, including inputs, outputs, and the solutions employed for resolution. Additionally, participants documented additional details of the build environment, such as program language versions and utilized tools. Upon completion of the build, participants summarized the entire process and provided their thoughts. The build issue report offers valuable insights into their decision-making and problem-solving strategies.
- **Feedback Survey:** After completing the build process and documenting the encountered issues and their corresponding solutions, we conducted a feedback survey to gather insights from participants. This survey focused on human factors affecting the build process, such as participants' familiarity with the PL, build system, and OS. Additionally, we collected self-reported build outcomes (success or failure). Insights collected from the comment section, which included obstacles and reflections, further aided in uncovering the challenges faced by participants during the build process.

After screening out participants who did not complete all tasks, we collected the following data for each phase:

1. **Resolution Strategies Study**: A total of 198 build results with snapshots containing terminal recordings and log files for the entire build process. In addition to 33 textual build issue reports with feedback surveys.
2. **Intervention Study**: A total of 132 build results with snapshots containing terminal recordings and log files for the entire build process and 22 textual build issue reports along with feedback surveys.

### 5.2.5 Qualitative and Quantitative Data Analysis

#### Resolution Strategies Study

To address **RQ1-2**, the first author manually verified the build outcomes reported by participants, cross-referencing them with terminal recordings, command histories, and environment information extracted from snapshots. If a build issue occurred that prevented participants from completing the build task, the error message was extracted, and it was labeled as *unresolved*. Subsequently, all snapshots from the final state VM were converted into running VMs to reproduce the collected error messages. This was done using an automatic script, with necessary adjustments made for specific build issues. If there was an identified inconsistency in reporting, we refined the error message and resolution state. If a build issue could not be reproduced, the original error message was retained. The last author validated the verified build outcomes and the extracted error messages. If there were any conflicts, a discussion was initiated until a consensus was reached. To address **RQ3-4**, two authors initially followed the open coding procedure [75] to label unresolved build issues with their root symptoms. These root symptoms were identified by examining the entire build process, which was based on terminal recordings and command histories. If any conflicts arose, further verification was performed on a running VM created from the snapshot, and a discussion was initiated until a consensus was reached. Next, based on the error messages, two authors counted the number of resolved build issues across all successful build tasks. We only include issues in successful build tasks to eliminate the uncertainty associated with chained failures, where a new failure might mask an old one. In addition, two authors identified common approaches that resolved these issues. To address **RQ5**, we first mapped the answers from the 5-point Likert scale of familiarity questions to an ordinal scale, ranging from 1 to 5 (with 1 representing "Not familiar at all" and 5 representing "Extremely familiar"). We then converted the build outcomes to a binary scale (1 for success, 0 for failure). Following this, we calculated the Spearman's rank correlation coefficient to investigate any potential relationship between prior experiences and build outcomes. The significance of the relationship was interpreted with the $p$-value.

**Table 5.1**: OSS projects in resolution strategies study

| Project | Version | PL | Build Sys. | Repository |
|---|---|---|---|---|
| Guava | 31.1 | Java | Maven | google/guava |
| Spring Boot | 2.6.12 | Java | Gradle | spring-boot |
| OpenCV | 4.6.0 | C++ | Cmake | opencv/opencv |
| Bitcoin | 23 | C++ | Autotools | bitcoin/bitcoin |

**Intervention Study**

To address **RQ6**, we performed the same verification process as in the resolution strategies study to obtain the verified build outcomes for the intervention study. We compared the build success rates before and after the intervention, and used the Chi-squared test ($\chi^2$) to determine if the two rates were significantly different. Furthermore, aside with a self-reported 5-point scale for helpfulness (ranging from 1-5, with an option for "not applicable"), we investigated the barriers mentioned in textual feedback survey during the intervention study to identify the impact of intervention.

## 5.3 Resolution Strategies Study

To answer RQ 1-5, we conducted a study on resolution strategies in 2022 and analyzed 198 fully monitored build results from 33 participants.

In a background survey conducted in 2022, we found that the top four PLs our participants were familiar with are Java (100%), Python (92.3%), C/C++ (76.9%), and JavaScript (64.1%). Given that Java was the default PL for other course projects and considering that C++ requires extra compilation during the build process, we selected Java and C++ as our target PLs. We selected OSS projects from GitHub [6]. After filtering out projects that were tutorials, involved mixed PLs, or were not in English, we chose the most popular projects for two different build systems for each PL. This resulted in four OSS projects, as shown in Table 5.1. We believe that the build processes and challenges of the most popular OSS projects on GitHub can be largely generalized to other projects using the same PL and build system. As the study by Lou et al. [55] observed the impact

of different OS on build issue resolution, we further selected three Linux distributions (Ubuntu 22.04 LTS, CentOS Stream 9, and openSUSE Leap 15.4) based on their popularity. We verified all projects on each OS to ensure they were ready to build and excluded some time-consuming test cases to match the computing resources available to participants.

We randomly divided the participants into two groups. The group, which worked on the Guava-Java and OpenCV-C++ projects, ended up with 14 participants. The other group, which worked on the Spring Boot-Java and Bitcoin-C++ projects, concluded with 19 participants. All participants began with the Java projects and then proceeded to complete the C++ projects. The project guidelines for the Java and C++ projects are identical, with the exception that we included an estimated build time and machine types for the C++ projects. Despite having trained the participants on the configuration of VMs on GCP, we still observed that some participants working on the Spring Boot project complained about the long build time due to incorrectly assigning system resources to VMs. Considering that this situation could have a significant negative impact on the completion of the future tasks, we added a recommended configuration for system resources to address the insufficient system resource issues for the C++ projects.

### 5.3.1 As learners of OSS, are students able to accurately interpret the build issues that block build tasks? (RQ1)

We manually verified the 198 build outcomes reported by 33 participants using the approach mentioned in Section 5.2.5. We observed the following inconsistencies in reporting:

1. Misinterpretation of build results (31 out of 198): Participants misinterpreted the failed build results as successful. This was extensively observed in OpenCV (24, 57.1%).

2. Incorrect build command (1 out of 198): A wrong build command was used, resulting in a partial build outcome.

3. Silent failure (2 out of 198): Some test cases were not executed due to a silent crash.

In total, 34 out of 198 build outcomes were misinterpreted, resulting in a misinterpretation rate of

Table 5.2: Success rate of verified build outcomes

| | Ubuntu | CentOS | openSUSE | 3 OS |
|---|---|---|---|---|
| Guava - 14 | 92.9% | 100.0% | 92.9% | 95.2% |
| Spring Boot - 19 | 26.3% | 15.8% | 10.5% | **17.5%** |
| Java - 33 | 54.5% | 51.5% | 45.5% | 50.5% |
| OpenCV - 14 | 35.7% | 35.7% | 35.7% | **35.7%** |
| Bitcoin - 19 | 68.4% | 63.2% | 0.0% | 43.9% |
| C++ -33 | 54.5% | 51.5% | 15.2% | 40.4% |
| Java & C++ - 66 | 54.5% | 51.5% | **30.3%** | **45.5%** |

17.2%. We believe this observation occurred due to two main reasons. Firstly, the OpenCV project required non-standard environment variables to pass all the test cases. As participants noted that the failed test cases were only a small portion of the passed test cases, they tended to believe it was a successful build. This could pose a serious problem if the missing tests were to catch potential defects in the software. Secondly, students lacked the background knowledge to fully interpret the build results since they had never seen a successful build of the new project before. They tended to scan the keywords in the build log, which led to the ignoring of silent failures.

### 5.3.2 How often do build tasks end with unresolved issues? (RQ2)

Table 5.2 presents the success rate of build outcomes verified with the correction of inconsistent reporting. Spring Boot has the lowest success rate among the four PLs because it suffered significantly from insufficient system resources. OpenCV has the second-lowest success rate, mainly because most participants neglect a non-standard environment variable. openSUSE is not a popular OS in the documentation of our OSS projects, which leads to the lowest success rate of 30.3%. The overall success rate is 45.5%, indicating that more than half of the build tasks are blocked by unresolved issues.

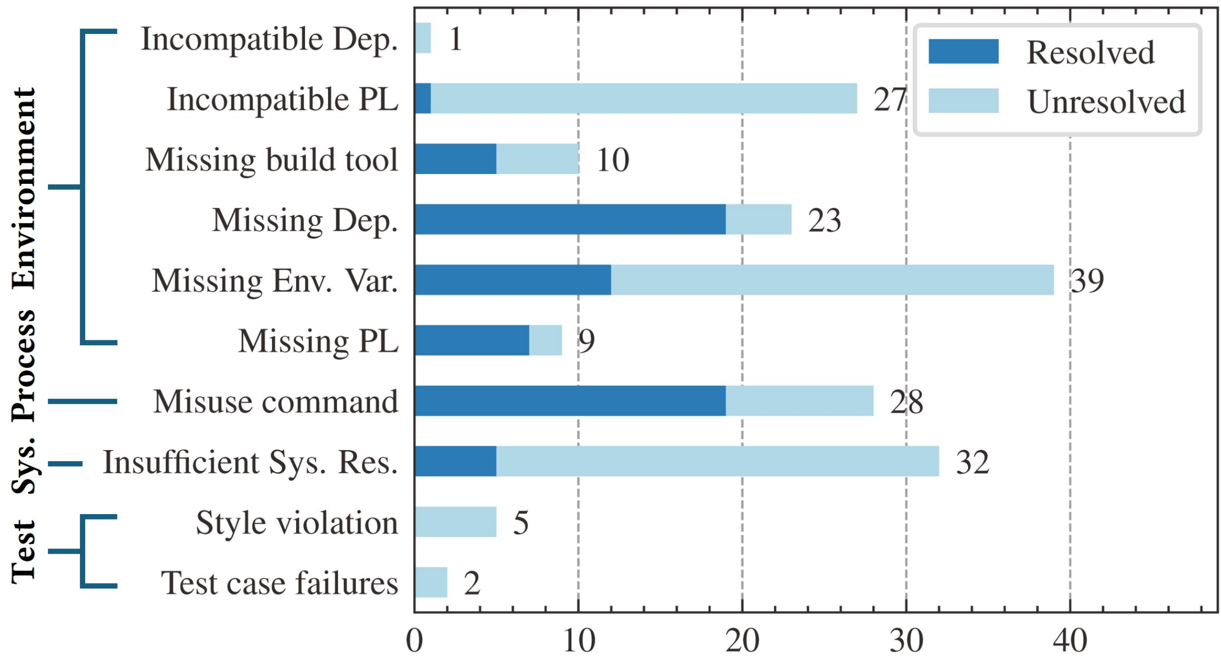



**Figure 5.2**: Root symptoms from resolution strategies study

### 5.3.3 What are the common symptoms of unresolved build issues, and to what degree can they be mitigated? (RQ3)

In unresolved build issues, we identified 18 unique errors, as detailed in Table 5.3. Furthermore, we analyzed the resolved build issues that contain these 18 unique errors in successful build outcomes. In Table 5.3, "#UR" represents the number of unresolved issues, while "#R" denotes the number of resolved issues related to the 18 unique errors. Figure 5.2 and Table 5.3 illustrate these root symptoms, which can be categorized by Environment-Related (*Incompatible ...* and *Missing ...*), Process-Related (*Misuse command*), System-Related (*Insufficient system resources*), and Test-Related (*Style violation* and *Test case failure*).

The number of unresolved incompatible and missing issues indicates that environment-related issues constitute a significant portion of the failed build tasks. The build issues from the OpenCV project, which lacked a non-standard environment variable, contributed to the *Missing environment variable*. Participants who built the Spring Boot project encountered a severe *Insufficient system resources*. Many participants made mistakes by forgetting to configure the Makefile, leading to

*Misuse command*.

We observed that the symptoms were not uniformly distributed. Consequently, students often encountered difficulties in resolving build issues caused by these recurring symptoms. Compared to the study by Lou et al. [55], our study shows that students still frequently experience the same symptoms as developers. However, these are mostly limited to issues caused by a lack of installation, version incompatibility errors, and environment variable issues. We also observed additional symptoms not covered by previous studies that are related to students' actions, such as *Misuse command* and *Insufficient system resources*.

**Table 5.3**: Errors, symptoms and resolutions

| Project | Symptom | Error & Description | #UR | #R | Resolution |
|---|---|---|---|---|---|
| Guava | Insufficient Sys. Res. | E1: Failed to execute goal org.apache.maven.plugins: maven-surefire-plugin:2.7.2:test (default-test) on project guava-testlib : There are test failures | 2 | 0 | |
| Spring-Boot | Insufficient Sys. Res. | E2: Gradle build daemon disappeared unexpectedly (it may have been killed or may have crashed) | 4 | 1 | Retry with incremental compilation |
| | | E3: Build is slow | 7 | 0 | |
| | | E4: Build crash | 10 | 2 | Retry with incremental compilation |
| | Incompatible PL (Java) | E5: Execution failed for task ':spring-boot-project: spring-boot:compileKotlin' | 4 | 1 | Use JDK instead of JRE |
| | | E6: Execution failed for task ':spring-boot-project: spring-boot:javadoc' | 11 | 0 | |
| | | E7: Execution failed for task ':buildSrc:test' | 1 | 0 | |
| | | E8: Execution failed for task ':spring-boot-project: spring-boot:test'. 4 test failures | 1 | 0 | |
| | Missing PL (Java) | E9: ERROR: JAVA_HOME is not set and no 'java' command could be found in your PATH | 2 | 7 | Install Java |
| | Style violation | E10: Execution failed for task ':checkstyleNohttp' | 5 | 0 | |
| | Test case failures (Flaky) | E11: Execution failed for task ':spring-boot-project: spring-boot:test'. flaky test cases | 2 | 0 | |
| OpenCV | Missing Env. Var. | E12: 7 FAILED TESTS | 27 | 12 | Set up OPENCV_TEST_DATA_PATH |
| Bitcoin | Insufficient Sys. Res. | E3: Build is slow | 3 | 1 | Wait patiently |
| | | E4: Build crash | 1 | 1 | Build in background |
| | Missing build tool (Make) | E13: make: command not found | 5 | 5 | Install Make |
| | Misuse command (no configure) | E14: make: *** No rule to make target 'check'. Stop | 9 | 19 | Configure Makefile |
| | Missing Dep. | E15: Libtool library used but 'LIBTOOL' is undefined | 1 | 11 | Install Libtool library |
| | | E16: configure: error: C++ compiler cannot create executables | 3 | 8 | Install build-essential |
| | Incompatible Dep. | E17: Segmentation fault | 1 | 0 | |
| | Incompatible PL (G++) | E18: configure: error: cannot figure out how to use std::filesystem | 9 | 0 | |

### 5.3.4 What strategies have students employed to successfully address these unresolved build issues, and are these strategies effective? (RQ4)

Based on the resolved build issues related to the unresolved build issues (observed in RQ3), we studied the resolutions employed by students and summarized them into two strategies: *Brute Force* and *Ad Hoc*, which seem not very effective as the overall resolution rate ($\frac{\sum \#R}{\sum(\#UR+\#R)}$) is 38.6%.

**Brute Force:** This strategy has been applied to most instances of the *Insufficient system resources* build issue. Benefiting from the incremental compilation feature of Gradle in the Spring Boot project, a few participants who encountered system-related build issues were able to overcome low resource issues. However, for most participants who suffered low resource issues, this strategy tended to fail. Our observations indicate that most participants were overly focused on this strategy, and the failure of each retry led to a tremendous sense of frustration.

**Ad Hoc:** Most participants applied the *Ad Hoc* strategy, which tended to be effective for *missing*-related symptoms. Participants simply needed to install the component that was reported as missing in the error message. However, for non-obvious symptoms, such as the *missing environment variable* symptom, it was easy for participants to overlook potential failures, not to mention the incompatibility issues. This strategy may further increase the incidence rate of encountering *trap issues*.

We further observed that some build issues, referred to as **trap issues**, can be the root cause of the low effectiveness of students' strategies. Based on the two common strategies, we identified several issues that tend to lead to a failed build for students. For instance, the issues with E5 can be resolved by changing the Java Runtime Environment (JRE) to the Java Development Kit (JDK), as indicated in the detailed error message "Make sure Kotlin compilation is running on a JDK, not JRE.". However, changing from JRE to JDK is more complex than installing JDK initially. Even with this explicit error message, other participants were still unable to resolve the issue. Another significant example is the issue associated with E10. In this scenario, participants downloaded the JDK package directly into the root directory of the Spring Boot project. This action led the

Checkstyle tool to inadvertently verify the JDK package, which ultimately resulted in the test crashing.

In essence, *trap issues* are build issues that contradict students' intuition. Consequently, build issues with a low resolution rate are considered *trap issues*. Predicting *trap issues* without sufficient information can be challenging. In this study, the incompatible issues and style violation issues can be identified as *trap issues* since most of them have a low resolution rate. The official documents of OSS usually don't contain recommended system resource information for builds. Therefore, the *insufficient system resources* build issue can be identified as a trap issue for students. Neither *brute force* nor *ad hoc* strategies can effectively resolve these *trap issues*, turning students' build activities into a game of chance.

### 5.3.5 Are there any prior experiences that may correlate with students' ability to resolve build issues? (RQ5)

Figure 5.3 presents the distribution of familiarity, ranging from 1 to 5 (with 1 representing "Not familiar at all" and 5 representing "Extremely familiar"). Most participants are more familiar with Java and Ubuntu than with other PLs and OSs. We calculated the Spearman's rank correlation coefficient between each type of familiarity and the build outcomes. Figure 5.4 presents the coefficients, where "F_" denotes familiarity and "B_" denotes build outcome.

Most prior experiences have a very weak correlation with build outcomes. However, the prior experience with C++ and the build outcome of OpenCV have a moderate negative correlation (-0.55, $p$-value$<$0.001). We further investigated the collected data and confirm the trend: participants who are more familiar with C++ tend to fail more often in the OpenCV project (Success rates: Not familiar at all=66.7%, Slightly familiar=50%, Moderately familiar=0%, Very familiar=0%). Since the majority of participants misreport their OpenCV build outcomes, a potential reason could be that participants more familiar with C++ tend to overlook details in the official documents, which leads to failures with E12.

The rest of the prior experiences, such as PLs, build systems, and OSs, do not show significant

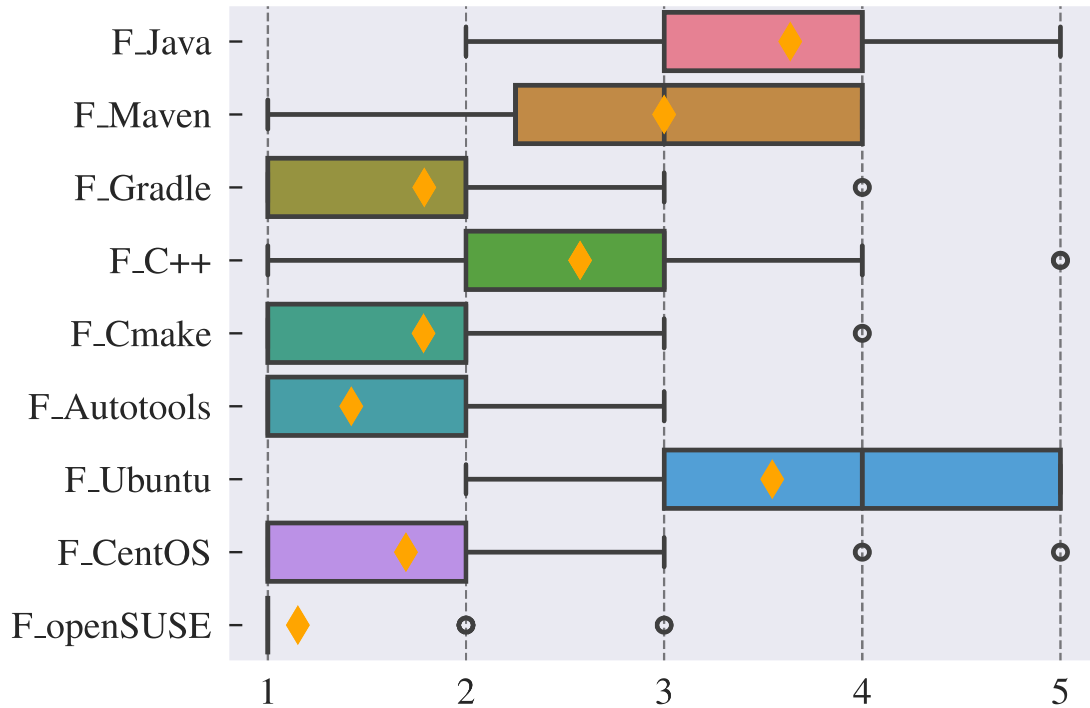



**Figure 5.3**: Distribution of prior experiences

correlation. As correlation does not imply causation, the impact of prior experiences on students' ability to resolve build issues may still need further exploration.

## 5.4 Intervention Study

In our resolution strategies study, we observed that participants frequently fell into *trap issues*. We believe this is the main reason why students were unable to resolve the build issues. We hypothesize that preventing these *trap issues* could be a more effective strategy than the common strategies currently employed by students. To test this hypothesis, we conducted an intervention study in 2023, where we added additional references to the project guidelines for participants. To be practical and follow most scenarios of using OSS in class, we allow students to use any external references they can find in both phases. The official documents were already provided in the first phase. As previously noted, the previous group that worked on the Spring Boot (Java) and Bitcoin (C++) projects encountered a wider variety of symptoms. Therefore, for 2023 participants, we selected Spring Boot and Bitcoin projects with the same release versions. We kept the rest of the

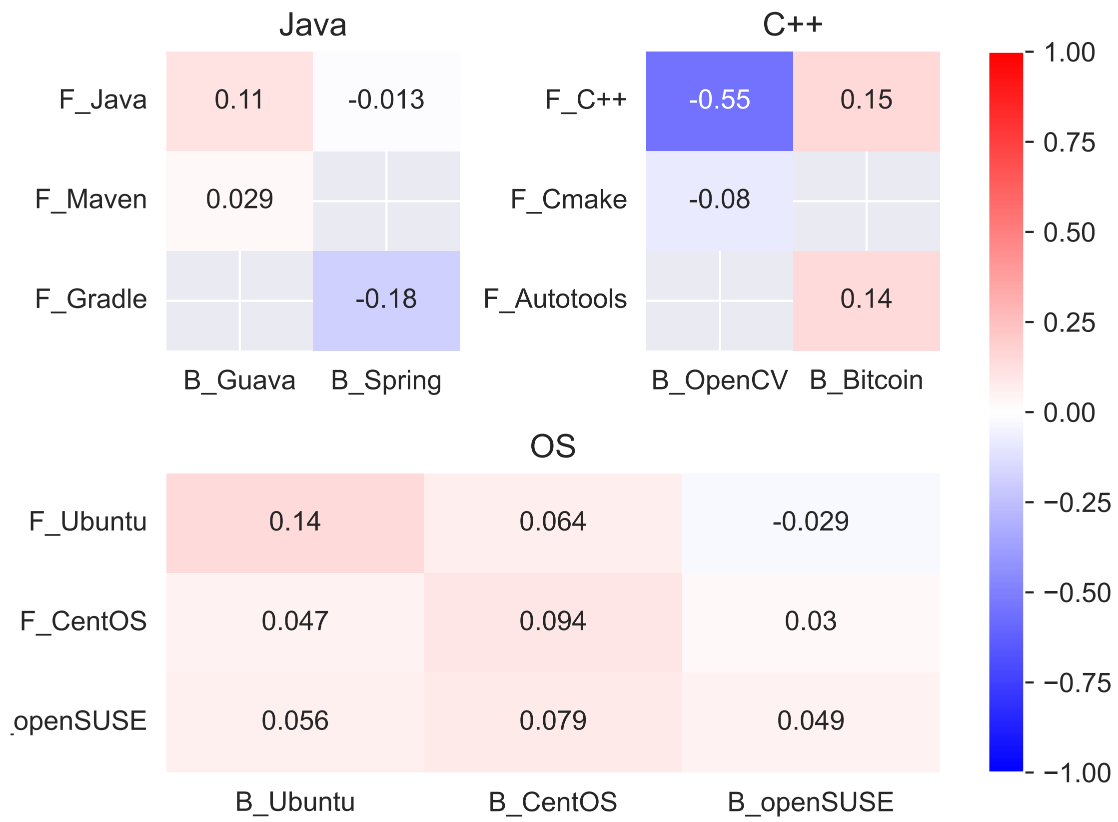



**Figure 5.4**: Spearman's rank correlation of prior experiences on build success

study setup as identical as possible.

### 5.4.1 Given the resolution strategies employed by students, what proactive measures can we introduce to effectively enhance their performance in resolving build issues? (RQ6)

The additional references provided for the intervention are as follows:

- **Official guide links with "how to build"**: in the 2022 study, we only provided links to general official guides. However, in the 2023 study, we improved this by providing links specifically to the "how to build" sections.
- **Example output for successful build results**: the screenshots demonstrate successful build outcomes. We also clarified the unrelated error message that appears in the CI reporting for the Spring Boot project.
- **Recommended PL versions**: we extract recommended PL versions from official documents and list them explicitly.
- **Dependencies**: information about how dependencies are required to be installed in different build systems, with links to the dependencies sections in official documents.
- **JDK vs. JRE**: we provide a link to the difference between JDK and JRE in Java, and remind that the package name might be different in Linux distributions.
- **CheckStyle**: background knowledge of the Checkstyle plugin and what it will do to a Spring Boot project.
- **Recommended system resource configuration for VMs**: as we mentioned in our study on resolution strategies, the information was initially provided only for C++ projects. In this phase, we have also extended it to Java projects. configuration

As shown in Table 5.4, the success rates for both projects were improved by intervention. The success rates of Spring Boot and Bitcoin improved from 17.5% to 91% and 43.9% to 82%,

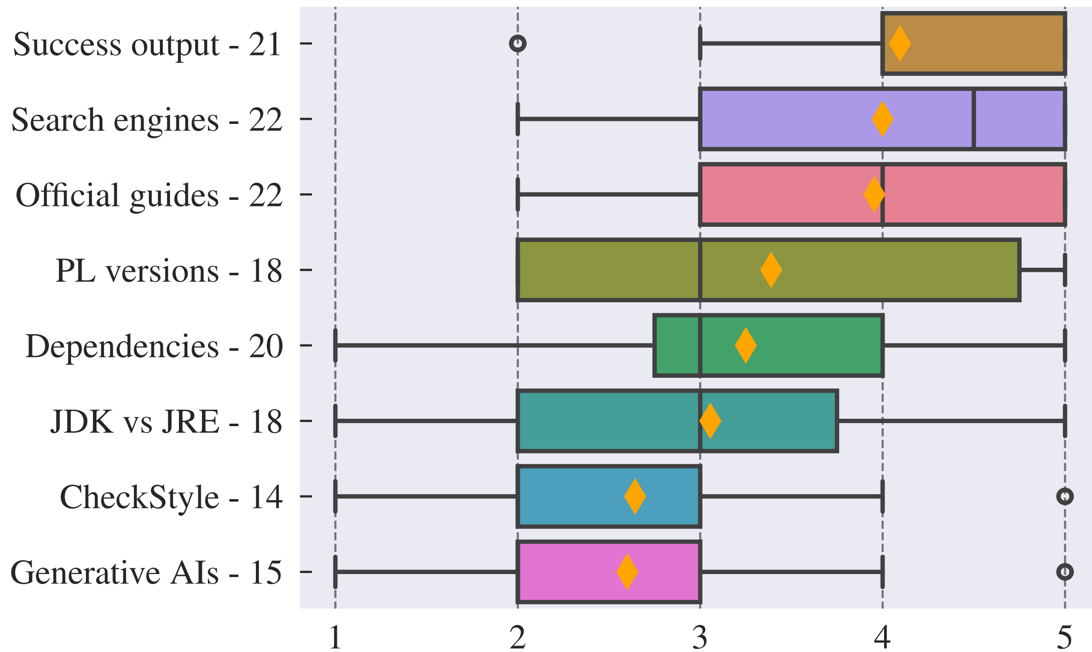



**Figure 5.5**: Helpfulness rating for intervention (diamond markers indicates means)

respectively. These improvements are statistically significant (both $p$-values of the Chi-squared test are less than 0.001).

Figure 5.5 presents the box plot of helpfulness rating from participants about intervention, ranging from 1 to 5, with an option for "not applicable". The example output (success output) outperforms other references. The search engines and official guides also have similar means. The references to PL versions and dependencies are helpful, and students also explicitly mentioned their usefulness in the barriers section of the feedback surveys. References to "JDK vs. JRE" and "CheckStyle", which are not found in official documents, rated to be less helpful than others.

**Table 5.4**: Build outcomes with intervention

| | Spring Boot - Java $\chi^2 = 64.2$ | $p < .001$ | Bitcoin - C++ $\chi^2 = 17.6$ | $p < .001$ |
|---|---|---|---|---|
| | Success | Failure | Success | Failure |
| Baseline - 57 | 10 (**17.5%**) | 47 (82.5%) | 25 (**43.9%**) | 32 (56.1%) |
| Intervention - 66 | 60 (**91%**) | 6 (9%) | 54 (**82%**) | 12 (18%) |

However, our observations from the feedback surveys indicate that participants did benefit from these external references. We did not observe any Checkstyle-related issues among the unresolved issues, and two participants explicitly stated that the barrier for them is the difference between JRE and JDK.

For the Spring Boot project, all six unresolved issues are incompatibility issues, which are the same as the *trap issues* identified in our 2022 study. One of the six unresolved issues was actually resolved after rebooting the VM. The official OSS documents usually don't include recommended system resource information for builds. Since we provided the recommended system resource configuration in this phase, none of the participants reported symptoms of insufficient system resources. The intervention on this OSS project was observed to reduce both *trap issues* and other build issues. However, self-reported helpfulness does not necessarily equate to actual effectiveness for all students.

For the Bitcoin project, we observed that 1 of the 12 issues was due to low system resource. Also, a *Misuse command (no configure)* issue was observed in the 12 unresolved issues. 10 of the 12 unresolved issues were related to incompatibility, 9 of which occurred in the openSUSE OS. In the openSUSE, for the Bitcoin project, the occurrence rate of *trap issues* (incompatible) reduced from 52.6% (10/19) to 40.9% (9/22), and other build issues were fully resolved after intervention. As the recommended system resource information was provided to students in all phases of the Bitcoin project, the intervention was observed to reduce both trap issues and other build issues, regardless of the system resource limits.

When looking at each student individually, all the references provided are beneficial to them. This is because every reference has been rated with a highest helpfulness score of either 4 or 5, as shown in Figure 5.5. Most of these helpful references can be easily obtained from an OSS project and used as proactive measures, such as well-organized official guides, recommended PL versions, and dependencies. The success output is difficult to extract from documents but could be accessed from the CI system or previous build logs. Identifying and presenting project-specific knowledge, such as "JDK vs. JRE" and "CheckStyle", to students as a proactive measure poses a challenge.

Despite often being overlooked by document contributors, these project-specific knowledge may still play a critical role in preventing trap issues for students.

## 5.5 LESSONS LEARNED

We categorize our lessons learned into five subsections based on the various stakeholders in the OSS and CS education communities.

### 5.5.1 Educators

As a general finding from our study, students often face build issues when they try to build real-world OSS projects. Therefore, educators should be aware of such challenges whenever they adopt real-world OSS projects in their SE course projects.

As students are often unfamiliar with the OSS build scenario and lack project-specific experience, misinterpretations of build issues do occur. Section 5.3.1 reveals a 17.2% misinterpretation rate in our exploratory study, which is not negligible. A list of commonly faced build error messages and their solutions could help students and prevent them from struggling without understanding the root causes. Furthermore, if educators rely on students' self-reported evaluations to design or adjust SE courses related to OSS builds, additional clarification or deeper investigation should be performed to address this mismatch.

The educational use of OSS in CS courses allows students to work with real-world projects, making it an important aspect of the SE curriculum. However, as Section 5.3.2 shows, build issues can pose significant barriers to the successful integration of OSS in course projects. Students' strategies for resolving build issues can be vulnerable, and they often fall into *trap issues*, which require significant time and effort to resolve. Therefore, educators should select reliable OSS projects, identify *trap issues* early, and implement proactive measures to mitigate the difficulties associated with the OSS projects build.

In the meantime, Section 5.4.1 shows that there might not be a silver bullet for proactive measures, so continuing to develop these measures could be a more viable approach. Additionally,

Section 5.3.5 reveals that prior experiences may not apply to builds. If OSS builds tend to be project-specific, gathering project-specific experiences from student feedback or key information from extensive OSS documents could provide better assistance to students than relying solely on their expertise and background.

### 5.5.2 Researchers

Build issues often arise while students work on OSS projects, presenting both a challenge and a research opportunity. Section 5.3.3 shows that these build issues are not only similar to a subset of developers' build issues but can also be unique from a student's perspective, warranting deeper exploration across a wider range of OSS subjects. Identifying *trap issues* is essential for addressing build issues. However, in cases where students have limited information, such as when they are working on an OSS project newly integrated into a CS course, it may be difficult for them to identify these issues. Future research could focus on developing more precise and effective methods for identifying *trap issues* so that students can avoid them and educators can prepare guidance on how to bypass them. Additionally, exploring the impact of prior experiences on build processes, as indicated by Section 5.3.5, could be another area for future research. Collecting a more refined dataset and involving a larger group of students may yield different results. Proactive measures have been proven effective in reducing build issues, but not all measures can be easily created without project-specific knowledge or key information. Therefore, research focused on extracting information for proactive measures is worth exploring.

According to Section 5.3.1, inconsistencies in students' self-reported build issues cannot be ignored. To avoid these threats, it is more effective to monitor the process in real-time rather than reproduce it later. While verifying the reported data can be time-consuming, it may be necessary.

### 5.5.3 Students

As students engage in OSS build activities, they take on the role of learners. To avoid common pitfalls, several pieces of advice can be beneficial. Previous experiences with PLs or build systems

may not be relevant to the current OSS project. Therefore, it is important to rely on the key information from OSS documentation. Students should be encouraged to try different strategies, such as resetting the build environments, even if they are not comfortable with them. Intuitive strategies might lead to deeper issues. Sometimes, students need to think outside the box, as some issues may be caused by external resources or even flaky tests. Recognizing their own inexperience is crucial, which includes accepting the possibility of misinterpretation and a lack of project-specific knowledge. This means that students may need additional support while building OSS, even if they appear confident in related areas. If possible, using a popular OS can improve the build success rate.

Students' experiences of build failure can be valuable if reported to educators or OSS contributors, as this feedback can help create proactive measures for future students.

### 5.5.4 OSS contributors

OSS contributors should recognize the increasingly significant role that OSS plays in CS education, which also facilitates students' participation in OSS projects. Given that students may not have the required knowledge, particularly about OSS projects builds, mutual understanding of different roles within the OSS community is critical. Adding some external references, even if they pertain to basic background knowledge, can be helpful for learners. Not only are project-specific experiences valuable for other roles, but well-organized documentation and public build information are also crucial. Our study finds that error logs from the compiler or build systems are often too general to be useful and can sometimes be misleading. OSS contributors should consider supplementing them with more project-specific information to avoid misinterpretations. Build error messages are also sometimes hidden due to the complicated build process (e.g., the compilation process was called by a script but the log was not preserved or passed to the caller process). So build scripts need to be designed and tested more carefully to explicitly show error messages.

The project-specific experiences of OSS contributors can lead to the development of proactive measures that ultimately benefit OSS itself. Understanding students' needs can help them become

potential contributors in the future.

### 5.5.5 Generative AI

As generative AIs became increasingly popular in 2023, we noted that using AI services may impact our intervention study compared with common external references found by search engines. We did not limit students' use of any external references in both phases to be practical. Figure 5.5 shows that students rated generative AIs as the least helpful (lowest mean). To understand the reason behind this observation, we further case-studied the AI services by querying some error messages on ChatGPT 3.5 and 4 [65] in February 2024. The results of the case study align with the students' feedback. For example, we tested the error "Execution failed for task :spring-boot-project:spring-boot-tools:spring-boot-antlib:javadoc." ChatGPT 3.5 gave 10 solutions, but only one of them mentioned the real root cause (JDK compatibility). ChatGPT 4 gave 3 unrelated solutions. The reason could be that AI services need key information to provide accurate suggestions in build issue resolution scenarios, but students may lack the experience to extract such information as input to AI services. Therefore, the answers from AIs were often overly general for students to find them helpful, because (1) so many different solutions may make students confused about which direction they should move to, and (2) the solutions are high level and do not include concrete actionable steps for the students to follow. In this OSS build scenario, future work may include adding more context to generative AIs, such as providing additional references to generative AIs.

## 5.6 Threats to Validity

### 5.6.1 Internal Validity

The major threat to the internal validity of our study is the reliance on self-reported data and manual verification processes. There may be errors in the process of data recording and analysis. To reduce this threat, we introduced a conflict resolution procedure and followed the open coding [75]. In addition, we leverage multiple information channels (e.g., reports, screenshots, command logs, and terminal recordings) for data recording and cross-validate the data among these channels.

Our study involving students who are one year apart can introduce a threat to internal validity. To mitigate this threat, we ensure that the curriculum and teaching methods remain consistent across the years to minimize differences in educational content. We use the same version of OSS projects and task setup and employ quantitative methods to analyze the overall build success rate. By collecting feedback from students, we aim to identify and address any potential issues that could affect the validity of our findings.

To ensure practicality, we permitted students to use external references, such as search engines, which may introduce threats to our study. Additionally, we did not track the usage of system resources, and providing a recommended system resource configuration to reduce the impact of build issues on low system resources could potentially weaken our findings. To mitigate these threats, we diligently analyze the collected data and investigate the details of each build issue in conjunction with our findings.

### 5.6.2 External Validity

The major threat to the external validity of our study is that our findings may not be generalizable to other students and OSS projects. To reduce this threat, we use popular OSS on GitHub [6] in different PLs, different build systems, and three OS variants to increase their representativeness. Given the exploratory nature of our study, our findings may not be generalizable to other institutions and courses that have different characteristics, such as varying enrollment capacities and diverse student demographics. Future research should consider these factors to enhance the applicability of the results across different educational settings.

## 5.7 Related Work

Our study is related to existing research efforts on the educational use of OSS projects, user studies of software build, empirical studies of build issues from the developer's perspective, and studies of novice programmers. We will discuss related works in each category in the following subsections.

**Educational Use of OSS Projects in CS Courses** The integration of open-source software

(OSS) projects into computer science (CS) education has been widely explored. Nascimento et al. [62] reviewed how OSS projects facilitate software engineering (SE) learning. Choi et al. [26] proposed models for integrating OSS into lower-level CS courses. Fang et al. [30] compared student contributions to general OSS projects and those aimed at social good, offering insights into educational interventions. Salerno et al. [74] examined the impact of OSS development courses on students' self-efficacy and the barriers they face. These studies collectively underscore the value of OSS projects in CS education, highlighting both the challenges and the potential for enhancing student learning and engagement. Unlike previous studies, our research specifically focuses on the build challenges encountered during the educational use of OSS projects in CS courses.

**User Studies of Software Build.** The most relevant research to ours involves user studies on software build tasks. Kwan et al. [50] examined IBM developers to see if team composition and coordination affect build success. Dawns et al. [29] evaluated a build management tool's impact on handling build failures. Philips et al. [67] studied Microsoft build teams, finding social challenges were significant. Kerzazi et al. [47] analyzed 3,214 builds, noting a 17.9% failure rate and significant time costs. Hilton et al. [40] interviewed developers about continuous integration (CI) processes. Vassallo et al. [83] assessed a tool for summarizing build failures. Due to the challenges of performing such user studies, we can also see that there are not many existing studies, and all the studies have relatively small scales. Different from our research, all these studies are from the perspective of project managers or senior developers instead of CS students (OSS learners).

**Empirical Studies on Build Issue** Besides user studies, there have been many empirical studies on software build history and failures. McIntosh et al. [58] studied version histories to estimate the effort required to revise and repair build scripts. Later, they correlated this effort with the type of build systems used [59]. Xia et al. [93] summarized the characteristics of bugs in build systems. Zhao et al. [100] found that build failures, though typically of lower severity, take more time to fix. Xia and Li [93] investigated the feasibility of predicting build failures using the TravisTorrent dataset [17]. Shridhar et al. [77] qualitatively analyzed changes in build scripts. Barrak et al. [16]

studied the correlation between build failures and code smells. Lou et al. [55] identified patterns in build fixes. Zolfagharinia et al. [102] found different distributions of build failures across environments. Licker and Rice [53] used mutation testing to detect incorrect rules in build scripts. Wu et al. [92] studied build failures in Docker environments. These existing studies focus on build failure logs and build failure fixes in the commit history. In contrast, our study monitors and analyzes the whole process of CS students finishing multiple building tasks and records all the system environment factors that affect the build process failures.

**Studies of Novice Programmers** Another research area related to our research is user studies of novice programmers. Lahtinen et al. [52] surveyed over 500 students and teachers in computer-related majors to identify challenges faced by novice developers. Warner et al. [88] evaluated CodePilot with eight novice developers to assess its usability and educational benefits. Marques et al. [56] monitored students over nine semesters to see if reflexive weekly monitoring aids in completing course projects and improving coding skills. Ardimento et al. [15] analyzed 40 novice developers' IDE usage patterns across five tasks. Romano et al. [73] examined 29 novice developers to determine the impact of test-driven development on positive affective states. Rehman et al. [70] reviewed 208 novice contributions on GitHub to identify common contribution types. Oliveira et al. [64] compared novice and experienced developers to study code smells. Compared with these studies, our work focuses on the software build process instead of general programming.

## 5.8 Conclusion

We present a dual-phase exploratory study involving 55 CS students, which includes a resolution strategies phase and an intervention phase. The aim is to gain a comprehensive understanding of build issue resolution among CS students. We investigate the reported build issues and the strategies students use to resolve them. Due to students' lack of background knowledge and project-specific experience, their intuitive strategies tend to be straightforward. This limitation can make them more susceptible to certain build issues that are hard to resolve. Proactively preventing these potential *trap issues* can significantly improve the success rate of their build outcomes, thereby

enhancing the OSS learning experience in the SE course. This work highlights directions for further research in this area that could benefit various stakeholders in the OSS and CS education communities.

## 5.9 Data Availability

The dataset for this study is available on Zenodo (https://doi.org/10.5281/zenodo.14885822). Due to local IRB restrictions, some sensitive data may not be publicly accessible.

# CHAPTER 6: CONCLUSION AND FUTURE DIRECTIONS

## 6.1 Dissertation Summary

Software environments are essential to modern software development, providing the necessary tools and frameworks to boost productivity, reduce costs, and improve software quality. Ensuring consistency across various environments is crucial for maintaining reliable and efficient builds, tests, and executions. However, managing consistency across software environments is a well-recognized challenge. Despite existing approaches aiming to make environments more reproducible and consistent, environment-related issues are still frequently encountered, especially by those facing additional restrictions or lacking domain knowledge.

Firstly, we focused on reporting cross-project failures (CPFs) while developers face constraints in sharing their environment. We addressed the research problem of extracting failures along with their execution environment by creating PExReport, a novel framework designed to extract both code and build dependencies and generate pruned executable CPF reports. PExReport automates the generation of stand-alone executable CPF reports by leveraging build systems to prune source code and dependencies. It analyzes the build process to create a pruned environment that accurately reproduces CPFs. We demonstrated PExReport within the Maven build system as PExReport-Maven, which can create concise CPF reproduction packages by trimming down source code, dependencies, and the Maven build environment. Our evaluation shows that PExReport-Maven successfully reproduced 184 out of 198 CPFs with exact failure types and messages, achieving a high reproduction rate of 92.93%. The average reduction rate is 55.37% for required source classes. It also outperformed in handling internal classes, source + internal classes, build configurations, and resources, with reduction rates of 75.94%, 72.97%, 82.96%, and 74.18%, respectively.

Secondly, we focused on addressing inconsistencies in the build environment from the perspective of non-contributors. Our pilot study explored how non-contributors (Computer Science students) handle build issues in open-source software (OSS) by collecting data from command history logs, environment variables, network information, and snapshots of virtual machine states. We

found that non-contributors often struggle with certain types of build issues. Furthermore, we conducted a dual-phase study involving 330 build tasks among 55 CS students. Phase I characterized the build issues students faced, their resolution attempts, and the effectiveness of those attempts, revealing that intuitive resolution strategies often led to “trap” issues, which are especially difficult to resolve. In Phase II, an intervention successfully improved the build success rate by proactively addressing these “trap” issues.

Overall, this dissertation addresses two significant research problems related to inconsistencies across software environments. The PExReport addressed the challenge of extracting failures along with their execution environment. Our research studies focused on non-contributors, laying the foundation for best practices in building OSS within CS education, and demanding enhanced tool support to simplify the OSS build process for non-contributors.

## 6.2 Future Research Directions

The research problem of extracting failures along with their execution environment has long been neglected and remains highly challenging in practice. Our research holds considerable practical value and offers substantial potential for real-world CPF reporting. More user studies are needed to understand how much PExReport can assist developers in reporting real-world CPFs. Additionally, we aim to further refine CPF reports through finer-grained analysis and detailed source code reduction. Currently, PExReport has been implemented only within Java ecosystems and the Maven build system. Expanding it to other software ecosystems and build systems could introduce new challenges and warrants further exploration.

On the research problem of addressing inconsistencies in the build environment from the perspective of non-contributors, it is essential to identify and resolve certain “trap” issues. The advent of generative AI enables the rapid bridging of domain-specific knowledge gaps through AI tools, which can also be leveraged to predict and proactively prevent“trap” issues. While AI acquires sufficient domain-specific knowledge, the build environment setup could be fully automated without human involvement. The future research domain, AI for Build, aims to leverage AI to address

challenges in maintaining consistency across build environments. This includes automating the setup process, predicting potential issues, and proactively preventing inconsistencies.

## BIBLIOGRAPHY


[1] Apache maven. https://maven.apache.org/, 2002.

[2] Apache maven archetype. https://maven.apache.org/guides/introduction/introduction-to-archetypes.html, 2002.

[3] Apache groovy. https://groovy-lang.org/, 2003.

[4] inotify. https://man7.org/linux/man-pages/man7/inotify.7.html, 2005.

[5] Gradle. https://gradle.org/, 2008.

[6] Github, http://github.com/, 2013.

[7] seb-m/pyinotify. https://github.com/seb-m/pyinotify, 2015.

[8] Difference between downloading bitcoire core from bitcoin.org and compiling from github. https://bitcoin.stackexchange.com/questions/59875, 2017. Accessed: 2024-03-21.

[9] How to fix docker: Got permission denied issue. stackoverflow.com/questions/48957195, 2018. Accessed: 2024.

[10] Apache maven help:effective-pom. https://maven.apache.org/plugins/maven-help-plugin/effective-pom-mojo.html, 2022.

[11] Proguard: Java and android apps optimizer. https://www.guardsquare.com/en/products/proguard, 2022-08-01.

[12] Introduction to the build lifecycle. https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html, 2023.

[13] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[14] Tim Allison. Tika/tika-2497. `https://issues.apache.org/jira/browse/TIKA-2497`, 2017.

[15] Pasquale Ardimento, Mario Luca Bernardi, Marta Cimitile, and Fabrizio Maria Maggi. Evaluating coding behavior in software development processes: A process mining approach. In *2019 IEEE/ACM International Conference on Software and System Processes (ICSSP)*, pages 84–93. IEEE, 2019.

[16] Amine Barrak, Ellis E Eghan, Bram Adams, and Foutse Khomh. Why do builds fail?-a conceptual replication study. *Journal of Systems and Software*, 177:110939, 2021.

[17] Moritz Beller, Georgios Gousios, and Andy Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 447–450. IEEE, 2017.

[18] Ouafa Bentaleb, Adam SZ Belloum, Abderrazak Sebaa, and Aouaouche El-Maouhab. Containerization technologies: Taxonomies, applications and challenges. *The Journal of Supercomputing*, 78(1):1144–1181, 2022.

[19] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318, 2008.

[20] John D Blischak, Emily R Davenport, and Greg Wilson. A quick introduction to version control with git and github. *PLoS computational biology*, 12(1):e1004668, 2016.

[21] Yevgeniy Brikman. *Terraform: up and running: writing infrastructure as code*. " O'Reilly Media, Inc.", 2022.

[22] Bobby R Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. Jshrink: In-depth investigation into debloating modern java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 135–146, 2020.

[23] Aleksander Bulajic, Samuel Sambasivam, and Radoslav Stojic. An effective development environment setup for system and application software. In *Proceedings of the Informing Science and Information Technology Education Conference*, pages 37–66. Informing Science Institute, 2013.

[24] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. Taming behavioral backward incompatibilities via cross-project testing and analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 112–124, 2020.

[25] Hong Cheng, David Lo, Yang Zhou, Xiaoyin Wang, and Xifeng Yan. Identifying bug signatures using discriminative graph mining. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 141–152, 2009.

[26] Emma Choi, Lisa Meng, and John Hott. Open source software practices in cs2. In *Proceedings of the 21st Koli Calling International Conference on Computing Education Research*, pages 1–5, 2021.

[27] S. Danicic, A. De Lucia, and M. Harman. Building executable union slices using conditioned slicing. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, pages 89–97, 2004.

[28] Jens Dietrich, Vyacheslav Yakovlev, Catherine McCartin, Graham Jenson, and Manfred Duchrow. Cluster analysis of java dependency graphs. In *Proceedings of the 4th ACM symposium on Software visualization*, pages 91–94, 2008.

[29] John Downs, Beryl Plimmer, and John G Hosking. Ambient awareness of build status in collocated software teams. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 507–517. IEEE, 2012.

[30] Zihan Fang, Madeline Endres, Thomas Zimmermann, Denae Ford, Westley Weimer, Kevin Leach, and Yu Huang. A four-year study of student contributions to oss vs. oss4sg with a lightweight intervention. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, pages 3–15, New York, NY, USA, November 2023. Association for Computing Machinery.

[31] Apache Software Foundation. Apache's jira issue tracker. `https://issues.apache.org/jira/secure/Dashboard.jspa`, 1999.

[32] Swapna S. Gokhale, ThÃ©rÃšse Smith, and Robert McCartney. Integrating open source software into software engineering curriculum: Challenges in selecting projects. In *2012 First International Workshop on Software Engineering Education Based on Real-World Experiences (EduRex)*, pages 9–12, 2012.

[33] Mark Harman and Robert Hierons. An overview of program slicing. *software focus*, 2(3):85–92, 2001.

[34] Nicolas Harrand, César Soto-Valero, Martin Monperrus, and Benoit Baudry. Java decompiler diversity and its application to meta-decompilation. *Journal of Systems and Software*, 168:110645, 2020.

[35] Foyzul Hassan, Shaikh Mostafa, Edmund SL Lam, and Xiaoyin Wang. 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)*, pages 38–47. IEEE, 2017.

[36] Foyzul Hassan, Rodney Rodriguez, and Xiaoyin Wang. Rudsea: recommending updates of dockerfiles via software environment analysis. In *Proceedings of the 33rd acm/ieee international conference on automated software engineering*, pages 796–801, 2018.

[37] Foyzul Hassan and Xiaoyin Wang. Hirebuild: An automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th international conference on software engineering*, pages 1078–1089, 2018.

[38] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 380–394, 2018.

[39] Steffen Herbold, Jens Grabowski, Stephan Waack, and Uwe Bünting. Improved bug reporting and reproduction through non-intrusive gui usage monitoring and automated replaying. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 232–241. IEEE, 2011.

[40] Michael Hilton, Nicholas Nelson, Danny Dig, Timothy Tunnell, Darko Marinov, et al. Continuous integration (ci) needs and wishes for developers of proprietary code. 2016.

[41] Lorin Hochstein and Rene Moser. *Ansible: Up and Running: Automating configuration management and deployment the easy way.* " O'Reilly Media, Inc.", 2017.

[42] Jeff Huang and Charles Zhang. Lean: Simplifying concurrency bug reproduction via replay-supported execution reduction. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 451–466, 2012.

[43] Ranjit Jhala and Rupak Majumdar. Path slicing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming language Design and Implementation*, pages 38–47, 2005.

[44] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, 2005.

[45] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, 2002.

[46] Michael Kerrisk. script - make typescript of terminal session. `https://man7.org/linux/man-pages/man1/script.1.html`, 2023.

[47] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why do automated builds break? an empirical study. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 41–50. IEEE, 2014.

[48] Heiko Klare, Max E Kramer, Michael Langhammer, Dominik Werle, Erik Burger, and Ralf Reussner. Enabling consistency in view-based system development-the vitruvius approach. *Journal of Systems and Software*, 171:110815, 2021.

[49] Gunjan Kotwani and Pawan Kalyani. Open source software (oss): Realistic implementation of oss in school education. 12 2011.

[50] Irwin Kwan, Adrian Schroter, and Daniela Damian. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *IEEE Transactions on Software Engineering*, 37(3):307–324, 2011.

[51] Irwin Kwan, Adrian Schroter, and Daniela Damian. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *IEEE Transactions on Software Engineering*, 37(3):307–324, May 2011.

[52] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. *Acm sigcse bulletin*, 37(3):14–18, 2005.

[53] Nándor Licker and Andrew Rice. Detecting incorrect build rules. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1234–1244. IEEE, 2019.

[54] Tongping Liu, Charlie Curtsinger, and Emery D Berger. Doubletake: Fast and precise error detection via evidence-based dynamic analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 911–922. IEEE, 2016.

[55] Yiling Lou, Zhenpeng Chen, Yanbin Cao, Dan Hao, and Lu Zhang. Understanding build issue resolution in practice: symptoms and fix patterns. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 617–628, 2020.

[56] Maira Marques, Sergio F Ochoa, Maria Cecilia Bastarrica, and Francisco J Gutierrez. Enhancing the student learning experience in software engineering project courses. *IEEE Transactions on Education*, 61(1):63–73, 2017.

[57] Mark Marron. A new generation of intelligent development environments. In *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments*, pages 43–46, 2024.

[58] Shane McIntosh, Bram Adams, Thanh HD Nguyen, Yasutaka Kamei, and Ahmed E Hassan. An empirical study of build maintenance effort. In *Proceedings of the 33rd international conference on software engineering*, pages 141–150, 2011.

[59] Shane McIntosh, Meiyappan Nagappan, Bram Adams, Audris Mockus, and Ahmed E Hassan. A large-scale empirical study of the relationship between build technology and build maintenance. *Empirical Software Engineering*, 20:1587–1633, 2015.

[60] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Auto-completing bug reports for android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 673–686, 2015.

[61] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. Experience paper: a study on behavioral backward incompatibilities of java software libraries. In *Proceedings of the 26th*

*ACM SIGSOFT international symposium on software testing and analysis*, pages 215–225, 2017.

[62] Debora MC Nascimento, Roberto Almeida Bittencourt, and Christina Chavez. Open source projects in software engineering education: a mapping study. *Computer Science Education*, 25(1):67–114, 2015.

[63] Mathieu Nayrolles, Abdelwahab Hamou-Lhadj, Sofiene Tahar, and Alf Larsson. Jcharming: A bug reproduction approach using crash traces and directed model checking. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 101–110. IEEE, 2015.

[64] Roberto Oliveira, Rafael de Mello, Eduardo Fernandes, Alessandro Garcia, and Carlos Lucena. Collaborative or individual identification of code smells? on the effectiveness of novice and professional developers. *Information and Software Technology*, 120:106242, 2020.

[65] OpenAI. Chatgpt (Feb 24 version) [large language model], 2024.

[66] Stack Overflow. Stack overflow: a question-and-answer website for computer programmers, 2024.

[67] Shaun Phillips, Thomas Zimmermann, and Christian Bird. Understanding and improving software build teams. In *Proceedings of the 36th international conference on software engineering*, pages 735–744, 2014.

[68] Gustavo Henrique Lima Pinto, Fernando Figueira Filho, Igor Steinmacher, and Marco Aurelio Gerosa. Training software engineers using open-source software: The professors' perspective. In *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T)*, pages 117–121, 2017.

[69] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source

software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 345–355. IEEE, 2017.

[70] Ifraz Rehman, Dong Wang, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. Newcomer candidate: Characterizing contributions of a novice developer to github. In *2020 IEEE international conference on software maintenance and evolution (ICSME)*, pages 855–855. IEEE, 2020.

[71] Xiaoxia Ren and Barbara G. Ryder. Heuristic ranking of java program edits for fault localization. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 239–249, 2007.

[72] Fernando Rodríguez-Haro, Felix Freitag, Leandro Navarro, Efraín Hernánchez-sánchez, Nicandro Farías-Mendoza, Juan Antonio Guerrero-Ibáñez, and Apolinar González-Potes. A summary of virtualization techniques. *Procedia Technology*, 3:267–272, 2012.

[73] Simone Romano, Davide Fucci, Maria Teresa Baldassarre, Danilo Caivano, and Giuseppe Scanniello. An empirical assessment on affective reactions of novice developers when applying test-driven development. In *Product-Focused Software Process Improvement: 20th International Conference, PROFES 2019, Barcelona, Spain, November 27–29, 2019, Proceedings 20*, pages 3–19. Springer, 2019.

[74] Larissa Salerno, Simone de FranÃ§a TonhÃ£o, Igor Steinmacher, and Christoph Treude. Barriers and self-efficacy: A large-scale study on the impact of oss courses on student perceptions. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, pages 320–326, New York, NY, USA, June 2023. Association for Computing Machinery.

[75] C.B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, July 1999.

[76] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE access*, 5:3909–3943, 2017.

[77] Mini Shridhar, Bram Adams, and Foutse Khomh. A qualitative analysis of software build system changes and build ownership styles. In *Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–10, 2014.

[78] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–122, 2007.

[79] Perdita Stevens. Towards sound, optimal, and flexible building from megamodels. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 301–311, 2018.

[80] MatÃºÅ¡ SulÃr and Jaroslav PorubÃ€n. A quantitative study of java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 17–25, Amsterdam Netherlands, November 2016. ACM.

[81] Frank Tip, Chris Laffra, Peter F Sweeney, and David Streeter. Practical experience with an application extractor for java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 292–305, 1999.

[82] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano DiÂ Penta, Rocco Oliveto, Andrea DeÂ Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4):e1838, April 2017.

[83] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C Gall. Every build you break: developer-oriented assistance for build failure resolution. *Empirical Software Engineering*, 25:2218–2257, 2020.

[84] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. 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)*, pages 183–193. IEEE, 2017.

[85] Qianqian Wang, Chris Parnin, and Alessandro Orso. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 1–11, 2015.

[86] Xiaoyin Wang, David Lo, Jiefeng Cheng, Lu Zhang, Hong Mei, and Jeffrey Xu Yu. Matching dependence-related queries in the system dependence graph. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 457–466, 2010.

[87] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhi-liang Zhu. Will dependency conflicts affect my program's semantics. *IEEE Transactions on Software Engineering*, 2021.

[88] Jeremy Warner and Philip J Guo. Codepilot: Scaffolding end-to-end collaborative software development for novice programmers. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 1136–1141, 2017.

[89] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, pages 155–166, 2010.

[90] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[91] David Willmor, Suzanne M Embury, and Jianhua Shao. Program slicing in the presence of database state. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 448–452. IEEE, 2004.

[92] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. An empirical study of build failures in the docker context. In *Proceedings of the 17th international conference on mining software repositories*, pages 76–80, 2020.

[93] Xin Xia, Xiaozhen Zhou, David Lo, Xiaoqiong Zhao, and Ye Wang. An empirical study of bugs in software build system. *IEICE TRANSACTIONS on Information and Systems*, 97(7):1769–1780, 2014.

[94] Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. Practical isolation of failure-inducing changes for debugging regression faults. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 20–29, Sept 2012.

[95] Iyad Zayour and Hassan Hajjdiab. How much integrated development environments (ides) improve productivity? *J. Softw.*, 8(10):2425–2431, 2013.

[96] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, pages 183–200, 2002.

[97] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 253–267. 1999.

[98] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Localizing failure-inducing program edits based on spectrum information. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, pages 23–32, 2011.

[99] Xiangyu Zhang, R. Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 319–329, 2003.

[100] Xiaoqiong Zhao, Xin Xia, Pavneet Singh Kochhar, David Lo, and Shanping Li. An empirical study of bugs in build process. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1187–1189, 2014.

[101] Hao-Nan Zhu and Cindy Rubio-González. On the reproducibility of software defect datasets. In *Proceedings of the 45th International Conference on Software Engineering*, 2023.

[102] Mahdis Zolfagharinia, Bram Adams, and Yann-Gaël Guéhénuc. Do not trust build results at face value-an empirical study of 30 million cpan builds. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 312–322. IEEE, 2017.

[103] Mahdis Zolfagharinia, Bram Adams, and Yann-GaÃ«l GuÃ©hÃ©neuc. A study of build inflation in 30 million cpan builds on 13 perl versions and 10 operating systems. *Empirical Software Engineering*, 24(6):3933–3971, 2019.

# VITA

Sunzhou Huang enrolled in the PhD program in Computer Science at the University of Texas at San Antonio (UTSA) in 2019 and started working under the supervision of Dr. Xiaoyin Wang. Prior to this, he joined UTSA as a master's student in 2017, working under the guidance of Dr. Wei Wang. Sunzhou earned his Master of Science degree in Computer Science and Graduate Certificate in Cloud Computing from UTSA in 2018. Sunzhou received his Bachelor's degree from Zhejiang University of Science and Technology in 2012. Sunzhou's research interests include software engineering and cloud computing. More specifically, his research focuses on improving the efficiency of software failure reporting, facilitating build issue resolution, and leveraging AI to optimize the build experience.

ProQuest Number: 31997415

INFORMATION TO ALL USERS
The quality and completeness of this reproduction is dependent on the quality
and completeness of the copy made available to ProQuest.

Distributed by
ProQuest LLC a part of Clarivate ( 2025).
Copyright of the Dissertation is held by the Author unless otherwise noted.

This work is protected against unauthorized copying under Title 17,
United States Code and other applicable copyright laws.

This work may be used in accordance with the terms of the Creative Commons license
or other rights statement, as indicated in the copyright statement or in the metadata
associated with this work. Unless otherwise specified in the copyright statement
or the metadata, all rights are reserved by the copyright holder.

ProQuest LLC
789 East Eisenhower Parkway
Ann Arbor, MI 48108 USA