

**UC Davis**

**UC Davis Electronic Theses and Dissertations**

**Title**

Modern Software Engineering: Reproducibility, Reliability, and Scalability

**Permalink**

<https://escholarship.org/uc/item/3sv6k634>

**ISBN**

9798189226694

**Author**

Zhu, Hao-Nan

**Publication Date**

2026-06-25

Peer reviewed|Thesis/dissertation

Modern Software Engineering:  
Reproducibility, Reliability, and Scalability

*by*

HAO-NAN ZHU  
DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

*in*

Computer Science

*in the*

OFFICE OF GRADUATE STUDIES

*of the*

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

---

Cindy Rubio-González, Chair

---

Tapti Palit

---

Aditya V. Thakur

Committee in Charge

2026

Copyright © 2026 by Hao-Nan Zhu

*All rights reserved.*

*To where I come from.*

# Contents

|                                                                  |            |
|------------------------------------------------------------------|------------|
| <b>Abstract</b>                                                  | <b>ix</b>  |
| <b>Acknowledgments</b>                                           | <b>xii</b> |
| <b>1 Introduction</b>                                            | <b>1</b>   |
| 1.1 Reproducibility of Software Engineering Artifacts . . . . .  | 2          |
| 1.2 Reliability in the LLM Era of Software Engineering . . . . . | 3          |
| 1.3 Scalability of Modern Software Infrastructure . . . . .      | 4          |
| 1.4 Contributions . . . . .                                      | 5          |
| 1.5 Dissertation Outline . . . . .                               | 6          |
| <b>I Reproducibility</b>                                         | <b>8</b>   |
| <b>2 A Comprehensive Survey of Software Defect Datasets</b>      | <b>9</b>   |
| 2.1 Survey Methodology . . . . .                                 | 13         |
| 2.2 Scope of Software Defect Datasets . . . . .                  | 16         |
| 2.2.1 Application Domains of Defective Software . . . . .        | 16         |
| 2.2.2 Types of Defects . . . . .                                 | 23         |
| 2.2.3 Programming Languages of Defective Software . . . . .      | 28         |
| 2.3 Dataset Construction . . . . .                               | 30         |
| 2.3.1 Source of Defects . . . . .                                | 31         |
| 2.3.2 Construction Methodology . . . . .                         | 37         |
| 2.4 Availability and Usability . . . . .                         | 45         |
| 2.4.1 Availability and Hosting Platforms . . . . .               | 46         |
| 2.4.2 Presentation . . . . .                                     | 48         |
| 2.5 Uses of Software Defect Datasets . . . . .                   | 49         |
| 2.5.1 Empirical Research . . . . .                               | 52         |
| 2.5.2 Technical Evaluations . . . . .                            | 53         |
| 2.6 Discussion and Opportunities for Future Work . . . . .       | 54         |
| 2.6.1 Creation of Software Defect Datasets . . . . .             | 54         |
| 2.6.2 Maintenance of Software Defect Datasets . . . . .          | 57         |
| 2.7 Threats to Validity . . . . .                                | 58         |
| Chapter Summary . . . . .                                        | 58         |

|           |                                                            |            |
|-----------|------------------------------------------------------------|------------|
| <b>3</b>  | <b>Reproducing the Software Engineering Pipeline</b>       | <b>60</b>  |
| 3.1       | On the Reproducibility of Software Defect Datasets         | 60         |
| 3.2       | Reproducibility of Java Defect Datasets                    | 64         |
| 3.2.1     | Reproducibility Criteria & Strategies                      | 65         |
| 3.2.2     | Software Breakages in Java Defect Datasets                 | 68         |
| 3.3       | A Case Study on Reproducibility                            | 73         |
| 3.3.1     | Breakage Frequency                                         | 74         |
| 3.3.2     | Breakage Root Causes and Fixes                             | 77         |
| 3.3.3     | Breakage Prevention                                        | 80         |
| 3.4       | Threats to Validity                                        | 85         |
| 3.5       | Related Work on Defect Dataset Reproducibility             | 86         |
| 3.6       | Reproducing GitHub Actions                                 | 88         |
| 3.7       | Background                                                 | 90         |
| 3.8       | Approach                                                   | 91         |
| 3.8.1     | Environment Reconstruction                                 | 91         |
| 3.8.2     | Build Script Generation                                    | 92         |
| 3.8.3     | Image Packaging                                            | 93         |
| 3.8.4     | Limitations of ACTIONSREMAKER                              | 94         |
| 3.9       | Evaluation                                                 | 94         |
| 3.9.1     | Experimental Setup                                         | 94         |
| 3.9.2     | Effectiveness at Reproducing Real-World Builds             | 95         |
| 3.9.3     | Root Causes of Unreproducible Builds                       | 96         |
| 3.10      | Related Work on Reproducing CI/CD Builds                   | 97         |
|           | Chapter Summary                                            | 98         |
| <br>      |                                                            |            |
| <b>II</b> | <b>Reliability</b>                                         | <b>99</b>  |
| <br>      |                                                            |            |
| <b>4</b>  | <b>Reliability in the LLM Era of Software Engineering</b>  | <b>100</b> |
| 4.1       | Delta Debugging for LLM-Integrated Systems                 | 100        |
| 4.2       | Approach                                                   | 104        |
| 4.2.1     | Trace Generation from Chain of Thought                     | 105        |
| 4.2.2     | Delta Debugging with Semantic Markers                      | 107        |
| 4.2.3     | Development Lifecycle Integration                          | 111        |
| 4.3       | Evaluation                                                 | 112        |
| 4.3.1     | Fidelity of Chain-of-Thought Traces                        | 113        |
| 4.3.2     | Effectiveness of Delta Debugging based on Semantic Markers | 116        |
| 4.3.3     | Delta Debugging-Guided Improvement                         | 121        |
| 4.4       | Related Work on Debugging LLM-Integrated Systems           | 124        |
| 4.5       | Evaluating Large Language Model Tools on CI Failures       | 126        |
| 4.6       | Overview of CI-BENCH                                       | 129        |
| 4.7       | Experimental Evaluation                                    | 131        |
| 4.8       | Threats to Validity                                        | 134        |
| 4.9       | Related Work on Benchmarking LLM-Based Program Repair      | 135        |
|           | Chapter Summary                                            | 135        |

|            |                                                                    |            |
|------------|--------------------------------------------------------------------|------------|
| <b>III</b> | <b>Scalability</b>                                                 | <b>137</b> |
| <b>5</b>   | <b>Detecting Scalability Faults in Distributed Systems</b>         | <b>138</b> |
| 5.1        | Empirical Study of Scalability Faults . . . . .                    | 142        |
| 5.1.1      | <i>Compute</i> Faults . . . . .                                    | 145        |
| 5.1.2      | <i>Unbound</i> Faults . . . . .                                    | 146        |
| 5.1.3      | <i>Bloat</i> Faults . . . . .                                      | 148        |
| 5.1.4      | <i>Logic</i> Faults . . . . .                                      | 149        |
| 5.1.5      | Key Insights . . . . .                                             | 150        |
| 5.2        | Our Approach: SCALENS . . . . .                                    | 151        |
| 5.2.1      | SCALEVIEW . . . . .                                                | 152        |
| 5.2.2      | SCALEPICK . . . . .                                                | 155        |
| 5.3        | Evaluation of SCALENS . . . . .                                    | 158        |
| 5.3.1      | Experimental Setup . . . . .                                       | 159        |
| 5.3.2      | Effectiveness and Comparison with Baseline . . . . .               | 160        |
| 5.3.3      | Finding Unknown Scalability Faults . . . . .                       | 161        |
| 5.4        | Threats to Validity . . . . .                                      | 163        |
| 5.4.1      | Construct Validity . . . . .                                       | 163        |
| 5.4.2      | Internal Validity . . . . .                                        | 164        |
| 5.4.3      | External Validity . . . . .                                        | 165        |
| 5.4.4      | Conclusion Validity . . . . .                                      | 166        |
| 5.5        | Related Work . . . . .                                             | 167        |
|            | Chapter Summary . . . . .                                          | 169        |
| <b>6</b>   | <b>Toward Scalable LLM Serving: Bottlenecks and New Dimensions</b> | <b>170</b> |
| 6.1        | Background . . . . .                                               | 174        |
| 6.2        | Study Methodology . . . . .                                        | 175        |
| 6.3        | Scaling Dimensions . . . . .                                       | 177        |
| 6.4        | Symptoms . . . . .                                                 | 180        |
| 6.5        | Root Causes . . . . .                                              | 184        |
| 6.6        | Fix Strategies . . . . .                                           | 188        |
| 6.7        | Discussion . . . . .                                               | 191        |
| 6.7.1      | Implications for Serving System Designers . . . . .                | 191        |
| 6.7.2      | Implications for Tool Builders . . . . .                           | 193        |
| 6.7.3      | Implications for Operators . . . . .                               | 194        |
| 6.8        | Threats to Validity . . . . .                                      | 194        |
| 6.9        | Related Work . . . . .                                             | 195        |
|            | Chapter Summary . . . . .                                          | 198        |
| <b>7</b>   | <b>Conclusions and Future Work</b>                                 | <b>199</b> |
| 7.1        | Summary of Contributions . . . . .                                 | 199        |
| 7.2        | Broader Impact . . . . .                                           | 201        |
| 7.3        | Future Directions . . . . .                                        | 201        |
| 7.4        | Concluding Remarks . . . . .                                       | 202        |

# List of Figures

|      |                                                                                                          |     |
|------|----------------------------------------------------------------------------------------------------------|-----|
| 2.1  | Topics covered in this survey. . . . .                                                                   | 11  |
| 2.2  | Collection and selection process for software defect dataset papers. . . . .                             | 14  |
| 2.3  | General statistics of software defect dataset papers. . . . .                                            | 16  |
| 2.4  | Breakdown of types of defects in each application domain. . . . .                                        | 24  |
| 2.5  | Trend of programming languages in software defect datasets. . . . .                                      | 28  |
| 2.6  | Programming languages across domains. . . . .                                                            | 29  |
| 2.7  | Defect types by programming languages. . . . .                                                           | 29  |
| 2.8  | Methods for collecting defects (top) from stages in the software development processes (bottom). . . . . | 31  |
| 2.9  | Availability and hosting platforms of software defect datasets. . . . .                                  | 46  |
| 2.10 | Uses of software defect datasets. . . . .                                                                | 52  |
| 3.1  | Count of artifacts with a number of breakages. . . . .                                                   | 75  |
| 3.2  | Number of newly reproducible and newly broken artifacts for each test suite. . . . .                     | 76  |
| 3.3  | Count of artifacts with a number of patches. . . . .                                                     | 77  |
| 3.4  | Reproducibility of cached and isolated artifacts. . . . .                                                | 82  |
| 3.5  | Steps of ACTIONSREMAKER. . . . .                                                                         | 90  |
| 4.1  | Semantic marker injection and chain-of-thought trace generation. . . . .                                 | 106 |
| 4.2  | Delta debugging via binary partitioning. . . . .                                                         | 109 |
| 4.3  | Delta debugging via suspiciousness scoring. . . . .                                                      | 110 |
| 4.4  | Reference accuracy of semantic markers. . . . .                                                          | 114 |
| 4.5  | Mutation testing results for chain-of-thought trace authenticity validation. . . . .                     | 115 |
| 4.6  | Components of CI-BENCH. . . . .                                                                          | 128 |
| 4.7  | Patches for <code>tananaev-traccar-64783123</code> . . . . .                                             | 131 |
| 5.1  | YARN-6188, an example of scalability faults. . . . .                                                     | 139 |
| 5.2  | Scalability fault anti-patterns. . . . .                                                                 | 144 |
| 5.3  | Code samples for the <i>compute</i> , <i>unbound</i> , and <i>bloat</i> faults. . . . .                  | 145 |
| 5.4  | SCALELENS workflow. . . . .                                                                              | 152 |
| 5.5  | Scaling workload example. . . . .                                                                        | 153 |
| 5.6  | Runtime instrumentation. . . . .                                                                         | 154 |
| 5.7  | Growth patterns. . . . .                                                                                 | 156 |
| 6.1  | Distribution of symptom categories across scaling dimensions. . . . .                                    | 183 |
| 6.2  | Distribution of root cause categories across scaling dimensions. . . . .                                 | 187 |



|     |                                                                          |     |
|-----|--------------------------------------------------------------------------|-----|
| 6.3 | Distribution of root cause categories across symptom categories. . . . . | 188 |
| 6.4 | Distribution of fix strategies across root cause categories. . . . .     | 190 |

# List of Tables

|      |                                                                     |     |
|------|---------------------------------------------------------------------|-----|
| 2.1  | Comparison between surveys of software defect datasets. . . . .     | 12  |
| 2.2  | Searching strategies to collect papers. . . . .                     | 13  |
| 2.3  | Application domains of defective software. . . . .                  | 17  |
| 2.4  | Types of defects. . . . .                                           | 23  |
| 2.5  | Programming languages targeted in defect datasets. . . . .          | 27  |
| 2.6  | Source of defects. . . . .                                          | 32  |
| 2.7  | Collection methodologies of defects. . . . .                        | 38  |
| 2.8  | Validation methodologies of defects. . . . .                        | 42  |
| 2.9  | Categorization methodologies of defects. . . . .                    | 44  |
| 2.10 | Availability of artifacts of surveyed datasets. . . . .             | 47  |
| 2.11 | Presentation of datasets. . . . .                                   | 48  |
| 3.1  | Datasets for reproducibility study. . . . .                         | 65  |
| 3.2  | Criteria of reproducibility for datasets. . . . .                   | 68  |
| 3.3  | Reproducibility of software defect datasets. . . . .                | 70  |
| 3.4  | Root causes and patches for breakages. . . . .                      | 76  |
| 3.5  | Reproducibility without internet access. . . . .                    | 83  |
| 3.6  | Fail-pass pair counts for each type of unreproducible pair. . . . . | 96  |
| 4.1  | Evaluation results with CI-BENCH. . . . .                           | 133 |
| 5.1  | Breakdown of faults in the empirical study. . . . .                 | 140 |
| 5.2  | Evaluation results for SCALELENS. . . . .                           | 158 |
| 5.3  | Ablation analysis of SCALELENS. . . . .                             | 162 |
| 6.1  | Summary of issue collection pipeline per system. . . . .            | 177 |
| 6.2  | Distribution of scalability faults by scaling dimension. . . . .    | 178 |
| 6.3  | Distribution of scalability faults by symptom category. . . . .     | 181 |
| 6.4  | Distribution of scalability faults by root cause category. . . . .  | 184 |
| 6.5  | Distribution of scalability faults by fix strategy. . . . .         | 189 |

# Abstract

This dissertation investigates three structural concerns that determine whether modern software engineering can sustain trustworthy computational systems: the reproducibility of software engineering artifacts, the reliability in the large language model (LLM) era of software engineering, and the scalability of modern software infrastructure.

Reproducibility determines whether software artifacts and empirical results can remain executable and trustworthy over time. However, while maintaining BUGSWARM, one of the software defect datasets that set the foundation for the evaluation of software engineering techniques, the concern of reproducibility of software engineering research became increasingly apparent. To examine this concern, this dissertation presents a comprehensive survey of 151 software defect datasets and the first systematic study of reproducibility across five widely used Java defect datasets. This study finds that reproducibility criteria vary across datasets and that all datasets are prone to breakages. A 13-month longitudinal case study of BUGSWARM then shows that 62.6% of artifacts break at least once and identifies the dominant root causes of these breakages. It further demonstrates that dependency caching and artifact isolation can preserve reproducibility at 95%, matching the level achieved by handcrafted curation. To support the continuous growth of reproducible software defect datasets, this dissertation also presents ACTIONSREMAKER, which extends this line of work to GitHub Actions, the dominant CI/CD platform for open-source software. ACTIONSREMAKER demonstrates its ability to reproduce 72.2% of randomly selected fail-pass pairs. It has since been incorporated as a key component of the BUGSWARM pipeline, mining

thousands of reproducible artifacts to date and establishing continuous integration as a sustainable source of dataset growth.

Reliability determines whether software systems, including those that incorporate LLMs, can be analyzed and repaired when they fail. However, due to the probabilistic and black-box nature of LLMs, new challenges arise for the reliability in the LLM era of software engineering. To examine this concern, this dissertation introduces semantic markers, an instrumentation technique that adapts delta debugging to LLM-integrated systems. Semantic markers isolate the input fragments responsible for incorrect outputs without requiring access to model internals. The approach, evaluated on Amazon Web Services production systems, demonstrates that prompt debugging can be transformed from an ad hoc manual process into a systematic procedure. Complementarily, to assess the reliability of LLMs themselves when applied to software engineering tasks, this dissertation also presents CI-BENCH, a framework for evaluating LLM-based program repair tools on real CI/CD failures. The evaluation results with CI-BENCH reveal that the strongest tool-model pairing repairs only 35% of these failures, quantifying how far LLM-based repair remains from broad real-world CI/CD coverage.

Scalability determines whether software infrastructures continue to behave correctly as workloads, data, models, and deployments grow. However, in the shadow of scaling laws, scalability faults emerge as distinct failure modes in modern software infrastructure. To examine this concern, this dissertation conducts the largest empirical study of scalability faults in distributed systems, manually analyzing 444 issues across 10 systems. The study finds that most scalability faults arise from the interaction between dimensional code fragments and a small set of scalability anti-patterns. Building on this finding, this dissertation presents SCALENS, which combines lightweight dynamic analysis with static pattern matching to detect such faults. SCALENS identifies  $4.2\times$  more dimensional code fragments than a memory-monitoring baseline and surfaces 334 latent issues in the latest Cassandra, HDFS, and Ignite codebases. To extend this characterization to modern AI infrastructure, this

dissertation also studies 109 scalability faults across Ray Serve, vLLM, and SGLang along four orthogonal axes. The analysis reveals a control-plane/data-plane bifurcation specific to LLM-serving stacks and identifies a small set of defect categories as the highest-leverage class for systematic detection and repair.

# Acknowledgments

From 2021 to 2026, my Ph.D. has been carried by many people. Each of them shaped some part of this work, or the years behind it, in ways I will not forget.

My deepest gratitude goes to my advisor, Professor Cindy Rubio-González. Cindy has been more than an advisor to me; her way of thinking and doing has shaped me far beyond this dissertation. I will always remember the first winter of my Ph.D., when we worked together toward our first paper submission. Through those many meetings, revisions, and discussions, I began to learn from Cindy how to shape an idea, make sense of results, and turn everything into a paper. Since then, Cindy has given me the freedom to build my own research path, but I always knew that I was not walking it alone. Cindy is the best advisor I could have ever asked for, and I consider myself extraordinarily fortunate to have been her student.

I am also grateful to the members of my dissertation and qualifying examination committees: Professors Slobodan Mitrović, Tapti Palit, Caleb Stanford, and Aditya V. Thakur. Their incisive questions and constructive feedback, offered with thoughtfulness and rigor, have shaped both the substance and the presentation of this dissertation.

I have been fortunate to work with a remarkable group of collaborators and mentors during my Ph.D. I thank Professor Haryadi S. Gunawi and Dr. Cesar A. Stuardo at the University of Chicago, as well as Professor Michael Pradel at the University of Stuttgart, for our academic collaboration. I also thank those who supported me during my industry internships: Zeya Chen, Tancrède Lepoint, Muhammed Numair Mansur, Martin Schäf,

and Willem Visser at Amazon Web Services; and Shan Lu and Suman Nath at Microsoft Research. I am grateful to all of them for their generosity with time, ideas, and trust.

I thank my colleagues in the lab, Goodness Ayinmode, Patrick J. Chapman, Robert M. Furth, Kevin Z. Guan, Ryan A. Jae, Haneul Lee, Dolores Miao, Raian Latif Nabil, Jackson Vanover, and Yutong Wang, for their friendship, feedback, and shared journey throughout my Ph.D.

My thanks also go to those who shared moments along the way: Sophia Bareiss, Sonia Bhaskaran, Shuyu Cai, Yang Cao, Lei Chen, Shujuan Chen, Ximin Chen, Yuqing Dai, Qianfei Ding, Xinhui Fu, Jiawei Gu, Xinzhuo Hu, Sunzhou Huang, Dakai Kang, Heather Koh, John C. Kolesar, Weixian Lan, Jason Lefever, Haoyuan Li, Ninghui Li, Dingning Liu, Jiahui Lou, Henry Ouyang, Nichole D. Peterson, Buster Porter, Joanna M. Siebert, Haokun Su, Qingyi Teng, Xin Tong, Xiatian Wu, Yue Wu, Yuhao Xia, Zhiqiang Xie, Weiting Xu, Yujing Yang, Zixiao Ye, Jinxiao Yu, Bo Zhang, Yixin Zhang, Xuebao Zhao, Yongyi Zhou, Jiaqi Zong, and Minfei Zou. Beyond all of this, my special thanks go to my family, especially Yi Ren, for their love, patience, and support throughout this journey.

Finally, I deeply appreciate Davis, this little cowtown, and all the nice people here, whose kindness reshaped my personality and my way of being.

# Chapter 1

## Introduction

Software has become the substrate of civilization. It mediates how people communicate, how markets operate, how organizations deliver services, and how science advances. Over the past decade, software systems have evolved from individual programs into globally distributed infrastructures, operating at unprecedented scale, complexity, and societal impact. More recently, large language models (LLMs) have been integrated into software engineering pipelines and production services, enabling systems to perform tasks that were once beyond the reach of traditional code. As a result, software engineering is no longer only about constructing correct programs. It is about sustaining the trustworthiness of the computational foundations on which modern society depends.

Yet these foundations are fragile in ways that are increasingly difficult to observe, reproduce, and repair. The research artifacts that the software engineering community relies on to evaluate techniques can silently decay as their build environments, dependencies, and execution platforms evolve. The failure modes of LLMs embedded in production software services resist conventional debugging techniques that depend on inspectable program state. Distributed systems and AI-serving infrastructures exhibit faults that remain latent at the test scales available to most developers, yet become catastrophic at the production scales where they matter most. Together, these challenges reveal a central tension in modern soft-



ware engineering: the systems we depend on are becoming more powerful and pervasive, while their failures are becoming harder to reproduce, localize, and understand.

This dissertation addresses that tension through three fundamental concerns of modern software engineering: the reproducibility of software engineering artifacts, the reliability in the LLM era of software engineering, and the scalability of modern software infrastructure. Reproducibility determines whether software artifacts and empirical results can remain executable and trustworthy over time. Reliability determines whether systems, including those that incorporate LLMs, can be analyzed and repaired when they fail. Scalability determines whether software infrastructures continue to behave correctly as workloads, data, models, and deployments grow. Across these three concerns, this dissertation examines how software engineering artifacts, LLM-integrated systems, and modern software infrastructures fail, and develops engineered tools that preserve, expose, and mitigate those failures in practice.

## 1.1 Reproducibility of Software Engineering Artifacts

Software defect datasets, e.g., BUGSWARM [351], are essential resources for empirical research and the standardized benchmarking of techniques in fault localization, test generation, automated program repair, and the rapidly emerging space of LLM-based software engineering tools. Numerous defect datasets have been developed over the years, yet picking the right benchmark is itself a research problem. For this, [Chapter 2](#) surveys 151 defect datasets along their scope, construction, availability, usability, and actual uses by the community, and identifies underrepresented kinds of defects as opportunities for future work.

Reproducibility is a precondition for any of these datasets to remain useful, but it is not guaranteed. [Chapter 3](#) presents the first systematic study of reproducibility across five widely used Java defect datasets, finding that reproducibility criteria differ across them and that all five are prone to breakages. A 13-month longitudinal case study of BUGSWARM then shows that 62.6% of 1,795 Java artifacts break at least once and 15.3% break multiple

times. We manually investigate the root causes and apply 10 handcrafted patches that automatically repair 1,055 distinct artifacts across 2,948 individual fixes. Building on these patches, we propose two reproducibility-preserving mechanisms, dependency caching and artifact isolation, and show that isolating artifacts preserves reproducibility at 95%, on par with the most reliable hand-curated dataset.

GitHub Actions has become the dominant CI/CD platform for open-source software, making it a critical source for future defect dataset growth. The same chapter presents ACTIONSREMAKER, the first tool designed to reproduce GitHub Actions workflow runs. ACTIONSREMAKER demonstrates its ability to reproduce 130 of 180 (72.2%) randomly selected fail-pass build pairs from 43 repositories. ACTIONSREMAKER has since been incorporated as a key component of the BUGSWARM pipeline, mining thousands of reproducible artifacts to date and establishing continuous integration as a sustainable source of defect dataset growth.

## 1.2 Reliability in the LLM Era of Software Engineering

LLMs are increasingly integrated into software services as automated decision-making components, taking on tasks that traditional code cannot. Unlike conventional code, an LLM provides no execution trace, no inspectable intermediate state, and no deterministic behavior, even at temperature zero. When a model produces an undesired output, engineers are often left with a long, multi-section prompt in which any fragment could be responsible, while repeated executions may yield different results. [Chapter 4](#) introduces semantic markers, an instrumentation technique that embeds unique identifiers in LLM inputs and recovers traceability information from model reasoning. Combined with classical delta debugging, semantic markers transform prompt debugging from an ad hoc manual process into a systematic methodology for isolating the minimal input subset responsible for unexpected behavior. This approach has demonstrated its effectiveness on both controlled benchmarks and Amazon Web Services production systems.

Beyond debugging individual LLM-integrated systems, a broader reliability question is whether LLM-based repair tools can generalize to real software failures. [Chapter 4](#) also presents CI-BENCH, a unified framework for evaluating LLM-based program repair on real CI/CD failures from BUGSWARM. CI-BENCH retrieves build failures, parses logs, constructs repair prompts, executes generated patches in faithful reconstructions of the original environments, and reports correctness through plausibility, syntactic equivalence, and manual inspection. We evaluate three state-of-the-art repair tools across three foundation models on 100 real artifacts. The strongest tool-model pairing repairs only 35% of these failures, establishing a baseline that quantifies how far LLM-based repair remains from broad real-world CI/CD coverage.

## 1.3 Scalability of Modern Software Infrastructure

A scalability fault is a fault that remains latent at the test scales available to developers and manifests only when an operational dimension, such as the number of nodes, data volume, concurrent requests, model size, or context length, crosses a critical threshold. These faults often remain hidden during ordinary development and testing, only to surface in production environments where their cost is high.

[Chapter 5](#) presents the largest empirical study of scalability faults to date, manually analyzing 444 scalability issue reports across 10 large-scale distributed systems, including Apache Hadoop, HBase, Cassandra, HDFS, and Spark. The study finds that most scalability faults arise from the interaction between dimensional code fragments (DCFs), pieces of code whose cost grows with a system dimension, and a small set of scalability anti-patterns associated with them. Based on this finding, SCALELENS combines lightweight dynamic analysis with static pattern matching to identify DCFs and their associated anti-patterns. On a benchmark of known scalability faults from Cassandra, HDFS, and Ignite, SCALELENS improves DCF detection over a memory-monitoring baseline by  $4.2\times$ . On the latest

stable versions of the same three systems, it surfaces 334 previously unreported DCFs with confirmed problematic behavior.

LLM-serving infrastructures inherit the classical scalability faults of distributed systems while introducing new scaling dimensions specific to the GPU serving stack. [Chapter 6](#) extends the analysis of scalability faults to this new layer of the software stack. We mine tens of thousands of issue reports from Ray Serve, vLLM, and SGLang and identify 109 scalability faults, which we classify along four orthogonal axes: scaling dimension, symptom, root cause, and fix strategy. These faults bifurcate into a control-plane regime in the serving framework and a data-plane regime in the inference engines, with near-disjoint profiles across all four axes. Cross-tabulating these axes reveals strong correlations between root causes and fix strategies, and identifies a small set of defect categories as the highest-leverage class for systematic detection and repair.

## 1.4 Contributions

This dissertation makes the following contributions:

1. **A comprehensive survey of software defect datasets** ([Chapter 2](#)). A systematic survey of 151 software defect datasets, characterizing their scope, construction, availability, usability, and actual uses by the research community. The catalog is available at <https://defect-datasets.github.io/>.
2. **An empirical and practical foundation for reproducible defect datasets** ([Chapter 3](#)). A systematic study of reproducibility across widely used software defect datasets, an analysis of the root causes of breakages in BUGSWARM, mechanisms for preserving reproducibility through dependency caching and artifact isolation, and ACTIONSREMAKER, an open-source tool for reproducing GitHub Actions workflow runs and mining reproducible fail-pass pairs from the GitHub Actions ecosystem. The artifact of the reproducibility study is available at <https://github.com/ucd-plse/>

[on-the-reproducibility](#), and ACTIONSREMAKER is available at <https://github.com/bugswarm/actions-remaker>.

3. **Systematic methodology and benchmarking framework for LLM-related software engineering** (Chapter 4). A methodology based on semantic markers and delta debugging for isolating failure-inducing input fragments in LLM-integrated systems, and CI-BENCH, a framework for evaluating LLM-based program repair on real CI/CD failures. The delta debugging approach for LLM-integrated systems has been implemented as development tooling at Amazon Web Services, and CI-BENCH is available at <https://github.com/bugswarm/ci-bench>.
4. **A fault model and detection tool for scalability faults in distributed systems** (Chapter 5). An empirical study of scalability faults that establishes the synergy between dimensional code fragments and anti-patterns as the dominant fault structure, and SCALELENS, a tool that combines lightweight dynamic analysis with static pattern matching to detect such faults. The empirical data and analysis are available at <https://github.com/ucd-plse/scalability>, and SCALELENS is available at <https://github.com/ucd-plse/scalelens>.
5. **A taxonomy of scalability issues in LLM-serving infrastructures** (Chapter 6). A characterization of code-level scalability faults specific to the LLM-serving stack, derived from manually analyzed faults across Ray Serve, vLLM, and SGLang. The taxonomy reveals the control-plane/data-plane structure of LLM-serving failures and identifies fault classes that are promising targets for systematic detection and repair.

## 1.5 Dissertation Outline

The dissertation is organized in three Parts.

**Part I (Reproducibility)** comprises Chapter 2, which surveys the software defect

dataset landscape, and [Chapter 3](#), which studies reproducibility issues and presents ACTIONSREMAKER.

**Part II (Reliability)** consists of [Chapter 4](#), which presents the delta debugging methodology with semantic markers in its first half and CI-BENCH in its second half.

**Part III (Scalability)** comprises [Chapter 5](#), which presents the empirical study and SCALELENS, and [Chapter 6](#), which extends the analysis to LLM-serving infrastructure.

[Chapter 7](#) concludes the dissertation.

# Part I

## Reproducibility

## Chapter 2

# A Comprehensive Survey of Software Defect Datasets

Given the escalating scale and complexity of real-world software systems, software defects are inevitable and have become increasingly challenging to manage [113, 139]. Software defects, commonly referred to as bugs, are unintentional errors in software that can cause software systems to behave unexpectedly or fail to meet their intended requirements. To address the challenges posed by software defects, in the last few decades, a variety of techniques have been proposed to detect, localize, and fix them. These techniques include, but are not limited to, static and dynamic analysis [80, 84], test generation and prioritization [136, 261], fault detection and localization [381], and automated program repair [208]. To evaluate the effectiveness and efficiency of these techniques, the research community heavily relies on software defect datasets, which serve as foundational resources for benchmarking and validation.

Software defect datasets are curated collections of real-world software bugs. These datasets serve as standardized benchmarks, fostering consistent and reproducible evaluations of novel techniques across diverse studies. Additionally, the realism, diversity, scale, and accessibility of these datasets support empirical analyses of defect characteristics and



enable the training and fine-tuning of machine learning models for software quality assurance. For instance, the widely adopted Defects4J dataset [190] has been used for a wide range of technical evaluations (e.g., [245, 276, 447]), empirical studies (e.g., [134, 325]), and to train machine learning models (e.g., [429]). As another example, the recently proposed SWE-Bench [187] dataset is widely used to evaluate the effectiveness of large language models (LLMs) in automating software engineering [403, 434] and serves as the basis for other benchmarks [68, 257], demonstrating the practical value of software defect datasets.

With the continuous growth in the complexity and scale of modern software systems, there is an increasing demand for comprehensive and high-quality defect datasets. This demand is further amplified by the rising popularity of artificial intelligence (AI)-based software engineering approaches [175, 404]. In response to this escalating demand, industry challenges, competitions, and workshops have been organized to encourage the development of software defect datasets of high quality. As a result, these datasets have experienced a significant increase in both quantity and diversity in recent years, becoming essential resources for researchers and practitioners in multiple fields, including software engineering, programming languages, computer security, computer systems, and artificial intelligence.

With the increasing number of software defect datasets, researchers and practitioners face challenges in navigating the landscape of software defect datasets for two main reasons. First, different datasets may have different focuses, such as types of defects or programming languages, requiring additional effort to identify the most suitable dataset for specific research or practical needs. Second, based on how the datasets are constructed, they exhibit diverse characteristics, such as size and usability, affecting the impact of these datasets. Moreover, creators and maintainers of software defect datasets also encounter challenges in developing datasets that address the evolving needs of software engineering research and practice, as well as in ensuring the long-term sustainability and usability of the created datasets.

To address these challenges, this chapter provides a comprehensive overview of existing software defect datasets. Specifically, we follow a systematic methodology (detailed in Sec-

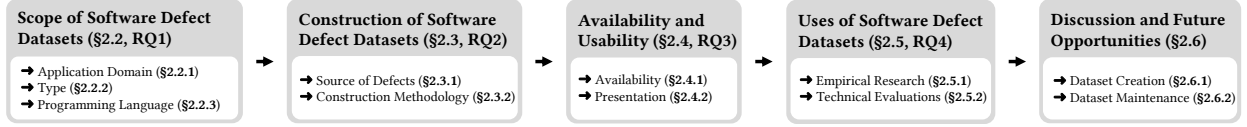


Figure 2.1: Topics covered in this survey.

tion 2.1) for collecting and filtering relevant literature, resulting in the identification of 151 software defect datasets. Our survey focuses on the following four research questions (RQs), as illustrated in Figure 2.1:

- **RQ1: What is the scope of existing software defect datasets? (Section 2.2)** For each dataset surveyed, we examine the application domain of the software systems from which the defects are collected (e.g., machine learning systems or mobile applications), the types of defects (e.g., functional, security, or performance), the programming languages (e.g., Java, C/C++, or Python), and other characteristics that define the focus of the dataset. We also explore potential interrelationships among these dimensions to provide a holistic understanding of the dataset landscape.
- **RQ2: How are existing software defect datasets constructed? (Section 2.3)** For each dataset, we investigate how the dataset is constructed. This includes the sources of defects (e.g., issue reports, version control systems, or continuous integration & delivery) and the methodologies used to obtain the defects (e.g., manual inspection, keyword matching, or machine learning techniques).
- **RQ3: To what extent are existing software defect datasets available and usable in practice? (Section 2.4)** For each dataset, we evaluate its availability and usability. Availability refers to whether the dataset is publicly available, and if so, where it is hosted. Usability refers to how the defects are presented in the dataset, such as providing only textual descriptions, code snippets, or executable artifacts to reproduce the defects.
- **RQ4: How are software defect datasets actually used by research community? (Section 2.5)** While prior literature reviews [171, 173] have provided high-level summaries

Table 2.1: Comparison between surveys of software defect datasets.

|                                      | Hirsch and Hofer<br>[171] | Holek et al. [173] | This Survey  |
|--------------------------------------|---------------------------|--------------------|--------------|
| Number of Datasets                   | 73                        | 14                 | 151          |
| Cutoff Date For Dataset Selection    | November 2021             | April 2022         | January 2025 |
| <b>Scope of Datasets</b>             |                           |                    |              |
| - Application Domains                | ✗                         | ✗                  | ✓            |
| - Types of Defects                   | ✗                         | ✗                  | ✓            |
| - Programming Languages              | ✓                         | ✗                  | ✓            |
| <b>Dataset Construction</b>          |                           |                    |              |
| - Source of Defects                  | ✗                         | ✓                  | ✓            |
| - Mining Methodology                 | ✗                         | ✗                  | ✓            |
| <b>Availability and Usability</b>    |                           |                    |              |
| - Availability and Hosting Platforms | ✓                         | ✓                  | ✓            |
| - Presentation                       | ✓                         | ✓                  | ✓            |
| <b>Uses of Datasets</b>              | ✗                         | ✗                  | ✓            |

of the intended uses of software defect datasets from the perspective of their creators, this research question seeks to offer a more comprehensive and detailed understanding of how these datasets are actually used. By analyzing citations of the most influential datasets, we aim to uncover usage patterns and trends over time, thereby revealing their impact on the research community.

Finally, based on our analysis of existing datasets, Section 2.6 identifies potential opportunities for future research, including automating the mining of high-quality datasets, e.g., by leveraging CI pipelines and LLMs. Additionally, we identify the need for specialized datasets targeting specific domains, defect types, and new languages, as well as techniques ensuring dataset reproducibility and continuous updates.

Our work contributes the first comprehensive survey on software defect datasets, providing a detailed discussion of their scope, construction, availability, and real-world usage. Orthogonal to existing surveys on software defects, which focus on categorizing [260, 423], detecting/localizing [74, 105, 222, 279, 381], and repairing software defects [178, 253, 428], this survey focuses on datasets that collect software defects prevalent in software codebases. Previous literature reviews on 73 benchmarks [171] and 14 bug datasets [173] provide high-level overviews of existing benchmarking datasets. However, as summarized in Table 2.1, these studies do not cover several important aspects of software defect datasets. In contrast,

Table 2.2: Searching strategies to collect papers.

| Strategy              | Search Content                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Key Phrase            | bug benchmark, bug dataset, defect benchmark, defect dataset, patch benchmark, patch dataset, fail pass, fault benchmark, fault dataset, mining software, software benchmark, software bug, software dataset, software defect, software fault, software patch, software vulnerabilit(y/ies), vulnerabilit(y/ies) benchmark, vulnerabilit(y/ies) dataset, empirical bug, empirical defect, empirical error, empirical failure, empirical fault, empirical issue, empirical patch, study bug, study defect, study error, study failure, study fault, study issue, study patch |
| Keyword<br>×<br>Venue | benchmark, bug, dataset, defect, fault, issue, patch, vulnerabilit(y/ies)<br>ASE, ESE, FSE, ICSE, ISSRE, ISSTA, MSR, OOPSLA, SANER, TOSEM, TSE, ESEC, AAAI, ACL, EMNLP, ICLR, ICML, NeurIPS, USENIX Security, IEEE S&P, CCS, NDSS                                                                                                                                                                                                                                                                                                                                           |

this survey provides (1) significantly broader coverage of 151 software defect datasets with a more recent cutoff date, (2) a systematic and in-depth analysis of their scope, construction, availability, and usability, (3) insights into the relationships between different dataset characteristics, and (4) an examination of longitudinal trends in the actual usage of the most influential datasets.

We envision this survey to be useful for both dataset users and creators. For dataset users, this survey provides a comprehensive overview of existing software defect datasets, enabling them to identify the most suitable datasets for their research or practical needs. For dataset creators, our work highlights the current focuses and methodologies used in existing datasets, which can guide the development of new datasets that fill the gaps in the current literature. In addition to this chapter, we provide an interactive website<sup>1</sup> that allows users to find datasets based on specific criteria, such as programming languages or application domains of interest.

## 2.1 Survey Methodology

This section describes the methodology used in this survey to identify, select, and analyze existing software defect datasets, guided by research questions described in the introduction. Our survey covers 151 software defect datasets. To identify these datasets, we follow a

<sup>1</sup><https://defect-datasets.github.io/>

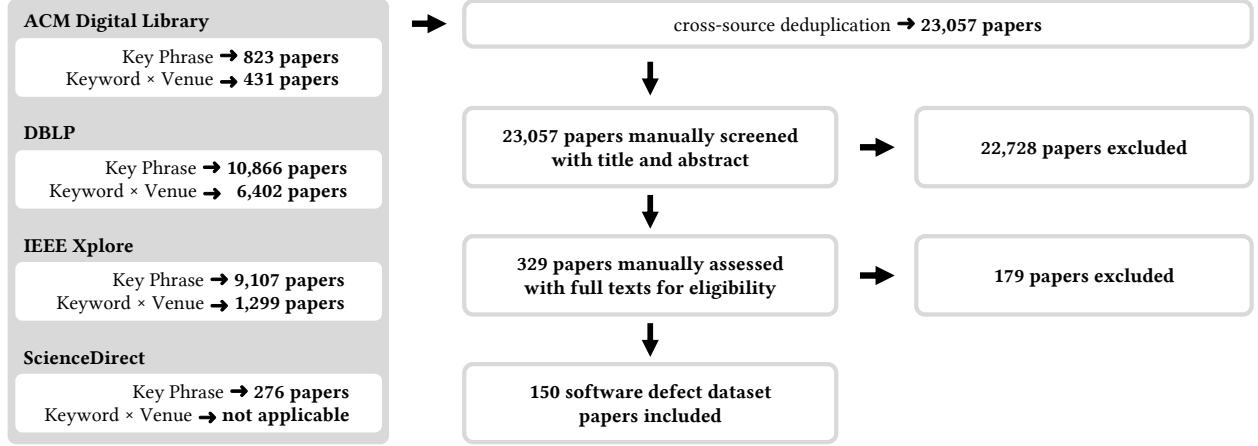


Figure 2.2: Collection and selection process for software defect dataset papers.

systematic process that collects papers from multiple bibliographic databases and selects papers through a rigorous manual review. Figure 2.2 illustrates the complete paper collection and selection process. We conduct a comprehensive literature search across four major bibliographic databases: ACM Digital Library, DBLP, IEEE Xplore, and ScienceDirect. For each source, we execute two complementary search strategies, summarized in Table 2.2. We include papers that have been peer-reviewed and published before January 1, 2025, excluding preprints.

**(1) Key Phrase-Based Search** We use specific key phrases to identify papers explicitly related to software defect datasets or empirical studies on software defects. We search for papers containing all words from a key phrase in their titles, abstracts, and keywords if available. This strategy yields 823 papers from the ACM Digital Library, 10,866 papers from DBLP, 9,107 papers from IEEE Xplore, and 276 papers from ScienceDirect.

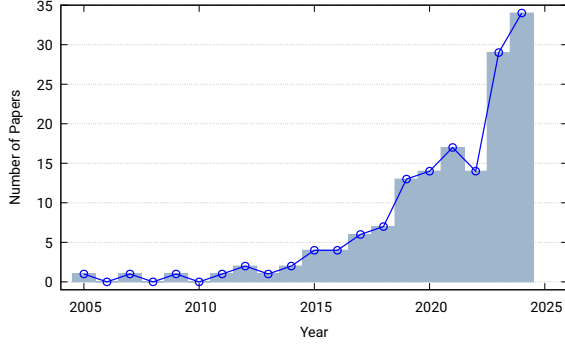
**(2) Keyword  $\times$  Venue Search** To complement the key phrase-based search, we use individual keywords combined with venue restrictions. As individual keywords yield many papers, we restrict this search to high-impact venues in software engineering, machine learning, and software security. We search for papers containing at least one keyword in their title, abstract, and keywords if available. This strategy yields 431 papers from the ACM

Digital Library, 6,402 papers from DBLP, and 1,299 papers from IEEE Xplore. Due to API query constraints, this strategy was not applicable to ScienceDirect.

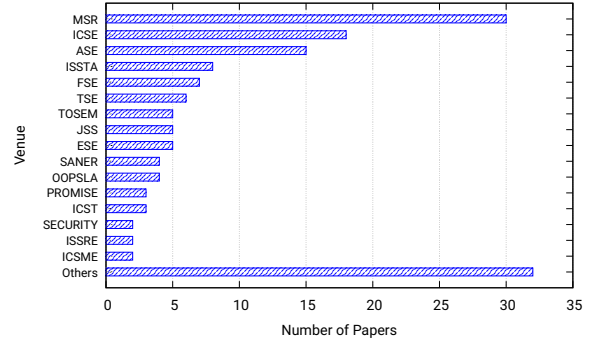
After merging results from both search strategies across all bibliographic databases and performing cross-source deduplication, we obtain 23,057 unique papers potentially related to software defect datasets. Two authors independently review the titles and abstracts of these papers to identify those potentially introducing a software defect dataset. The authors then compare their selections and reach consensus through discussion, excluding 22,728 irrelevant papers and identifying 329 papers for full-text assessment. For the full-text assessment phase, the same two authors independently review each of the 329 papers to determine their eligibility for inclusion. A paper is included if its main contribution is one or more software defect datasets, or if it includes software defects as byproducts. Papers are excluded if they only use existing datasets without introducing a new dataset, present datasets for purposes unrelated to software defects (e.g., code generation or performance benchmarking), or focus solely on techniques for mining, injecting, or labeling defects. The inter-rater agreement between the two authors for the full-text assessment is almost perfect, with a Cohen’s kappa [117] of 0.871 (95% CI: 0.818–0.924). The authors agreed on 308 out of 329 papers (93.6%), and disagreements on the remaining 21 papers were resolved through discussion and, when necessary, consultation with a third author. This rigorous assessment ultimately leads to 150 papers, from which we identified 151 software defect datasets (one paper [206] presents two datasets).

Finally, the same two authors conduct an in-depth manual review of all included datasets to extract and annotate their attributes (e.g., application domain, defect type, and programming language for RQ1; sources of defects and construction methodologies for RQ2; and availability and usability for RQ3). Each dataset is independently cross-checked by both authors to ensure consistency, with ambiguities resolved through discussion. The annotation process is performed manually over several months.

Figure 2.3a plots the number of papers related to software defect datasets published



(a) Number of papers by year.



(b) Number of papers by venue.

Figure 2.3: General statistics of software defect dataset papers.

by year, which shows a significant increase in the number of new software defect datasets published in the past eight years. The venues that publish papers related to software defect datasets are shown in Figure 2.3b, where different tracks of a conference are grouped together. The top venues that publish papers related to software defect datasets include MSR, ICSE, and ASE. All venues that have only one paper included in this study are grouped into the *others* category, such as ICLR and SoCC.

## 2.2 Scope of Software Defect Datasets

Software defect datasets often have specific purposes. In this section, we discuss and compare the scope of the various datasets. In particular, we consider three dimensions: (1) the application domain of the software that contains the defects (Section 2.2.1), (2) the type of defects that are included in the dataset (Section 2.2.2), and (3) the programming languages used in the defective software (Section 2.2.3).

### 2.2.1 Application Domains of Defective Software

The *application domain* of defective software refers to the specific field or area in which the software operates, such as machine learning or cloud computing. Of the 151 software defect datasets considered here, the majority (over 60%) focus on general software develop-

Table 2.3: Application domains of defective software. One dataset may appear in multiple categories. For clarity, only datasets that explicitly specify the corresponding attribute (i.e., application domain) are listed. The complete list of 151 datasets is available at <https://defect-datasets.github.io/> and in the supplementary materials [457]. The same convention applies to all tables in Sections 2.2 and 2.3.

| Category                    |                        | Datasets                                                | Count |
|-----------------------------|------------------------|---------------------------------------------------------|-------|
| Machine Learning            | <i>Applications</i>    | [111, 179, 199, 219, 255, 362, 432]                     | 7     |
|                             | <i>Frameworks</i>      | [158, 172, 230, 340, 440]                               | 5     |
|                             | <i>Compilers</i>       | [130, 317]                                              | 2     |
| Compilers & Runtime Engines |                        | [130, 282, 299, 317, 371, 373, 395, 409, 410, 414, 433] | 11    |
| Mobile Applications         |                        | [111, 123, 229, 252, 297, 311, 342, 344, 378, 391]      | 10    |
| Blockchain                  | <i>Smart Contracts</i> | [99, 106, 370, 427]                                     | 4     |
|                             | <i>Clients</i>         | [198]                                                   | 1     |
| Cloud Computing             |                        | [129, 146, 159, 396]                                    | 4     |
| Autonomous Systems          |                        | [148, 350, 360]                                         | 3     |
| Quantum Computing           | <i>Applications</i>    | [439]                                                   | 1     |
|                             | <i>Platforms</i>       | [278, 440]                                              | 2     |
| Scientific Computing        |                        | [81, 142, 258]                                          | 3     |
| Databases                   |                        | [119, 228]                                              | 2     |
| Computational Notebooks     |                        | [125]                                                   | 1     |
| Internet of Things          |                        | [243]                                                   | 1     |

ment and are not tied to any specific application domain. Among the remaining datasets that explicitly target specific domains, we identify eleven distinct application domains (Table 2.3), including machine learning, compilers and runtime engines, mobile applications, and blockchain.<sup>2</sup> Some domains attract more attention from dataset creators than others. For example, there are 14 datasets related to machine learning defects, which we further categorize into machine learning applications, machine learning frameworks, and machine learning compilers. The rest of this section discusses the application domains focused on by the software defect datasets, highlighting the uniqueness of defects in each domain.

## Defects Related to Machine Learning

The reliability and robustness of machine learning applications and their underlying software platforms are crucial for their successful deployment. Machine learning-related software

<sup>2</sup>In this and other classifications of datasets, a single dataset may appear in multiple categories.



defects can cause severe consequences, such as incorrect predictions, biased decisions, and performance degradation [432]. We identify three subdomains of machine learning-related software, each with several defect datasets: machine learning applications, machine learning frameworks, and machine learning compilers.

**Machine Learning Applications** Defects in machine learning applications often arise during model development, training, and deployment, leading to critical issues like incorrect predictions or silent errors. Datasets such as defect4ML [255], gDefects4DL [219], and Denchmark [199] focus on common machine learning defects, including API misuses, performance problems, and numerical errors. There are also machine learning defect datasets provided as by-products of empirical studies, such as studies on bugs from programs relying on TensorFlow [432], on numerical defects in deep learning programs [362], and on deployment-related bugs in machine learning-based mobile applications [111].

**Machine Learning Frameworks** Machine learning frameworks, such as PyTorch and TensorFlow, are crucial for building and optimizing models, but themselves may suffer from defects. Software defect datasets for machine learning frameworks typically result from empirical research. For example, MOB-Dataset [158] collects real-world machine learning optimization defects from TensorFlow and PyTorch, and comes with an empirical study of model optimization. Likewise, empirical studies of machine learning frameworks have produced defect datasets of common bugs in PyTorch [172] and quantum machine learning frameworks [440]. Other work presents a dataset and a defect taxonomy of aging-related bugs in machine learning libraries [230]. These studies provide critical insights into the defects that can impact the functionality, efficiency, and stability of machine learning frameworks.

**Machine Learning Compilers** In addition to applications and frameworks, machine learning compilers play a vital role in optimizing models, but they also introduce unique bugs. Based on datasets of defects in machine learning compilers, Du et al. [130] and Shen

et al. [317] both examine frequent issues in these compilers, such as incorrect optimizations, numerical inaccuracies, and incompatibilities with hardware accelerators.

## Defects in Compilers and Runtime Engines

Compilers and runtime engines are critical components of software systems, responsible for translating and executing code. Defects in compilers and runtime engines can significantly impact software performance, security, and execution correctness. We here mean by “runtime engine” both stand-alone implementations of a programming language, e.g., the Java Virtual Machine, and engines integrated into a larger software, e.g., a JavaScript engine embedded into a web browser. Datasets of defects in compilers and runtime engines often come as by-products of empirical research. For example, Yu and Wang [410] conduct an empirical study on bugs in implementations of the Rust language, presenting a dataset of 194 bugs from Rust compilers and Rust infrastructure. Xu et al. [395] investigate security bugs introduced by compilers for C/C++, highlighting the potential risks when silent bugs go undetected. In addition, as also mentioned above, Du et al. [130] and Shen et al. [317] study defects in deep-learning compilers, identifying common root causes and symptoms. For runtime engines, Bugs in Pods [409] provides 429 bugs in the container runtime systems containerd, cri-o, gvisor, and runc. Additionally, Paul et al. [282] provide a detailed dataset of security-related changes in the Chromium OS project, and Zaman et al. [414] present a dataset of performance-related bugs in web browsers, focusing on how such bugs degrade system performance. Besides browsers, datasets constructed from studies of other runtime engines include defects in WebAssembly engines [371, 433], and bugs that impact the correctness and performance of JavaScript engines [373]. Altogether, these datasets are critical for understanding and addressing defects in compilers and runtime environments.

## Defects in Mobile Applications

Mobile applications are prone to various types of defects, including functional bugs, security vulnerabilities, and resource issues. Several datasets, mostly based on Android applications, have been developed to represent these defects. For example, LVDAndro [311] and Ghera [252] focus on security vulnerabilities, providing labeled vulnerability datasets and benchmarks for AI-based detection models and vulnerability analysis. Functional bugs are collected by Andror2 [378] and by Xiong et al. [391], both of which analyze the behavior of defective mobile applications by manually reproducing the bugs. More specialized defects, such as data loss issues and resource leaks, are captured by DataLossRepository [297] and DroidLeaks [229], respectively. Finally, DroixBench [342] provides defective APK files particularly for evaluating automated repair techniques on Android applications. These datasets play a crucial role in enhancing the reliability and security of Android applications by supporting bug detection, repair, and vulnerability research.

## Defects Related to Blockchain

Blockchain, as a decentralized and distributed ledger technology, has been widely used in various applications, such as cryptocurrencies [444]. In general, blockchain applications are implemented as smart contracts, which are self-executing contracts with the terms of the agreement between multiple parties directly written into code. Defects in blockchain applications can have severe implications, such as financial loss. Several datasets focus on identifying and analyzing defects in smart contracts. From a general perspective, Jiuzhou [427] is a dataset of Ethereum smart contract bugs that are classified into 49 categories. Specifically for security vulnerabilities, SBP-Dataset [99] contains smart contract defects generated by removing guards and turning safe functions into unsafe ones. In addition, there are two datasets constructed from empirical studies. One defines 20 categories of smart contract bugs [106], and the other identifies common patterns of bug fixes [370]. Unlike the above datasets, Kim et al. [198] focus on energy bugs in Ethereum client software instead of smart

contracts. The dataset contains 507 bugs that may cause energy inefficiency mined from the buggy commits of client software operated in the Ethereum network.

### **Defects in Cloud Computing**

Cloud computing systems provide on-demand access to computing resources over the internet and are prone to various software defects that can affect performance, reliability, and security. Datasets of software defects in cloud computing often arise from empirical studies. For instance, both Gunawi et al. [159] and Gao et al. [146] investigate real-world issues in distributed cloud systems, with the latter focusing on crash recovery bugs. Additionally, Xu et al. [396] present a dataset of bugs in cloud orchestration systems, while Drosos et al. [129] provide a dataset of misconfigurations in cloud infrastructure-as-code scripts.

### **Defects in Autonomous Systems**

Autonomous systems, such as autonomous vehicles, aircraft, or robots, rely on robust software to operate safely and efficiently. Defects in autonomous systems can lead to severe consequences, such as accidents, injuries, and fatalities. Wang et al. [360] and Garcia et al. [148] both present empirical studies of software defects in autonomous systems. The former study has collected a dataset of 569 bugs from PX4 and ArduPilot, which are well-known open-source autopilot software systems for unmanned aerial vehicles. The latter study presents a dataset of 499 autonomous vehicle bugs from Apollo and Autoware. In addition, ROBUST [350] offers a dataset of 221 bugs from seven popular robot operating systems. Together, these datasets highlight the unique challenges caused by defects in autonomous systems.

### **Defects Related to Quantum Computing**

Defects in quantum computing systems, including both quantum applications and quantum platforms, present unique challenges due to the complexity and novelty of the technology.

Bugs4Q [439] provides a benchmark of bugs in quantum programs, particularly focusing on the Qiskit framework, enabling controlled testing and debugging studies. Additionally, there are defect datasets constructed within empirical studies, such as one that gathers and categorizes bug patterns across various quantum platforms [278], and another that focuses on defects within the intersection of quantum computing and machine learning [440]. These datasets are useful for advancing the reliability of quantum software systems.

## Defects Related to Scientific Computing

Scientific computing systems are often complex and require high performance and accuracy. Defects in scientific computing can lead to incorrect results, degraded performance, or even compromised system security. Murphy et al. [258] collect security defects from open-source scientific software projects written in Julia, a programming language designed for numerical analysis and computational science. Likewise, Azad et al. [81] build a dataset by investigating 1,729 performance bugs from 23 real-world high-performance computing (HPC) applications such as molecular simulation and computational fluid dynamics. In addition to scientific computing applications, Franco et al. [142] study numerical bugs across scientific computing libraries and present a dataset of 269 numerical bugs from six libraries, including NumPy, SciPy, and LAPACK. Collectively, these datasets provide valuable insights into the functional, performance, and security challenges faced in scientific computing.

## Other Application Domains

In addition to the above, software defect datasets have been created for various other application domains. For example, for database-related software, datasets are available for both transaction bugs in database systems [119] and access bugs in database applications [228]. Other application domains include computational notebooks [125] and internet of things (IoT) [243]. While these domains have only a small number of datasets, these datasets provide insights into the challenges caused by domain-specific bugs.

Table 2.4: Types of defects. One dataset may contain multiple types.

| Type        | Datasets                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | Count |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| Functional  | [66, 70, 72, 73, 75, 76, 79, 87, 90, 98, 106, 114, 118, 119, 122, 125, 129, 130, 135, 140–142, 146, 148, 157–159, 161, 172, 179, 186, 187, 190–192, 195, 196, 199, 204, 206, 206, 209, 219, 220, 226, 228, 230, 231, 239, 240, 242, 243, 255, 264, 272, 274, 277, 278, 291, 296, 297, 299, 303, 304, 317, 321, 328, 329, 340–342, 348, 350, 351, 354, 358, 360, 362, 365, 370, 371, 373, 374, 378, 379, 385, 391, 394, 396, 397, 405, 409, 410, 421, 427, 431–433, 439, 440] | 100*  |
| Security    | [62, 72, 91, 95, 97, 99, 106, 110, 120, 123, 128, 137, 148, 154, 159, 161, 176, 184, 201, 221, 243, 252, 258, 266, 267, 272, 282, 284, 289, 291, 293, 301, 306, 311, 344, 368, 370, 374, 383, 384, 395, 398, 409, 410, 427, 435, 443, 446]                                                                                                                                                                                                                                   | 48    |
| Performance | [81, 119, 129, 130, 142, 158, 159, 161, 172, 179, 196, 198, 228, 231, 243, 272, 291, 293, 305, 310, 317, 340, 360, 365, 371, 374, 409, 410, 414, 427, 432, 440, 441]                                                                                                                                                                                                                                                                                                         | 33    |
| Concurrency | [82, 145, 159, 182, 218, 223, 231, 317, 350, 364, 413]                                                                                                                                                                                                                                                                                                                                                                                                                       | 11    |
| Others      | [111, 229, 295]                                                                                                                                                                                                                                                                                                                                                                                                                                                              | 3     |

\*The ManyBugs and IntroClass Benchmarks [206] are counted as two distinct datasets. This convention applies to all other tables.

## 2.2.2 Types of Defects

This section categorizes datasets based on the types of defects they contain. Across the 151 software defect datasets we survey, we identify four commonly considered defect types: functional bugs, security vulnerabilities, performance bottlenecks, and concurrency issues. In addition, we include a fifth category, *others*, to account for datasets that do not clearly fall into any of the four primary categories.

Table 2.4 lists the datasets for each category, where a single dataset may appear under one or more categories. Functional defects are the most common type of defects in software defect datasets, with 100 (66.2%) datasets in this category. Security vulnerabilities are the second most common type, which are present in 48 (31.8%) datasets. Performance defects and concurrency issues are less common, with 33 (21.9%) and 11 (7.3%) datasets respectively. Figure 2.4 shows the breakdown of types of bugs for each of the application domains discussed in Section 2.2.1. We see that functional defects dominate most application domains and other types are also prevalent in specific domains based on the nature of the applications. The rest of this section further discusses these defect types and the datasets that target them. For each defect type, we discuss the characteristics of the defects and highlight representative datasets.

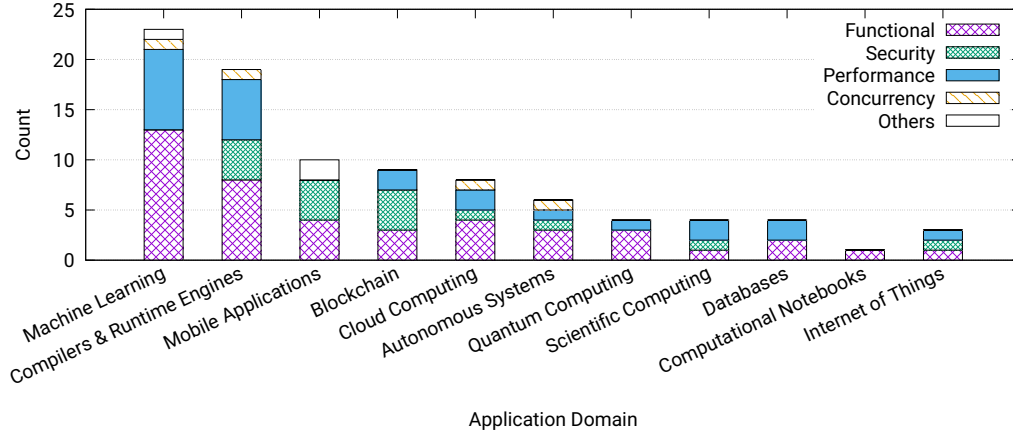


Figure 2.4: Breakdown of types of defects in each application domain.

## Functional Defects

Functional defects, or functional bugs, refer to errors in software that cause the software to diverge from the intended behavior and to violate its (usually informally provided) specifications. It is the most significant type of software defect, not only in the context of software defect datasets, but also in general software development [216, 335]. Functional defects impact users in various ways, ranging from minor inconveniences, over incorrect outputs, to application crashes. Figure 2.4 shows that functional defect datasets dominate most application domains, with the notable exceptions of blockchain and scientific computing.

Functional bugs are the most common focus in software defect datasets, as they form the backbone of software reliability studies. These datasets either provide a general collection of functional bugs across various categories (e.g., Defects4J [190], Bears [239], Bugs.jar [304], BugSwarm [351]) or target a specific subset of functional bugs to facilitate detailed analysis and testing. Examples of such specialized datasets include API misuses [364], regular expression defects [366], single-statement bugs, [192], type errors [431], string related defects [135], and numerical bugs [142, 362]. Overall, functional bug datasets cover a wide variety of topics, making them versatile resources for software defect research.

## Security Defects

Security defects, also known as vulnerabilities, are errors in software that make systems vulnerable to attacks or unauthorized access. These defects can compromise the integrity, confidentiality, and availability of a system, leading to severe consequences, such as data breaches, system crashes, or unauthorized access to sensitive information [205]. Security defects are particularly critical in software systems where data protection and privacy are paramount, such as in financial, healthcare, or governmental applications. While these defects can overlap with other types of bugs, their specific focus is on potential exploits that can be leveraged by malicious actors. As security is a major concern in software development, many datasets focus on identifying and analyzing security vulnerabilities to strengthen defenses. Figure 2.4 shows that datasets of security defects cover 7 out of 11 application domains, ranging from mobile applications to the internet of things. The interest in security defects even surpasses that in functional defects for blockchain-related software, which we attribute to the direct financial impact that vulnerabilities in this domain may have.

Different vulnerability datasets focus on different kinds of defects. One main type of security software defect dataset, e.g., ReposVul [368], Big-Vul[137], and CVEjoin [120], focuses on well-defined vulnerabilities, such as those cataloged in standardized databases like CVE (Common Vulnerabilities and Exposures) and categorized by CWE (Common Weakness Enumeration). Another kind of dataset focuses on buffer overflows [201], which are critical flaws where programs write memory locations beyond buffer boundaries, and compiler-introduced security bugs [395], which arise due to incorrect compiler optimizations. Lastly, cross-language vulnerabilities, which involve multiple programming languages, are captured in datasets like CrossVul [267], providing a basis for multi-language defense techniques.

## Performance Defects

Performance defects are software issues that cause performance degradation, leading to inefficient use of resources, increased latency, and longer execution times. These bugs are



critical in high-performance computing, web applications, and real-time systems where responsiveness and resource management are essential [188]. Performance bugs can arise from improper algorithms, inefficient memory usage, excessive computation, or mismanagement of concurrency. These defects can significantly impact user experience and operational costs, making their identification and resolution a priority in software engineering. As shown in Figure 2.4, datasets of performance issues are prevalent in 9 of 11 application domains, with the exceptions of mobile applications and computational notebooks.

Software defect datasets for performance issues have been constructed for both general applications and specific domains. For example, TANDEM [305] offers a taxonomy and dataset of real-world performance bugs, while ECench [198] introduces a benchmark of energy bugs specifically in Ethereum client software, highlighting issues where performance inefficiencies lead to excessive energy consumption. There are also datasets constructed from empirical research of performance issues, such as studies on performance degradation in browser environments [414], in JavaScript projects [310], in database systems [119] and applications [228], in high-performance computing (HPC) systems [81], in numerical libraries [142], and in cloud systems [159]. Such datasets could facilitate future research on pinpointing or mitigating performance issues.

## Concurrency Defects

Concurrency defects arise in software when multiple threads or processes execute simultaneously and interact in unintended ways, leading to nondeterministic behavior, crashes, or data corruption. These defects are particularly challenging because they often do not manifest consistently, making them difficult to detect, reproduce, and fix. Concurrency bugs include race conditions, deadlocks, and atomicity violations, which occur when proper synchronization mechanisms fail, or incorrect access to shared resources occurs. Given the complexity of multithreaded systems and the rise of concurrent programming models, identifying and studying concurrency bugs has become critical in ensuring the stability and reliability of

Table 2.5: Programming languages targeted in defect datasets. One dataset may cover multiple languages.

| Language   | Datasets                                                                                                                                                                                                                                                                               | Count |
|------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| Java       | [62, 70, 73, 79, 82, 97, 98, 111, 122, 123, 140, 141, 145, 146, 159, 186, 190, 192, 195, 198, 209, 220, 223, 226, 228, 229, 239, 240, 243, 252, 264, 277, 293, 296, 297, 304, 306, 311, 321, 328, 329, 342, 344, 348, 351, 354, 365, 368, 378, 384, 385, 391, 396, 397, 421, 441, 446] | 57    |
| C/C++      | [72, 73, 79, 81, 90, 110, 114, 119, 128, 130, 137, 142, 148, 157, 158, 172, 182, 184, 196, 199, 201, 206, 206, 218, 221, 226, 230, 231, 243, 278, 282, 284, 317, 340, 341, 344, 348, 350, 360, 368, 371, 373, 395, 398, 409, 414, 433, 435, 440, 441, 443, 446]                        | 52    |
| Python     | [66, 75, 79, 111, 125, 129, 130, 142, 158, 172, 179, 187, 191, 199, 219, 220, 226, 230, 242, 243, 255, 278, 293, 317, 340, 348, 350, 351, 362, 365, 368, 379, 385, 431, 432, 439–441, 446]                                                                                             | 39    |
| JavaScript | [91, 118, 135, 161, 243, 274, 299, 310, 364, 365, 371, 373, 446]                                                                                                                                                                                                                       | 13    |
| Go         | [119, 176, 198, 303, 396, 409, 413]                                                                                                                                                                                                                                                    | 7     |
| C#         | [73, 198, 278, 306, 446]                                                                                                                                                                                                                                                               | 5     |
| Rust       | [198, 299, 371, 410, 433]                                                                                                                                                                                                                                                              | 5     |
| Solidity   | [99, 106, 370, 427]                                                                                                                                                                                                                                                                    | 4     |
| Kotlin     | [87, 344, 391]                                                                                                                                                                                                                                                                         | 3     |
| PHP        | [73, 291]                                                                                                                                                                                                                                                                              | 2     |
| Ruby       | [129, 240]                                                                                                                                                                                                                                                                             | 2     |
| Others     | Fortran [142], F# [278], Groovy [87], Haskell [76], Julia [258], Q# [278], Scala [82]                                                                                                                                                                                                  | 7     |

modern software systems. As shown in Figure 2.4, concurrency defects exist in application domains where parallelism and real-time execution are critical, e.g., machine learning, compilers and runtime engines, cloud computing, and autonomous systems.

Several datasets have been developed to address different aspects of concurrency bugs. For C/C++, RaceBench [218] provides triggerable and observable race condition bugs to test race detectors, and RADBench [182] offers a concurrency bug benchmark suite focused on race conditions, deadlocks, and atomicity violations. For Java, Jbench [145] presents a suite of data race bugs specifically designed for concurrency testing, and JaConTeBe [223] focuses on concurrency bugs in real-world projects, offering manually written test cases for verification. Additionally, there are some datasets of concurrency bugs constructed from empirical studies, e.g., on actor-based concurrency systems [82], cloud computing systems [159], deep learning compilers [317], and Node.js [364], which offer invaluable insights for future studies.

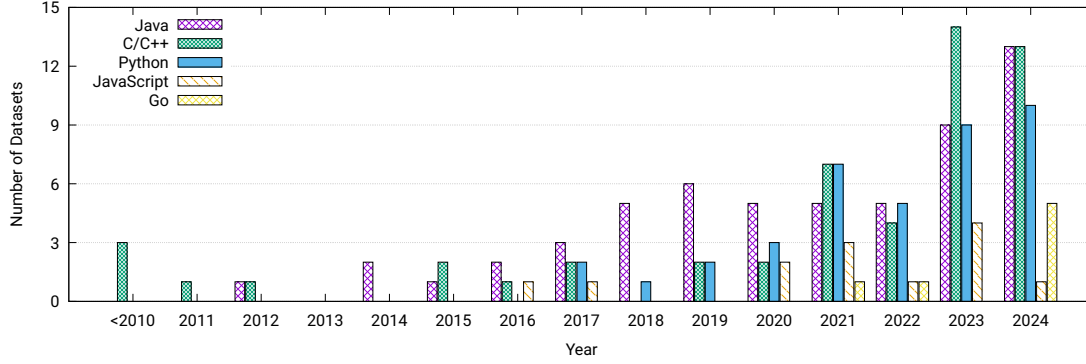


Figure 2.5: Trend of programming languages in software defect datasets.

## Other Types of Defects

Some datasets do not fall neatly into the four categories of functional, security, performance, or concurrency bugs. For example, such datasets consider critical bugs that prevent other bugs from being resolved [295], deployment faults in deep learning-based mobile applications [111], or resource leak bugs in Android applications [229]. These datasets address specialized defect types that fall outside the four main categories.

### 2.2.3 Programming Languages of Defective Software

Another way of categorizing datasets is by the programming language in which the defective code is written. 88.7% of our surveyed defect datasets focus on one or multiple programming languages. These languages include Java, Python, C/C++, JavaScript, Go, and others. Table 2.5 presents the distribution of programming languages across software defect datasets. Java stands out as the most represented language, with a total of 57 (37.7%) datasets, followed by C/C++ (52 datasets, 34.4%) and Python (39 datasets, 25.8%). The dominance of these languages is likely due to their popularity in software development and due to the focus of the research community on these languages. Other languages, such as F#, Groovy, Haskell, and Julia, have fewer datasets available.

The distribution of languages in defect datasets has evolved over time. Figure 2.5 shows the trend for the top five programming languages: Java, C/C++, Python, JavaScript, and

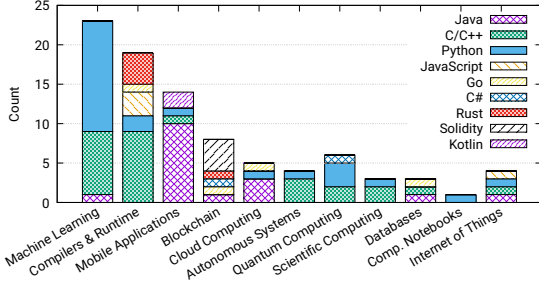


Figure 2.6: Programming languages across domains.

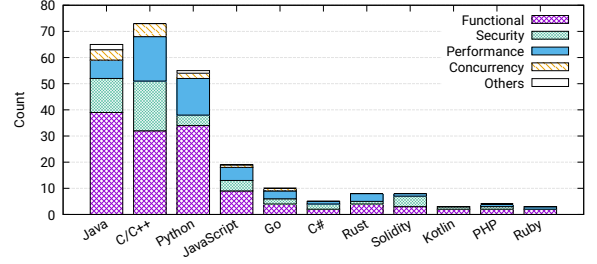


Figure 2.7: Defect types by programming languages.

Go. These five languages also represent the top five most used languages on GitHub [45], suggesting that dataset creation reflects real-world development trends. Java datasets have increased steadily over the past decade, while Python has gained significant popularity since 2017. JavaScript and Go have experienced peaks within the past five years, and C/C++ datasets have appeared consistently over time. It is important to note the surge in software defect datasets since 2023. While the general increase in publication volume contributes to this rise, another factor is the growing interest in domain- or type-specific defects. The surge is particularly pronounced for C/C++ datasets, with 14 papers in 2023 (a 250% increase from 2022), driven by growing interest in performance bugs [81, 441], security vulnerabilities [110, 395], and bugs in emerging domains such as machine learning frameworks [158, 172], compilers [371, 373], and quantum computing [440]. The trend continues in 2024 with 13 C/C++ datasets, suggesting sustained research momentum in infrastructure-level defect analysis. Another contributing factor is the rising demand for large-scale datasets to train and evaluate machine learning and large language models, which has spurred automated bug mining and benchmark construction studies in 2023 [72, 79, 110, 120, 186, 196, 242, 329, 405, 446]. This trend will be further discussed in Section 2.5.

Figure 2.6 shows the distribution of the most popular programming languages across different application domains. Python and C/C++ dominate machine learning-related defect datasets, and Java and Kotlin cover the major datasets for mobile applications. Compilers and runtime engines have a variety of programming languages, including C/C++,

Rust, Python, JavaScript, and Go. Most blockchain-related defects are from Solidity smart contracts, except for one dataset that collects defects from blockchain clients [198], which are implemented in Java, Go, C#, and Rust. In other domains, such as autonomous systems, quantum computing, scientific computing, and computational notebooks, C/C++ and Python are commonly used, whereas datasets for cloud computing, databases, and the internet of things involve Java, Go, C/C++, Python, and JavaScript.

Defect types are not equally distributed across programming languages. Figure 2.7 shows the distribution of software defect datasets across different programming languages and defect types. Java, C/C++, Python, JavaScript, and Go are common across all defect types, while C#, Rust, Solidity, and PHP cover functional, security, and performance defects only. Kotlin and Ruby have the least coverage, with Kotlin covering only functional and security defects, and Ruby covering only functional and performance defects.

**RQ1 Summary:**

- Over 60% of the datasets target general-purpose software, while the rest cover eleven specific domains including machine learning, compilers & runtime engines, mobile applications, and blockchain. Machine learning is the most popular specific domain with 14 datasets.
- Software defect datasets cover functional, security, performance, and concurrency defects. Functional bugs dominate (66.2%), followed by security vulnerabilities (31.8%), performance issues (21.9%), and concurrency defects (7.3%). Security and performance datasets are more common in domain-specific contexts such as compilers & runtime engines or blockchain.
- Java, C/C++, and Python appear in over 75% of all datasets, with JavaScript and Go showing increasing presence recently. Python and C/C++ predominate in machine learning and compilers & runtime engines, while Java and Kotlin dominate mobile applications. Language trends partially align with defect types: Java, C/C++, Python, JavaScript, and Go span all defect types, while other languages associate with specific types.

## 2.3 Dataset Construction

Creators of new defect datasets must take various decisions that will impact the quality, quantity, representativeness, and usability of the dataset. In particular, the process of con-

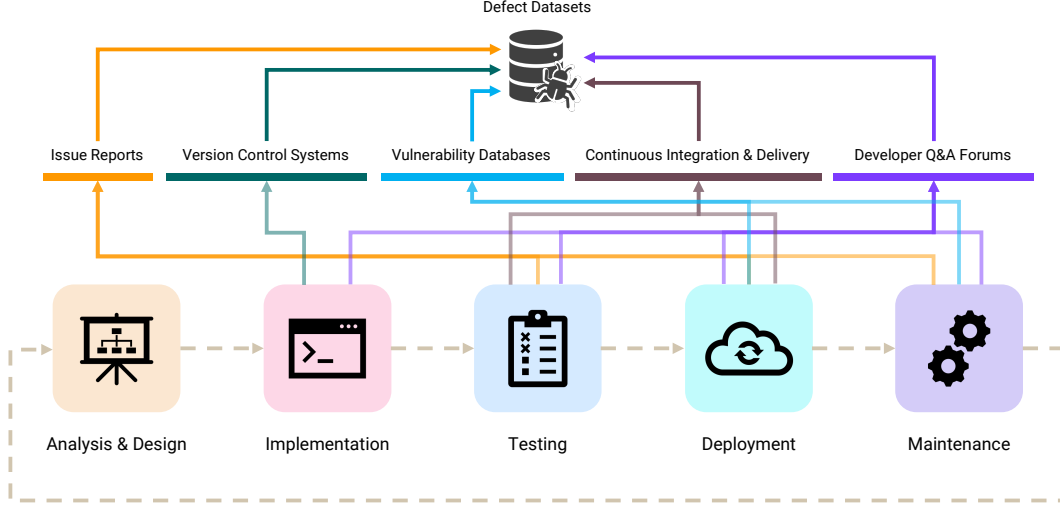


Figure 2.8: Methods for collecting defects (top) from stages in the software development processes (bottom).

structuring a defect dataset involves selecting an appropriate source and a suitable methodology. There are various sources of defects, such as bug reports from GitHub or Apache Jira, continuous integration systems like Travis-CI or GitHub Actions, and other pre-existing datasets. To extract defects from these sources, existing methodologies range from fully automated, rule-based approaches to semi-automated techniques that incorporate human validation or machine learning models, striving to ensure that the defects collected are representative of real-world scenarios. This section discusses the sources of defects (Section 2.3.1) and the construction methodologies (Section 2.3.2) employed by the datasets considered in our study.

### 2.3.1 Source of Defects

The *source* of a defect refers to the origin or platform from which the defect data is collected. As listed in Table 2.6, there are diverse sources for dataset creators to mine defects from, such as issue reports, version control systems, vulnerability databases, developer Q&A forums, and others. Most of these sources are closely tied to the software development process, as illustrated in Figure 2.8, especially in software implementation, testing, deployment, and

Table 2.6: Source of defects. One dataset may be based on multiple sources.

| Source                      | Datasets                                                                                                                                                                                                                                                                                                                        | Count |
|-----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| Issue Reports               | [73, 75, 76, 90, 91, 111, 119, 122, 123, 129, 130, 141, 142, 146, 148, 158, 159, 172, 179, 182, 186, 187, 190, 195, 199, 204, 206, 219, 223, 228, 230, 243, 272, 274, 277, 282, 289, 295, 297, 299, 304, 310, 317, 340, 342, 350, 358, 360, 362, 364, 371, 373, 374, 378, 383, 391, 395, 396, 409, 410, 414, 432, 433, 439–441] | 66    |
| Version Control Systems     | [66, 70, 72, 75, 76, 79, 81, 87, 90, 97, 110, 118, 125, 129, 135, 157, 161, 179, 186, 190–192, 198, 209, 229, 242, 255, 267, 278, 291, 293, 299, 328, 329, 344, 350, 354, 365, 368, 370, 379, 384, 394, 397, 409, 413, 431, 441, 443, 446]                                                                                      | 50    |
| Vulnerability Databases     | [114, 120, 137, 154, 176, 184, 201, 221, 266, 267, 284, 301, 368, 384, 405]                                                                                                                                                                                                                                                     | 15    |
| Other Pre-Existing Datasets | [70, 97, 140, 146, 191, 196, 274, 305, 405, 427]                                                                                                                                                                                                                                                                                | 10    |
| Developer Q&A Forums        | [82, 106, 111, 125, 179, 255, 362, 432, 433, 439]                                                                                                                                                                                                                                                                               | 10    |
| CI/CD                       | [239, 240, 258, 296, 303, 321, 351, 421]                                                                                                                                                                                                                                                                                        | 8     |
| Defect Synthesis            | [98, 99, 128, 218, 264, 348, 398, 435]                                                                                                                                                                                                                                                                                          | 8     |
| Coding Assignments          | [206, 220, 226, 306, 341, 348, 385]                                                                                                                                                                                                                                                                                             | 7     |

maintenance stages. Other sources, such as coding assignments and defect synthesis, offer a more controlled environment for collecting or creating defects. Among these sources, issue reports and version control systems are the most popular, accounting for 66 (43.7%) and 50 (33.1%) software defect datasets, respectively. Coding assignments, CI/CD, and defect synthesis are less frequently used, with no more than 8 (5.3%) datasets utilizing each. The rest of this section provides a detailed discussion of the primary sources of defects in software defect datasets, highlighting their unique characteristics.

## Issue Reports

A bug issue report is a documented record of a defect or problem encountered in a software system, typically submitted by developers or users through a bug tracking system, such as GitHub issues or Apache Jira. Issue reports are valuable sources of real-world software defects, providing insight into bugs as documented during the software lifecycle.

Several datasets collect defects directly from GitHub issues. These include gDefects4DL [219], HasBugs [76], Andror2 [378], and Bugs4Q [439]. In addition to GitHub, Apache Jira serves as another popular bug-tracking system, specifically for the large-scale software projects hosted by the Apache Software Foundation. Datasets like Defects4J [190], ApacheJIT [195], Bugs.jar [304], and Ohira et al. [272] source defects from Apache Jira, where developers

log comprehensive bug reports, including detailed reproduction steps, resolutions, and in many cases, links to the relevant commits that introduce and fix the bugs. Bugzilla is another prominent bug-tracking system that has been leveraged as a defect source by several software defect datasets, including the Eclipse and Mozilla Defect Tracking Dataset [204], BugHub [73], and CISB [395].

These datasets leverage the wealth of defect information available in issue reports, providing robust resources for research on bug prediction, fixing strategies, and software maintenance practices. However, relying exclusively on issue reports introduces several limitations. First, many issue reports lack explicit links to the commits or code regions that caused or fixed the defect, making it difficult to establish accurate traceability between natural-language descriptions and source-code changes [170, 451]. Second, issue reports are often incomplete or poorly described, missing key environmental or configuration information, which complicates defect reproduction and automated analysis [89, 460]. Consequently, datasets built from issue reports typically require extensive manual curation to ensure correctness and reproducibility.

## Version Control Systems

The commit histories of version control systems offer a detailed view of software evolution, providing a rich source for collecting software defects. By analyzing commit messages and code changes, researchers can identify bug-fixing commits and their corresponding buggy versions. Several datasets leverage Git commit histories as their primary source of defects. For instance, BugsPHP [291] mines commit pairs from GitHub to capture bug fixes in PHP projects, while Defectors [242] focuses on Python, extracting bug-inducing commits through a combination of rule-based and keyword-based approaches. FixJS [118] sources bug-fixing in JavaScript commits, grouping them into function-level changes, and Minecraft [79] offers a dataset with precise bug location information mined from Git commit histories. Similarly, D2A [443] conducts differential analysis on commit histories, using static analysis tools to



identify bug-fixing commits.

Besides GitHub commits, other datasets gather commits from GitLab, such as BugsC++ [72], or utilize the GitHub archive for large-scale multilingual dataset construction, like MBF [446]. These varied sources ensure a broad and diverse collection of bug data across different platforms and programming languages. Collecting defects from version control systems allows dataset creators to capture real-world bugs systematically, but this approach faces challenges. For example, identifying which commits actually introduce or fix defects is difficult, as approaches such as SZZ [200, 322] often misclassify refactorings or multi-commit fixes [263]. Additionally, commit messages vary widely in quality, making keyword-based mining unreliable [349] and often requiring additional context to fully understand the defect.

## **Vulnerability Databases & Other Datasets**

Another prominent source of software defects is pre-existing datasets, particularly those focused on security vulnerabilities, such as the Common Vulnerabilities and Exposures (CVE) dataset and the National Vulnerability Database (NVD). These datasets provide standardized, well-documented vulnerabilities, which are invaluable for creating benchmarks and conducting empirical studies. For example, CrossVul [267] leverages CVE data to build comprehensive datasets of Java and cross-language vulnerabilities, aimed at studying program repair and vulnerability detection. Similarly, CVEjoin [120] consolidates information from CVE and other threat intelligence sources to create a unified dataset for security research. VulinOSS [154] mines vulnerabilities from NVD and CVE to study security issues in open-source projects.

Besides the vulnerability databases mentioned above, some datasets rely on other well-established sources of defects. For example, both PreciseBugCollector [405] and BugOss [196] build defect datasets from OSS-Fuzz [313], a continuous fuzzing platform by Google. These datasets, built upon existing structured vulnerability repositories, provide a solid foundation for evaluating and improving security practices, as well as for developing more robust vul-

nerability detection and repair techniques. However, these datasets are limited to previously mined defects, preventing the inclusion of newly emerging ones. Furthermore, their quality inherently depends on the underlying databases, which can vary significantly across different sources.

## Developer Q&A Forums

Q&A platforms, e.g., Stack Exchange and Stack Overflow, provide valuable real-world insights into common bugs faced by developers. These platforms contain a wealth of developer-reported issues, which can be mined and categorized to capture bug symptoms, root causes, and their resolutions. For example, Morovati et al. [255] and Chen et al. [106] extract bugs from Q&A platforms to study defects in machine learning and blockchain systems, respectively. These datasets concentrate on practical, frequently encountered defects accompanied by detailed descriptions and resolutions, making them particularly valuable for empirical research. However, the unstructured nature of data on these platforms poses challenges related to data quality [290], readability [133], and human-factors [367], while LLM-based approaches show promise in automating the extraction and structuring of defects from such sources [121].

## Continuous Integration & Delivery

A continuous integration and continuous delivery (CI/CD) system records histories of the outcome (e.g., *failed* or *passed*) of automated builds and tests after each code change, offering a valuable source of real-world failures and their resolutions. CI/CD is closely tied to version control systems, as it usually monitors code repositories to trigger builds and tests on each commit or pull request. By utilizing fail-pass patterns, several software defect datasets have been built to capture defects that arise during regular development cycles, ensuring their relevance for both research and practice. For instance, BugSwarm [351] and Bears [239] mine CI pipelines from Travis-CI, leveraging fail-pass transitions in the test outcomes to

identify and record bug fixes. Similarly, with this model, BugSwarm [351, 456], GitBug-Java [321], and GitBug-Actions [303] also collect a wide range of defects across different software systems from GitHub Actions. These datasets emphasize reproducibility by providing detailed environments and scripts for setting up and replaying failures, making them valuable for execution-based benchmarking and debugging. Moreover, by utilizing the fail-pass pattern from CI/CD systems, these datasets capture real-world defects in an automated, reproducible manner, making them particularly valuable for studies that require large-scale, real-world, and reproducible defect datasets. Nevertheless, collecting defects from CI/CD pipelines inherently excludes defects that do not manifest within these automated processes, potentially overlooking issues that are caught before committing code into the repository or that manifest only in a production environment.

## Synthesis

Some datasets generate defects through automated code mutation or bug injection. While this methodology cannot guarantee that the resulting defects are representative of real-world bugs, it provides a controlled and repeatable process for creating benchmarks for research. Specifically, it allows creators to control the location and types of defects, which is not feasible with naturally occurring bugs. Ngo et al. [264] and RaceBench [218] introduce synthetic defects, such as variability and concurrency bugs, respectively, to evaluate bug localization and testing tools. FixReverter [435] focuses on bug injection by reverting CVE-related fixes to recreate real-world vulnerabilities for fuzz testing. Similarly, Bures et al. [98] and Cai et al. [99] inject faults into open-source projects and smart contracts, respectively, to assess security vulnerability detection. Additionally, with LLMs, DebugBench [348] injects faulty code into correct LeetCode solutions to create a defect dataset for evaluating the debugging ability of LLMs. These datasets provide a systematic way to study the effects of specific types of bugs and the efficacy of tools designed to detect or repair them.

## Coding Assignments

Another source of defects is submissions to coding assignments, programming competitions, and online judging platforms. While these defects may not reflect bugs in complex, real-world software, they often come with detailed descriptions and test cases, making them useful for evaluating bug detection and repair techniques. ConDefects [385] draws from coding platforms like AtCoder to collect bugs observed in a competitive programming environment. IntroClass [206] consists of bugs in student coding assignments and small C programs, providing benchmarks for automated program repair. Codeflaws [341] collects defects from competitive programming contests, while QuixBugs [220] is based on the Quixey programming competition, providing a multi-language benchmark for evaluating program repair techniques.

### 2.3.2 Construction Methodology

The *construction methodology* of a software defect dataset refers to the specific process by which defects are collected, and in some cases, validated and categorized. While different datasets can be constructed in radically different ways, some strategies appear repeatedly. Many datasets integrate several distinct strategies at different stages of their construction process, such as combining automated keyword-based search with subsequent manual validation of defects. This means that the construction methodologies outlined in this section are not rigid categories, but instead composable strategies that can be mix-and-matched.

In the following, we separate the construction process into three stages: defect collection, defect validation, and defect categorization. Defect collection involves gathering defects from the sources discussed in Section 2.3.1. Defect validation is the process of verifying that the collected defects genuinely represent software faults and defect categorization refers to assigning additional classification information to each defect within the dataset. Datasets with fully synthetic defects, as discussed in Section 2.3.1, are excluded from this section, since their construction does not involve defect mining methodologies. However, datasets

Table 2.7: Collection methodologies of defects. One dataset may use multiple methodologies.

| Methodology         | Datasets                                                                                                                                                                                                                                                                                                              | Count |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| Keyword Matching    | [75, 79, 81, 82, 87, 110, 111, 118, 119, 122, 123, 125, 129, 142, 146, 148, 157, 158, 161, 179, 184, 191, 192, 198, 199, 223, 228, 229, 242, 243, 255, 258, 278, 282, 291, 293, 297, 299, 305, 310, 317, 329, 340, 342, 350, 354, 362, 364, 365, 370, 374, 378, 391, 394–396, 409, 413, 414, 427, 431, 432, 441, 446] | 64    |
| Mapping             | [66, 91, 97, 110, 122, 141, 142, 154, 158, 172, 182, 184, 186, 190, 195, 199, 209, 219, 221, 223, 242, 266, 282, 284, 289, 295, 297, 301, 304, 329, 342, 358, 360, 364, 368, 371, 373, 383, 384, 391, 410, 446]                                                                                                       | 42    |
| Manual Collection   | [70, 76, 159, 179, 182, 201, 221, 228, 230, 272, 274, 289, 340, 350, 439, 441]                                                                                                                                                                                                                                        | 16    |
| External Oracle     | [196, 206, 226, 239, 296, 303, 321, 341, 348, 351, 385, 397, 421]                                                                                                                                                                                                                                                     | 13    |
| SZZ Algorithm [322] | [79, 141, 192, 195, 242, 282, 295]                                                                                                                                                                                                                                                                                    | 7     |
| Static Analysis     | [141, 192, 284, 311, 368, 443]                                                                                                                                                                                                                                                                                        | 6     |
| Machine Learning    | [368, 383, 443]                                                                                                                                                                                                                                                                                                       | 3     |
| Others              | [62, 90, 95, 99, 106, 120, 130, 135, 145, 187, 204, 206, 240, 328, 379, 433, 440]                                                                                                                                                                                                                                     | 17    |

that obtain code from external sources and subsequently mutate it, such as SBP [99], remain within the scope of this discussion.

## Stage I: Defect Collection

Defect collection refers to the process of gathering defects from one or more mining sources. The particular strategies that dataset creators employ depend on their choice of defect sources and on the intended scope of the dataset. For instance, if a dataset is sourced from CI/CD systems, then collecting defects based on historical test failures is a natural choice. Table 2.7 lists several common defect collection strategies, showing the diversity of approaches used to construct software defect datasets. Keyword matching and mapping are the most common strategies, with 64 and 42 datasets using them, respectively. Other strategies, such as manual collection, external oracle use, and specialized algorithms [322], are less common, but still used by a number of datasets.

**Keyword-Based Collection** Many datasets attempt to find defects by scanning for specific keywords (e.g., “bug” or “fix”) in commit messages, issue reports, and other relevant text fields. Zhong et al. [446] make for a prototypical example: they examine all GitHub

push and pull request events in their target repositories, and consider a bug-fixing event to be any commit or pull request that contains one of the terms “fix,” “repair,” “patch,” “bug,” “issue,” “problem,” or “fault” in the commit message or pull request description. Those keywords are all typical for this kind of approach, built off the assumption that a commit that mentions bugs, repairs, and so on is likely attempting to fix a defect. Keyword-based defect collection has limited precision [349], but it is still one of the most common collection approaches observed in this survey, presumably due to its simplicity and its abundance (64 datasets, 42.4%) in previous literature.

**Mapping** Another common approach in defect collection is to map defects from one source to objects from another source, such as mapping commits from a project’s Git history to related bug reports documented in its issue tracker. This strategy is necessary for datasets that (1) source defects from bug reports or vulnerability databases (Section 2.3.1) and (2) wish to include source-code-level information (Section 2.4.2). Accordingly, mapping is common, with 42 datasets (27.8%) utilizing it. The difficulty of this process can vary based on the mapping source and destination. For instance, mapping GitHub issues to GitHub commits is quite easy, since GitHub tracks commits that reference issues and exposes that relationship in its website and API. In contrast, mapping CVE entries to fixing commits can be more challenging, since many CVE datasets do not reference the commits that fix a vulnerability, and likewise many vulnerability-fixing commits do not reference their associated CVEs. Jiang et al. [184] outline their process for mapping CVEs to bug-fixing commits in detail, using six different strategies to construct these relationships.

**Manual Collection** Some dataset creators forego automatic defect collection, but instead manually collect defects. For instance, Jalbert et al. [182] create RADBench by searching through the issue trackers of several projects to find concurrency bugs. An advantage of this approach is that it allows researchers to make qualitative decisions about what defects to include in their datasets, such as only including “vital” issues, which can depend on a variety

of factors that are not easy to judge automatically. Conversely, a notable disadvantage is the substantial increase in manual effort imposed on dataset creators. Consequently, it is unsurprising that manually collected datasets typically exhibit a smaller median size compared to automatically collected datasets (298 versus 1158 entries, respectively). Also, while qualitative judgments about a defect can be useful, they are also inherently subjective, which may deter potential dataset users who prioritize objectivity. These disadvantages are probably why this strategy is, with only 16 datasets, relatively uncommon among the surveyed datasets.

**External Oracle** Some datasets use historical test failures or continuous integration (CI) failures as external oracles to distinguish buggy from non-buggy software. This should not be confused with datasets that *reproduce* test failures, which are described in Section 2.3.2. For example, BugSwarm [351, 456] is created by scanning through the commit history of repositories that use GitHub Actions or Travis-CI, looking for cases where one commit’s CI run fails and a subsequent commit’s run passes. Using historical failures to distinguish bugs from non-bugs allows for better confidence in a dataset’s accuracy, given that failing tests are a tell-tale sign of a bug. However, since this strategy relies on historical test or CI results, it inherently limits the bugs that can appear in a dataset to those that have this historical information available. This naturally limits the defect source of a dataset, which can be an issue if the dataset creators are concerned about dataset size. Moreover, in the case of CI failures, care must be taken to ensure that a given failure was actually caused by a failing test (as opposed to, say, a code style checker).

**SZZ-Based Collection** The SZZ algorithm, introduced by Sliwerski et al. [322] and refined multiple times since [200, 380], attempts to identify the software change that introduced a bug given the change that fixes the bug. This addresses a weakness in some other collection strategies (e.g., keyword search), which is that while they can readily find software changes that *fix* bugs, they have no way to identify the change(s) that *caused* those bugs. There

are seven datasets in our survey using SZZ or a similar algorithm to mine bug-introducing commits. Mahbub et al. [242] demonstrate a typical approach: after finding bug-fixing pull requests in their target repositories, they use SZZ to associate one or more bug-introducing commits to each bug fix.

**Static Analysis** There are six datasets in the survey that use static analysis to collect defects. To this end, the dataset creators run existing or newly created static analyses on candidate code. As an example of the former, Zheng et al. [443] create D2A by running the static analyzer Infer on pairs of commits, looking for detected issues that are present in one commit but absent in the next. As an example of the latter, ManySStuBs4J [192] uses a custom static analysis approach that compares the ASTs of candidate buggy and fixed programs and searches for a set of specific bug-fix patterns. An advantage of using static analysis is that, depending on the specific tools or techniques used, one can achieve better precision than strategies such as keyword matching (i.e., fewer non-bugs classified as bugs). The main disadvantage is that relying on static analysis to separate bugs from non-bugs will limit a dataset’s scope to the abilities of the static analysis. Many static analyzers only detect specific classes of bugs (e.g., null pointer bugs), and even more generalist analyzers like Cppcheck will necessarily fail to catch some bugs [162].

**Machine Learning-Based Collection** There are three datasets in the survey that use machine learning techniques as part of their data collection processes. Wang et al. [368] input candidate patches into LLMs, asking the models to evaluate whether those patches fix software vulnerabilities. Their dataset also illustrates that extra care must be taken to ensure that LLM output is sound, which they achieve by using LLMs in conjunction with static analysis tools and by including only defects where the two agree. Similarly, Zheng et al. [443] and Wu et al. [383] use machine learning models to distinguish between defects and non-defects based on semantic information, such as bug reports or commit messages.



Table 2.8: Validation methodologies of defects. One dataset may use multiple methodologies.

| Methodology         |                | Datasets                                                                                                                                                                                                                                                                                  | Count |
|---------------------|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| Manual Inspection   | <i>Full</i>    | [76, 81, 82, 87, 90, 111, 119, 123, 125, 130, 135, 142, 146, 158, 159, 161, 172, 179, 190, 196, 198, 201, 209, 219, 223, 228–230, 239, 255, 258, 278, 282, 289, 293, 297, 304, 317, 321, 340, 342, 348, 350, 362, 364, 365, 374, 378, 379, 391, 394–396, 409, 413, 427, 431–433, 439–441] | 62    |
|                     | <i>Partial</i> | [66, 79, 122, 129, 141, 157, 199, 243, 274, 299, 328, 354, 368, 370, 373, 383, 414, 443, 446]                                                                                                                                                                                             | 19    |
| Defect Reproduction |                | [75, 76, 87, 91, 95, 97, 145, 161, 182, 186, 187, 190, 196, 206, 209, 239, 291, 296, 297, 303, 304, 310, 321, 328, 340, 348, 350, 351, 378, 379, 384, 391, 395, 397, 409, 413, 431, 432, 439]                                                                                             | 39    |

## Stage II: Validation

Dataset validation is about ensuring that the defects included in a dataset are actual defects. This is an important step for making a useful dataset, since a high false positive rate (i.e., many non-bugs classified as bugs) can taint further research that uses a dataset. Even so, not all dataset creators take this step, and there are legitimate reasons for doing so. For instance, a dataset might be so large that validation becomes impractical (e.g., [73, 110, 118, 120, 242]), or a dataset’s data collection approach might already have a low false positive rate (e.g., [99, 106, 128, 226, 329, 341]). Table 2.8 shows common validation methodologies and datasets using them.

**Manual Inspection** Some dataset creators, such as Lee et al. [209], Wang et al. [365], and Silva et al. [321], opt to manually inspect the code and description of every bug in their datasets (i.e., full manual inspection). This allows researchers to construct a dataset with a low false positive rate and to make qualitative judgments about what bugs should be included, without spending the time and resources needed for manual defect collection. Full manual inspection is quite popular among the datasets surveyed, with 62 (41.1%) datasets employing it. The largest [348] and smallest [196] datasets that employ this strategy have 4,253 and 18 entries respectively, with a median of 260 entries.

Conversely, some dataset creators choose to only manually inspect a subset of the defects in their dataset (i.e., partial manual inspection). This is less common than full inspection, with only 19 (12.6%) datasets employing it. However, it is attractive for particularly large

datasets where full inspection is infeasible, such as D2A [443], the largest dataset that uses this approach, which has over 1.3 million entries. The smallest such dataset [414] still contains 191 entries, with the median size being 5,565 entries.

Typically, partial inspection has no effect on the actual construction of the dataset, but is performed to demonstrate the accuracy of an automated methodology [443]. However, there is at least one case where partial inspection is used to filter out defects from a dataset. Zhong et al. [446] automatically classify each bug in their dataset into one of multiple fault categories. They then manually analyze 50 bugs from each category to ascertain that category’s accuracy, and only include bugs from categories with over 90% accuracy in the final dataset.

**Defect Reproduction** Some dataset creators aim to ensure the accuracy of their datasets by reproducing each defect they collect. Reproduction typically involves running the buggy program in a controlled environment and observing the faulty behavior, often through failing test cases or other verifiable symptoms. While the exact meaning of *reproducibility* can vary depending on the goals and assumptions of different datasets [455], it generally requires the ability to build and run the faulty version of the software.

Among our surveyed datasets, 39 (25.8%) involve defect reproduction, and only one dataset attempts to reproduce only a subset of its defects, while the rest aim for full reproduction. This process is often facilitated by sources that provide a rich environment and execution context, such as continuous integration (CI) systems. For example, BugSwarm [351] leverages CI configurations and historical build logs to automatically reproduce bugs in Docker-based environments, thus making the automation of reproduction feasible.

However, automation has limitations. Many defects are not covered by existing tests, or their reproduction may require complex setup steps beyond test execution (e.g., user interface-related bugs in AndroR2 [378]). Still, automated reproduction enables the construction of much larger datasets than manual validation strategies. The largest dataset in

Table 2.9: Categorization methodologies of defects. One dataset may use multiple methodologies.

| Methodology           | Datasets                                                                                                                                                                                                                                                                                                                      | Count |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| Manual Categorization | [70, 76, 81, 82, 90, 106, 111, 119, 123, 125, 129, 130, 135, 142, 145, 146, 148, 158, 159, 172, 179, 182, 184, 191, 196, 198, 201, 206, 206, 219, 223, 228–230, 243, 255, 258, 272, 278, 282, 293, 299, 305, 310, 317, 329, 340, 342, 350, 360, 362, 364, 365, 371, 374, 391, 396, 409, 410, 413, 414, 427, 431–433, 439–441] | 68    |
| Static Analysis       | [141, 191, 219, 306, 311, 341, 370, 443]                                                                                                                                                                                                                                                                                      | 8     |
| Machine Learning      | [79, 329, 394]                                                                                                                                                                                                                                                                                                                | 3     |
| Others                | [97, 110, 154, 195, 239, 284, 296, 301, 351, 358, 421]                                                                                                                                                                                                                                                                        | 11    |

this category [95] contains approximately 168,000 entries. The median size, 225 entries, is comparable to that of datasets which use full manual inspection.

### Stage III: Categorization

To enhance the utility of their datasets, many dataset creators opt to classify the defects in their datasets along different axes, such as the root causes and symptoms of defects. Defect categorization does not involve adding or removing entries from a dataset, but simply adding metadata to defects already in a dataset. Table 2.9 shows the categorization strategies for the datasets that use one, which include manual categorization, static analysis, and machine learning.

**Manual Defect Categorization** By far the most common way for dataset creators to categorize the defects in their datasets is through manual examination, with 68 (45%) datasets falling into this category. Similar to manual validation, manual categorization allows researchers to operate reasonably efficiently and flexibly. Indeed, it is not uncommon for datasets to employ both manual validation and manual categorization, (e.g., ECench [198] and ExcePy [431]), with 48 (31.8%) datasets in this group.

Defect categorization need not be entirely automatic or entirely manual. For instance, Ohira et al. [272] label “surprise” and “dormant” bugs automatically, since the definitions they use are easy to detect, while manually labeling “blocking,” “security,” “performance,” and “breakage” bugs.

**Categorization via Static Analysis** Eight datasets in the survey use static analysis to classify defects. For instance, Senanayake et al. [311] run two static analyzers, Qark and MobSF, on the APKs and source files of Android applications and store the analysis results in their dataset. Categorizing defects based on static analysis has some of the same benefits and drawbacks of static analysis-based collection, namely a fairly high precision, but a limited scope of possible categories and the possibility of false negatives (i.e., bugs with a missing classification).

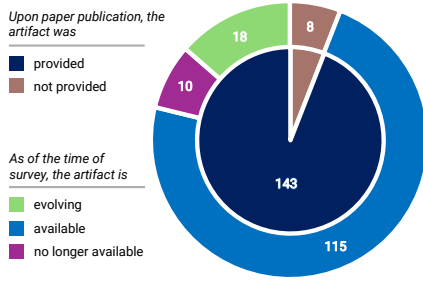
**Categorization via Machine Learning** In total, three of the surveyed datasets classify their entries using machine learning approaches. Song et al. [329] train a neural network to classify bugs given a bug report, fixing commit message, bug-fix location, and bug-inducing location. Avula et al. [79] use an off-the-shelf LLM to generate a description of what each bug fix in their dataset does. Xu et al. [394] use latent Dirichlet allocation [94] (a kind of Bayesian network) to extract topic keywords from pull requests and add them to each defect’s metadata.

**RQ2 Summary:**

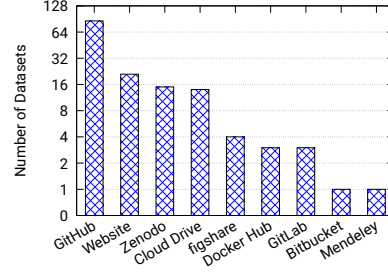
- The majority of software defect datasets are sourced from software development processes, with issue reports and version control systems being most common. A smaller number rely on pre-existing datasets, vulnerability databases, developer Q&A forums, coding assignments, and CI/CD systems. Some datasets employ defect synthesis techniques to generate synthetic defects for controlled experimentation.
- Dataset construction generally involves three stages: defect collection, validation, and categorization. Collection is most often performed through automated mining strategies such as keyword matching (42.4%) or mapping (27.8%) between issue reports and commits. Fewer datasets rely on manual collection, external oracles, the SZZ algorithm, static analysis, or machine learning techniques. Validation and categorization practices vary widely, though manual inspection remains most common in both stages.

## 2.4 Availability and Usability

The availability and usability of a dataset play a crucial role for its impact on other research and practical applications. Most of the dataset papers in this study provide accompanying



(a) Availability of artifacts.



(b) Hosting platforms. One dataset may be hosted on multiple platforms.

Figure 2.9: Availability and hosting platforms of software defect datasets.

artifacts that range from simple download links for datasets to more complex, executable frameworks designed to automate bug reproduction or testing. There are a few instances where the actual datasets are not provided at all, or where the artifacts are not easily accessible or usable. In this section, we examine and discuss the availability and usability of the 151 datasets we survey.

### 2.4.1 Availability and Hosting Platforms

Ideally, all datasets should be publicly available to facilitate future studies. As shown in Figure 2.9a and Table 2.10, we found that all but eight datasets were made publicly available at the time of paper publication. In total, 143 (95.3%) dataset papers have released their datasets alongside the corresponding paper. For the 143 publicly available datasets, we further examined their availability at the time of our survey and categorized them into the following three groups:

(1) *Available* means the dataset is still available at the time of our survey, as it was when the corresponding paper was published, but has not been updated. This group accounts for 76.2% of the datasets we survey.

(2) *Evolving* means the dataset is publicly available, and it has been updated or evolved since the time of paper publication. Such evolution could include adding new defects, deprecating stale defects, or updating the framework or tool to facilitate the use of the dataset. This

Table 2.10: Availability of artifacts of surveyed datasets.

| Availability |                            | Datasets                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | Count |
|--------------|----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| Provided     | <i>Available</i>           | [62, 66, 73, 75, 79, 81, 82, 87, 90, 95, 98, 99, 106, 110, 111, 118–120, 122, 123, 125, 128, 129, 135, 137, 140–142, 145, 148, 154, 157, 158, 161, 172, 176, 179, 184, 187, 191, 192, 195, 196, 198, 199, 204, 206, 206, 218–221, 223, 226, 228, 229, 240, 242, 243, 255, 264, 266, 267, 274, 277, 278, 282, 284, 289, 291, 296, 297, 299, 301, 303–306, 310, 311, 317, 340–342, 348, 350, 354, 358, 360, 362, 365, 370, 371, 373, 374, 378, 383, 384, 391, 394–397, 405, 409, 427, 431–433, 435, 439–441, 443, 446] | 115   |
|              | <i>Evolving</i>            | [70, 72, 76, 91, 97, 186, 190, 209, 239, 252, 293, 321, 328, 351, 368, 379, 385, 413]                                                                                                                                                                                                                                                                                                                                                                                                                                | 18    |
|              | <i>No Longer Available</i> | [114, 130, 146, 159, 201, 258, 272, 329, 364, 421]                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | 10    |
| Not Provided |                            | [182, 230, 231, 295, 344, 398, 410, 414]                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | 8     |

group accounts for 11.9% of the surveyed datasets.

(3) *No Longer Available* means the dataset was available at some point in time, but is not available anymore as of the time of our survey. Further investigation shows the reasons include (a) institutional webpage used to host the dataset becoming no longer loadable [114, 201, 272, 364], (b) datasets being removed from a hosting service [130, 329], (c) unresolvable shortened links [258], and (d) datasets being lost by accident [159]. This group accounts for 6.6% of the surveyed datasets.

Hosting platforms play a significant role in ensuring the ease of access to datasets, as our analysis of the datasets that are no longer available shows that most of them are offline due to discontinued or unstable hosting services. Figure 2.9b shows the distribution of hosting platforms for the available datasets we survey. Note that some datasets are hosted on multiple platforms. The most popular hosting platform is GitHub, which hosts 86 (60.1%) available datasets. The second most popular hosting approach is to self-host a dedicated website for the dataset. Other hosting platforms, such as Zenodo, cloud storage services (e.g., Google Drive, OneDrive, Box), are also frequently used, while other platforms such as figshare, Docker Hub, GitLab, Bitbucket, and Mendeley are less common.

Table 2.11: Presentation of datasets.

| Presentation    |                  | Datasets                                                                                                                                                                                                                                                                                                                    | Count |
|-----------------|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| Meta Level      | <i>Metadata</i>  | [73, 79, 81, 82, 106, 111, 119, 120, 122, 123, 125, 129, 135, 137, 141, 142, 148, 157, 158, 172, 176, 179, 184, 191, 195, 199, 204, 219, 223, 228, 229, 240, 242, 243, 267, 274, 277, 278, 282, 289, 293, 299, 305, 306, 311, 317, 340, 354, 358, 360, 362, 365, 370, 371, 373, 374, 383, 391, 394–396, 409, 433, 440, 441] | 65    |
| Code Level      | <i>Snippet</i>   | [66, 99, 110, 140, 221, 368, 443, 446]                                                                                                                                                                                                                                                                                      | 8     |
|                 | <i>Diff</i>      | [192, 198, 405, 435]                                                                                                                                                                                                                                                                                                        | 4     |
|                 | <i>File</i>      | [62, 98, 118, 145, 206, 264, 284, 310, 341, 427, 439]                                                                                                                                                                                                                                                                       | 11    |
| Execution Level | <i>Project</i>   | [206, 252, 255, 432]                                                                                                                                                                                                                                                                                                        | 4     |
|                 | <i>Binary</i>    | [95, 342, 378]                                                                                                                                                                                                                                                                                                              | 3     |
|                 | <i>Framework</i> | [70, 72, 75, 76, 87, 90, 91, 97, 128, 154, 161, 186, 187, 190, 196, 209, 218, 220, 226, 239, 266, 291, 296, 297, 301, 303, 304, 321, 328, 348, 350, 351, 379, 384, 385, 397, 413, 431]                                                                                                                                      | 38    |
|                 |                  |                                                                                                                                                                                                                                                                                                                             |       |

## 2.4.2 Presentation

There are many ways to present the defects in a dataset, ranging from simply providing the link to the corresponding issue reports to creating complex executable frameworks. The way datasets present defects can impact their usability for both research and practical applications. We categorize the presentation of defects into three levels: (1) meta level, (2) code level, and (3) execution level. Table 2.11 shows the presentation levels of 133 surveyed datasets that are available or evolving.

(1) *Meta Level*. Meta-level datasets provide the details about defects only in textual form. For example, a table listing all defects with links to corresponding issue reports is considered meta-level. Such metadata may also include the date of creation for an issue report, a severity label, or other annotations. Meta-level datasets are helpful as a reference or resource for analysis but do not include direct means for code analysis or defect reproduction. From the datasets that are still available at the time of this survey, 65 (48.9%) are meta-level.

(2) *Code Level*. Code-level datasets provide isolated code snippets or source files from the defective program, as well as code diffs that show the changes made to the program to fix the defects. For example, a dataset that provides both buggy and fixed versions of source files without providing the entire project is considered code-level. Code-level datasets are helpful for source code analysis, but do not include direct means to run or reproduce the defects. It takes extra effort to integrate (e.g., via `git apply`) the provided code into a

complete project to reproduce the defects. From the datasets that are still available at the time of this survey, 23 (17.3%) are code-level.

(3) *Execution Level.* Execution-level datasets offer more hands-on resources by providing executable artifacts to help reproduce the bugs. For example, a dataset that provides a framework to automatically check out the defective program and reproduce the building and testing process is considered execution-level. Other forms of execution-level datasets include those that provide complete projects with build configurations or datasets that provide executable binaries. Execution-level datasets are particularly useful for practical applications, such as the evaluation of fault localization, test generation, or automated program repair techniques. From the datasets that are still available at the time of this survey, 45 (33.8%) are execution-level.

#### **RQ3 Summary:**

- Nearly all datasets (94.7%) were released publicly at publication. Among them, 80.4% remain available, 12.6% have evolved with updates, and 7% are no longer available due to discontinued hosting or link decay. Long-term persistence remains a concern.
- GitHub dominates hosting, with 64.7% of available datasets. Dedicated websites, Zenodo, and cloud storage are also common. Datasets that disappeared were typically hosted on institutional or self-managed servers, indicating that unstable hosting contributes to data loss.
- Dataset usability varies by presentation level. About 48.9% provide only meta-level information (issue links, descriptions), 17.3% include code-level artifacts (buggy files, diffs), and 33.8% offer execution-level resources (complete projects, reproduction frameworks). Execution-level datasets enable direct evaluation but require additional maintenance.

## **2.5 Uses of Software Defect Datasets**

Understanding how software defect datasets are used in practice is essential not only for assessing their impact on research and practice, but also for revealing current needs. To this end, we conduct a systematic analysis of research work citing the top-five most-cited<sup>3</sup> software defect dataset papers among the ones considered in this survey, which are Defects4J [190],

---

<sup>3</sup>As of January 1, 2025, on Google Scholar.



Bug Prediction Dataset [122], Big-Vul [137], the ManyBugs and IntroClass benchmarks [206], and SWE-Bench [187]. All five datasets are general-purpose and not limited to any specific application domain. Four datasets primarily focus on functional defects, while Big-Vul specifically targets security vulnerabilities. In terms of programming languages, Defects4J and the Bug Prediction Dataset collect defects from Java projects; Big-Vul and the ManyBugs and IntroClass benchmarks focus on C/C++ defects; and SWE-Bench, the most recent among them, is in Python.

**Methodology** We categorize the uses of software defect datasets by conducting a systematic analysis of papers citing these datasets. Initially, we manually review a subset of citing papers to develop a preliminary categorization. Using an open-coding approach, we continuously refine the categories as we review additional papers. This taxonomy construction process involves over 150 randomly selected citing papers, leading to the following categorization scheme:

- **Empirical Research:** Research on existing software defects and existing software engineering techniques, which can be further divided into empirical analyses and empirical evaluations.
  - Empirical Analyses: Papers that analyze existing software defects, such as defect characteristics, root causes, and fix patterns.
  - Empirical Evaluations: Papers that perform retrospective evaluations solely on existing software engineering techniques using software defect datasets.
- **Technical Evaluations:** Evaluations of existing or new software engineering techniques using software defect datasets, which can be further divided into detection, testing, and repair.
  - Detection: Papers that use software defect datasets to evaluate bug detection and localization techniques.

- Testing: Papers that use software defect datasets to evaluate testing techniques, such as test generation and test prioritization.
- Repair: Papers that use software defect datasets to evaluate automated program repair techniques, such as code repair and issue solving.
- **New Benchmark**: Papers that introduce new benchmarks.
- **Others**: Papers that cite software defect datasets only as related work but without directly using them.

The “empirical research” and “technical evaluations” categories are not mutually exclusive, as a paper may use a dataset for both purposes. For instance, a paper that empirically evaluates *existing* program repair tools using established software defect datasets is classified into both empirical research and the repair category of technical evaluations, while a paper that introduces a *new* repair technique and evaluates it falls solely under technical evaluation.

Based on the above scheme, we use the `o3-mini` model from OpenAI to automatically categorize papers using their titles and abstracts. The model is instructed with a structured prompt<sup>4</sup> that enumerates our taxonomy and labeling rules, including that a paper may have multiple labels, that *new benchmark* is mutually exclusive with other categories, and that ambiguous cases should be marked as *others*. Two authors jointly validate a random sample of 100 LLM-labeled papers, independent of the open-coding process, since open coding did not record per-paper labels. All 100 sampled papers have satisfactory classifications: 93 exactly match the human labels, and only 7 show minor discrepancies (e.g., non-contradictory extra categories). For instance, paper [386] is labeled as both “Empirical Study” and “Repair” by the model, whereas the human label includes only “Repair”, because the abstract highlights substantial empirical evaluation. Likewise, paper [127] is labeled as both “Empirical Study” and “Detection” for the same reason, while the human label includes only “Detection”. We

---

<sup>4</sup>The full prompt is provided in the supplementary materials [457].

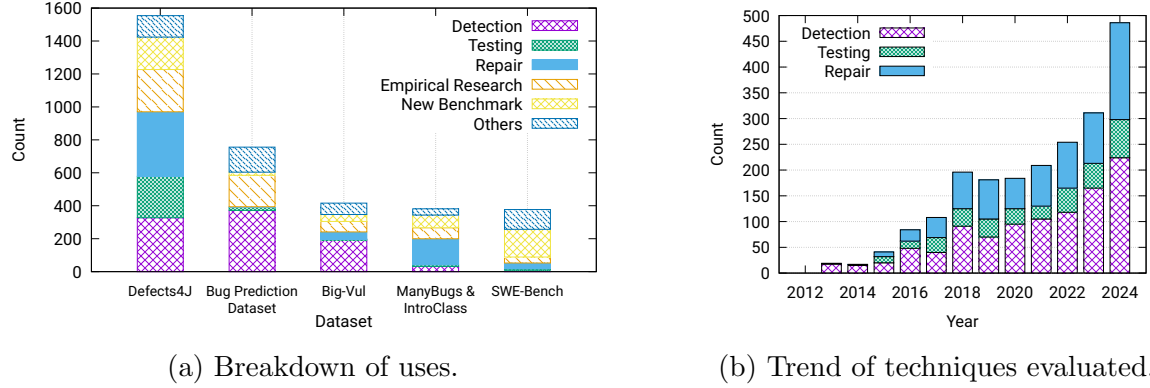


Figure 2.10: Uses of software defect datasets.

have also inspected the model-provided reasoning summaries, which generally (i) propose candidate categories based on cues in the title and abstract, (ii) apply the taxonomy rules to include or exclude labels, and (iii) resolve overlaps by prioritizing the primary contribution over supporting components.

Figure 2.10a illustrates that empirical research and technical evaluations are the primary use categories for the citing papers analyzed across the five most-cited datasets. Technical evaluations, which can be further divided into evaluations of detection, testing, and repair techniques, demonstrate a growing demand for software defect datasets, as shown in Figure 2.10b. This section focuses on these two primary categories to provide a comprehensive overview of the uses of the most-cited software defect datasets. As our analysis specifically focuses on direct dataset utilization, we exclude papers that introduce new benchmarks and the “others” category from the detailed discussion.

### 2.5.1 Empirical Research

Software defect datasets can serve as valuable resources for empirical research to understand various aspects of software defects. Such empirical research can be broadly categorized into two groups. On one hand, studies may focus directly on analyzing software defects themselves, typically with regard to their manifestations, symptoms, root causes, and fixes. For example, Sobreira et al. [325] present an in-depth analysis of 395 patches from Defects4J. By

analyzing the patch size, location, and the underlying repair patterns, the study provides insights and future direction for automated program repair, such as highlighting the necessity for repair strategies capable of effectively handling method calls. On the other hand, empirical research can also explore interactions between software defects and existing software engineering techniques. For example, Durieux et al. [134] conduct an empirical evaluation on eleven automated program repair tools using five software defect datasets. Their study shows that the effectiveness of automated program repair tools varies significantly across different datasets and concludes various causes of failures for the automated program repair. Both types of empirical research significantly benefit from pre-collected, well-structured software defect datasets, as these datasets offer diverse, realistic, and accessible instances of software defects, thus enabling rigorous analysis and reliable comparisons across different studies.

### 2.5.2 Technical Evaluations

Another primary use of the most-cited software defect datasets is to evaluate existing or new software engineering techniques, such as fault detection/localization (e.g., [71, 197, 268, 285, 307–309, 338, 363, 377, 399, 400, 419]), test generation/prioritization (e.g., [69, 93, 102, 112, 152, 181, 300, 436]), and automated program repair (e.g., [103, 138, 225, 346, 406]). Fault detection/localization aims to identify defects in software in an effective, efficient, and automated manner. Test generation/prioritization is about generating or prioritizing test cases to maximize the likelihood of detecting defects, while minimizing the cost of testing. Both detection and testing techniques have long attracted significant interest from the software engineering, programming languages, and other communities, as such techniques are essential for improving software quality and reliability. The goal of automated program repair [207] is to fix defective software without human intervention, which is a challenging task that has gained increasing attention in recent years. The evaluation of these techniques is crucial for understanding their effectiveness, efficiency, and limitations, and software defect datasets play a critical role in this process. By providing standardized, realistic, and reproducible

instances of software defects, software defect datasets enable researchers and practitioners to evaluate their techniques in a controlled environment.

Figure 2.10a illustrates that detection and repair techniques are the most frequently evaluated. Defects4J is used for evaluating all three types of techniques, and it is also the pre-dominant dataset used for evaluating testing techniques. The Bug Prediction Dataset [122] and Big-Vul [137] are primarily used for evaluating detection techniques, while both the ManyBugs and IntroClass benchmarks [206] and SWE-Bench [187] are used more extensively for evaluating repair techniques. Figure 2.10b illustrates the unprecedented growth in the number of proposed software engineering techniques, particularly in the area of automated program repair. This surge in recent years underscores the increasing importance and demand for high-quality software defect datasets.

**RQ4 Summary:**

- Software defect datasets are primarily utilized in empirical studies and technical evaluations. Empirical work investigates defect characteristics, root causes, and fixes, while technical evaluations emphasize testing, detection, and automated repair.
- The most-cited datasets are frequently reused in software engineering research. Their adoption indicates a shift toward evaluating automated techniques like automated repair, test generation, and defect detection, highlighting the increasing demand for executable and large-scale datasets suitable for data-driven and LLM-based methods.

## 2.6 Discussion and Opportunities for Future Work

In this section, we discuss the challenges and opportunities in creating and maintaining software defect datasets, and provide future directions for enhancing the utility of such datasets for the research community. We focus on dataset creation (Section 2.6.1) and maintenance (Section 2.6.2).

### 2.6.1 Creation of Software Defect Datasets

**Scale, Diversity, Quality, and Cost** Though various methodologies for constructing defect datasets have been proposed (Section 2.3), the balance between scale, diversity, quality,

and cost remains a challenge. Intuitively, datasets of high quality tend to be smaller and more costly to create, such as the hand-crafted Defects4J [190]. In this sense, further efforts are still needed for fully automated defect mining. However, an important bottleneck for automated mining is the lack of a ground truth that indicates the actual defects present in the codebase. For example, keyword-based mining, the most commonly used technique, shows a relatively low precision of 51.9% and recall of 58.8% [349]. It would be desirable to establish a ground truth without manual labelling or testing. BugSwarm [351] and Bears [239] have shown that by leveraging the fail-pass pattern in CI/CD pipelines, one can automatically mine defects with high precision. Such techniques may be adopted into the mining processes for other sources, without narrowing to CI/CD build histories. Similarly, ReposVul [368] have shown that the combination of LLMs and static analysis can achieve up to 95% precision in defect mining. Such approaches can be further explored to improve the quality of defect datasets while keeping the manual effort low.

**Scope Selection** Software defect datasets vary in scope, ranging from general-purpose datasets to domain- or type-specific datasets. Datasets with specialized scopes are desirable for more effective benchmarking of targeted techniques, or for fine-tuning LLMs for specific purposes [144]. However, as highlighted in Section 2.2, there is a lack of datasets focusing on specific domains, non-functional defects, and some fast-growing programming languages. For example, despite the widespread adoption of multi-threading capabilities in Python, there is currently no available software defect dataset dedicated to concurrency defects in this language. Future research efforts could prioritize creating datasets that cover more specialized application domains, defect types, or programming languages, to better support the development of targeted software engineering techniques.

**Precise Defect Isolation and Annotation** Though tremendous efforts have been made in defect mining, the granularity of defect datasets is usually at the repository-, file-, or method-level, without fully precise defect location information that pinpoints defects, e.g.,

at the line-level. Ideally, datasets should isolate defects, e.g., by removing any unrelated code changes from a bug-fixing commit. For example, to isolate actual defects from newly added features, refactorings, and other code changes, Defects4J [190] has manually reviewed and filtered the code diffs that fix a given bug. Such a manual review and filtering process is costly and time-consuming, though; thus, it is preferable to have automated techniques to precisely locate defects. Such isolation techniques could be based on refactoring detection [185] or probabilistic delta debugging [329]. Another related problem is the annotation of defects, e.g., with the defect’s category. Though some datasets provide such annotations, they are typically manually labeled or inferred from other metadata such as exceptions [351]. Recently, machine learning and natural language processing techniques have shown promising results for automated defect annotation [64, 329, 394]. More investigation is needed on the effectiveness and efficiency of such techniques, and on how they can be integrated into the process of creating defect datasets.

**Dataset Organization** Effectively organizing a dataset is crucial for maximizing its availability and usability. Currently, there is no standardized format for organizing software defect datasets, which can lead to difficulties in dataset access and usage. GitHub is the most popular platform for hosting datasets and tool chains, but it does not ensure long-term dataset availability and is susceptible to data loss or deletion. It is desirable to have datasets archived in a more stable and reliable repository, such as Zenodo or figshare, in addition to GitHub [1]. Furthermore, the organization of datasets should facilitate future evaluation of new techniques. Section 2.4.2 shows that half of the datasets provide only textual metadata (i.e., meta-level), which is insufficient for running software defects. Our analysis indicates that, after normalizing citation counts by dataset age (i.e., citations per year), execution-level datasets exhibit higher uptake (mean 53.65, median 12.00) than code-level (mean 21.01, median 12.75) and meta-level datasets (mean 18.07, median 8.50). Future efforts should focus on providing more comprehensive and reproducible software artifacts for defects. At a

minimum, we recommend the following artifact reproducibility checklist: (1) a metadata index that records each defect and links it to the corresponding buggy and fixed code versions; (2) an environment descriptor, such as a container image or environment specification; (3) clear build and test instructions with expected outcomes; and (4) a runnable reproduction driver that demonstrates the defect and confirms the fix.

## 2.6.2 Maintenance of Software Defect Datasets

**Reproducibility** Ensuring the reproducibility of defects in software defect datasets is crucial for researchers to validate their findings and advance existing work. However, achieving reproducibility poses a significant challenge in software defect datasets, regardless of the construction methodology or dataset organization [455]. Recent work shows that the major reason for defects not being reproducible is inaccessible dependencies, which datasets can mitigate by caching dependencies [303, 321, 455]. In this direction, more techniques are required to improve the long-term reproducibility of datasets while maintaining the balance between dataset size and usability.

**Dataset Evolution** Software defect datasets can benefit from further evolution after initial publication, but most are not updated after release, leading to outdated information. Datasets that do not evolve may cause data leakage when evaluating machine learning-based approaches, especially LLMs, as defects may be in the training corpus [347]. Future datasets should support continuous updates by adding new defects while maintaining historical data for longitudinal studies. Continuous mining with automated pipelines [351, 456] and community-driven efforts [246] show promise for dataset evolution. However, frequent updates complicate comparisons with prior work, as different studies may use distinct dataset versions. Clear versioning, documentation, and stable baselines are crucial for fair and reproducible comparisons. Future research should explore best practices for balancing new defects with historical data maintenance and incentivizing dataset creators to evolve pub-



lished datasets.

## 2.7 Threats to Validity

**Selection and implementation bias** Despite our comprehensive search across four major bibliographic databases, we may miss work published outside these databases, or in venues not covered by our selection criteria. Also, excluding preprints and non-peer-reviewed works ensures quality but may miss emerging datasets. Automated scripts in data collection may contain unnoticed errors, and human judgment in screening may introduce subjectivity.

**Annotation and interpretation bias** Manual annotation of dataset attributes involves subjective interpretation. Although we measure inter-rater agreement across multiple independent annotations, minor biases may still persist. The LLM-based analysis in Section 2.5 is manually validated, but some errors may remain due to model limitations.

**Construct validity** Our analytical dimensions may omit factors such as community engagement, and differing classification granularity within each dimension may affect cross-study comparability. Our usage analysis prioritizes the most-cited datasets and may not generalize; dataset evolution after publication may also affect the stability of findings.

## Chapter Summary

The landscape of software defect datasets is broad and uneven. Across the 151 datasets we surveyed, defect coverage is concentrated on functional defects in general-purpose software, and specific categories remain underrepresented. Construction methodologies vary widely, and that choice strongly affects both reproducibility and bug realism. Availability is similarly uneven: many datasets are public, but broken links, custom formats, and decayed infrastructure are common obstacles. These findings motivate Chapter 3, which addresses

the maintenance and growth side of dataset infrastructure.

This chapter is based on the paper “From Bugs to Benchmarks: A Comprehensive Survey of Software Defect Datasets,” co-authored with Robert M. Furth, Michael Pradel, and Cindy Rubio-González, and published in ACM Computing Surveys in 2026. The work was supported by NSF grants CNS-2016735, CNS-2346396, and CCF-2119348; by European Research Council (ERC) grant agreements 851895 and 101155832; and by the German Research Foundation through the ConcSys, DeMoCo, and QPTest projects. The catalog is published as an interactive website at <https://defect-datasets.github.io/>.

# Chapter 3

## Reproducing the Software Engineering Pipeline

[Chapter 2](#) mapped the landscape of defect datasets and surfaced two failure modes in the infrastructure that supports software engineering research: broad but uneven coverage and silent decay of the artifacts over time. This chapter addresses the decay problem and the growth problem in turn. The first half presents the first systematic study of reproducibility across five widely used Java defect datasets, followed by a 13-month longitudinal case study of BUGSWARM that identifies the root causes of breakages and develops mechanisms that restore reproducibility to handcrafted dataset quality. The second half presents ACTIONSREMAKER, a tool that reproduces GitHub Actions builds offline so the field can keep mining new defects as production CI pipelines evolve. The two halves operate on the same artifact infrastructure but pull in opposite directions: looking backward to identify what breaks, then forward to supply replacements.

### 3.1 On the Reproducibility of Software Defect Datasets

Reproducibility in software engineering means that a result from an experiment can be repeatedly observed by others following the same procedure and using the same software

artifact [55]. Reproducibility is imperative for research [83]. The ability for software artifacts to build and execute as they did originally with the same end result is crucial. The inability to repeat or extend the results of an experiment with a software artifact would defeat its purpose. In particular, software defect datasets [114, 185, 190, 206, 231, 239, 304, 351] have a large impact on the software engineering research community. These datasets enable the study and design of new techniques for fields such as automated program repair [153, 376, 389, 390], test prioritization [233, 236, 244, 249, 271, 318], test generation [149–151, 174, 315, 388], and fault localization [162, 283, 326, 425]. These fields have helped make software more robust, efficient, and secure.

Software defect datasets contain hundreds of defects collected from real-world projects. These datasets are comprised with a buggy version of the source code and the corresponding fixed version. Additionally, the process of building and testing is automated, and does not require domain-specific knowledge. An important aspect of any such dataset is to provide a reliable way to successfully build its artifacts so that researchers can replicate experiments or conduct new studies. Without this, their adoption and usefulness would be limited.

A troubling fact is that software defect artifacts are not immune to breakages after their creation [334]. Breakage in this context means that the original defects and fixes are no longer reproducible. Software breakage in these artifacts is no different, and perhaps more prone. Such artifacts are historical versions of the software, which are more likely to suffer from missing dependencies or external resources that affect their build outcome. Additionally, software defect artifacts can become broken due to previously passing tests that start failing due to deprecated APIs, or unavailable resources required during the testing process. The inability to evaluate or study these artifacts hinders research progress by prohibiting comparisons in future evaluations. Surprisingly, to the best of our knowledge, there is no literature on the reproducibility of software defect artifacts and their proneness to breakages.

In this section, we present a study on the reproducibility of software defect datasets. First,

we conduct a study on reproducibility of five state-of-the-art Java software defect datasets: DEFECTS4J [190], GROWINGBUGS [185], BUGS.JAR [304], BUGSWARM [351], and BEARS [239]. We aim to answer two research questions: (1) What criteria and strategies do datasets apply to determine reproducibility and to reduce the risk of software breakage, and (2) How are datasets affected by breakages. We find that reproducibility has different meanings across datasets. For example, while some datasets consider the *existence* of any test failure sufficient to consider a bug reproducible, others require matching the *number* and *names* of failing tests. As for reproducibility strategies, different approaches are applied to ensure reproducibility, from reproducing multiple times upon artifact creation to managing project dependencies and using containerization. Furthermore, we find that software breakages affect all datasets, especially those that adopt automated approaches for their creation whose reproducibility ranges from only 26.6% to 69%.

Second, we conduct a case study on the reproducibility of the BUGSWARM dataset. BUGSWARM is designed to continuously grow, and adopts containerization to ensure reproducibility, i.e., each artifact is packed as a Docker image. This facilitates the study and the fix of breakages. The goal is to determine (1) How software defect artifacts experience breakages through time, (2) What are the root causes and corresponding patches for breakages, and (3) How to prevent future breakages to ensure long-term reproducibility. We systematically monitor the reproducibility of 1,795 Java artifacts over a 13-month period. We find that 1,124 (62.6%) artifacts break at least once, and 275 (15.3%) break multiple times. With a test interval of 11.7 days, on average 32 artifacts are newly broken during each test. To identify root causes of breakages, we manually inspect 1,606 breakages. We identify 11 root causes and handcraft patches for 10 of them. Those patches are applied to 1,055 (93.9%) distinct broken artifacts in an automated manner in 2,948 fixes. Among all the fixes, 44% of them are related to broken project dependencies retrieved by the build system. To prevent breakages in the future, we propose dependency caching, which caches project dependencies in the Docker container of each artifact. Dependency caching is successful on 1,700 (94.7%)

artifacts with an average size increase of only 5.1%. Finally, we isolated 1,257 out of 1,700 artifacts, which are reproducible even without internet. An evaluation of the reproducibility of isolated artifacts over an 8-month period revealed that reproducibility reaches 95% or higher.

There has been previous research in fixing broken builds [166, 238, 247, 256, 343, 356]. These approaches can be applied automatically to fix broken artifacts due to dependency-related issues. They range from including additional repositories from which dependencies are retrieved, to specifically changing dependency versions. However, these repair strategies do not consider environmental factors (e.g., changing the JDK or TLS versions), which we find to be critical in the long-term reproducibility of software artifacts. Furthermore, the above neither study the nature of software breakage through time nor apply prevention strategies to ensure long-term reproducibility.

Our study on reproducibility has revealed open opportunities for the research community that are applicable to defect datasets, and software in general. Software fragility is a real problem. This is compounded by the unavailability of software dependencies and bad practices in writing non-isolated tests. We highlight this by quantifying their effects on the reproducibility of BUGSWARM. We have developed multiple approaches for repairing and preventing breakage that we hope can be learned from and adopted by other datasets, as well as software engineering in general.

The contributions of this section are as follows:

- We present a study on the reproducibility of five state-of-the-art Java software defect datasets, and show how applied reproducibility strategies fail to prevent software breakages (Section 3.2).
- We examine the reproducibility of BUGSWARM in a 13-month period to investigate the frequency, root causes, and fixes of software breakages (Section 3.3).
- We develop dependency caching and artifact isolation to reduce the risk of breakage and

to ensure long-term reproducibility. We successfully isolate 1,257 artifacts, which have a reproducibility of handcrafted quality while being mined automatically (Section 3.3).

## 3.2 Reproducibility of Java Defect Datasets

Software defect datasets lay an essential foundation for software engineering (SE) research. In the past decade, along with the focus of fault localization, test generation and automated program repair on Java projects, several Java defect datasets [169, 185, 190, 220, 239, 304, 328, 351] have been created by the SE community, acting as benchmarks to evaluate proposed techniques on real-world software while helping advance the state of the art. In particular, areas such as fault localization and automated program repair require *reproducible* software defects. Reproducibility in this context means that the code can be built, tests can be run, and expected failures are observed. Sudden breakage of defect artifacts, i.e., broken builds, the absence of expected failures or the occurrence of unexpected failures, can be quite problematic for the SE community as it would lead to wrong findings, nonreplicable past studies, or even to the inability to perform future studies.

This section investigates the reproducibility of five state-of-the-art Java software defects datasets: DEFECTS4J, GROWINGBUGS, BUGS.JAR, BUGSWARM and BEARS. Each artifact consists of a buggy version of the source code and the corresponding fixed version. All datasets, except for BUGS.JAR, provide scripts to build and run existing tests. Note that we only focus on datasets whose artifacts consist of whole projects. Datasets that only provide code snippets or single source files without build configurations, such as QUIXBUGS [220], are not included in this study. Also, we do not include datasets of non-functional bugs [163, 293] and web application bugs [330]. Lastly, recent datasets [169, 328] not available at the time of this study are not included. The rest of this section aims to answer the following research questions:

**RQ1** What criteria and strategies do datasets use to define and ensure reproducibility?

Table 3.1: Datasets for reproducibility study. *Release* indicates the exact version picked for the study.

| Dataset     | # Projects | # Artifacts | Artifact Source | Build System         | Bug Location    | Provided Reference       | Release |
|-------------|------------|-------------|-----------------|----------------------|-----------------|--------------------------|---------|
| DEFECTS4J   | 17         | 864         | Issue Tracker   | Ant / Maven          | Source          | List of Failing Tests    | 397075d |
| GROWINGBUGS | 150        | 570         | Issue Tracker   | Ant                  | Source          | List of Failing Tests    | 6071840 |
| BUGS.JAR    | 8          | 1,158       | Issue Tracker   | Maven                | Source          | Build Log for Buggy      | 8410717 |
| BUGSWARM    | 120        | 1,795       | Travis-CI       | Ant / Gradle / Maven | Source / Config | CI Log for Buggy & Fixed | 181f304 |
| BEARS       | 72         | 251         | Travis-CI       | Maven                | Source          | List of Failing Tests    | 912bb98 |

**RQ2** How are datasets affected by software breakages?

### 3.2.1 Reproducibility Criteria & Strategies

Table 3.1 captures the details of the selected datasets including the number of defect artifacts included in each dataset as of writing of this section, the number of different projects these artifacts belong to, the source of the defects (issue tracker or continuous integration) and their location (source files and/or configuration files), and the kind of reference provided by the datasets (list of failing tests, build logs at time of creation, or CI historical build logs). The rest of this section focuses on investigating the reproducibility criteria and strategies employed by Java datasets to ensure reproducibility, both at creation time and long term. For this, we have examined three sources of information when available: dataset documentation, publications, and dataset infrastructure code. Below we describe our findings for each dataset.

#### Defects4J

DEFECTS4J [40, 190] is one of the most widely-used software defect datasets, which consists of 864 artifacts manually mined over a span of 8 years. A list of reference failing tests is provided by DEFECTS4J for each artifact. These are the failing tests captured upon the *creation* of the artifact, which failed on the buggy version but passed on the fixed version. DEFECTS4J considers an artifact to be reproducible if all of its reference failing tests are observed when reproducing (running again) the buggy version, and no failing tests are observed when reproducing the fixed version. To ensure long-term reproducibility, DEFECTS4J lists



the system requirements common to all artifacts, including required versions of Java, Git, SVN and Perl. Furthermore, specific dependency packages of projects are downloaded from the official web server of DEFECTS4J in the initialization stage. To check the reproducibility of DEFECTS4J, users can run a verification script provided by the maintainers.

## GrowingBugs

GROWINGBUGS [33, 34, 185] extends DEFECTS4J’s framework to *automatically* mine 570 artifacts via BUGBUILDER. Like DEFECTS4J, GROWINGBUGS provides a list of failing tests for each artifact as reference. GROWINGBUGS adopts DEFECTS4J’s reproducibility criteria and relies on *some* of its reproducibility strategies. While the initialization step from DEFECTS4J is automatically run to download dependency packages from DEFECTS4J’s web server, GROWINGBUGS does *not* update this server with new dependency packages used by newly added artifacts, which can result in missing packages that may need to be downloaded from third-party locations. Finally, its compatibility with DEFECTS4J allows the reuse of DEFECTS4J’s verification script to examine dataset reproducibility.

## Bugs.jar

BUGS.JAR [35, 304] provides 1,158 artifacts automatically mined from bug reports of 8 large open-source projects. For each of its artifacts, BUGS.JAR provides a reference log for the buggy version, which was generated at the time the dataset was created. Also at creation time, BUGS.JAR removes tests that fail when running the fixed version based on the assumption that such tests may be unrelated to the fix. Thus, BUGS.JAR considers an artifact reproducible if there are no observed failing tests when running the curated fixed version, and if *any* failing test is observed when running the buggy version. To ensure reproducibility at creation time, each artifact is run 10 times to detect non-reproducible and flaky artifacts, which are excluded from the dataset. No additional actions are taken for long-term reproducibility. To the best of our knowledge, BUGS.JAR does not provide scripts

to run artifacts and verify their reproducibility.

## BugSwarm

BUGSWARM [36, 351] is a software defect dataset that leverages Continuous Integration (CI) to automatically mine 1,941 artifacts from projects hosted in GitHub. Unlike BUGS.JAR, BUGSWARM provides reference logs for buggy and fixed versions. The reference logs are historical build logs generated by Travis-CI when the code was originally committed to GitHub and tested in the cloud. In determining reproducibility, BUGSWARM checks whether the *number* of passing & failing tests and the *names* of observed failing tests match those listed in the reference logs as well as the status of the CI build (**passing**, **failing** or **error**). No test failures are expected for the fixed version to be considered reproducible. To ensure reproducibility during creation, BUGSWARM tested the reproducibility of each artifact 5 times. The dataset includes artifacts that are reproducible more than once but less than five times, which are labeled as “flaky.” We exclude all 146 flaky artifacts and consider the remaining 1,795 artifacts in this study. BUGSWARM containerizes each artifact as a Docker image in an attempt to ensure long-term reproducibility. The Docker image aims to replicate the exact environment used by Travis-CI to run the code and includes tailored scripts to run buggy and fixed versions.

## Bears

Like BUGSWARM, BEARS [32, 239] automatically mines 251 artifacts from 72 GitHub projects that use Travis-CI. However, instead of reference logs, BEARS provides for each artifact a JSON file with information collected at creation time, such as reference failing tests. Similar to BUGS.JAR, BEARS considers an artifact reproducible if there are no failing tests in the fixed version, and there is at least *one* failing test in the buggy version. During creation, each artifact is run twice; artifacts with flaky tests are excluded from the dataset. BEARS relies on Maven for the build process. No other actions are taken for long-term

Table 3.2: Criteria of reproducibility for datasets.

| Dataset     | Definition of Reproducibility |        |      |        |
|-------------|-------------------------------|--------|------|--------|
|             | Existence                     | Number | Name | Status |
| DEFECTS4J   | ✓                             | ✓      | ✓    |        |
| GROWINGBUGS | ✓                             | ✓      | ✓    |        |
| BUGS.JAR    | ✓                             |        |      |        |
| BUGSWARM    | ✓                             | ✓      | ✓    | ✓      |
| BEARS       | ✓                             |        |      |        |

reproducibility. A script is available to run all artifacts, but reproducibility is not verified.

**RQ1 Summary:** Reproducibility criteria vary across datasets ranging from the existence of *any* failure to matching the number and/or names of failing tests, and CI status. Various reproducibility strategies are applied at creation time, the most popular of which runs artifacts multiple times to identify flakiness. Strategies for long-term reproducibility include managing dependencies and using containers.

### 3.2.2 Software Breakages in Java Defect Datasets

In this section, we discuss how state-of-the-art datasets are affected by software breakages given the various reproducibility criteria and strategies discussed in the previous section.

#### Criteria for Reproducibility

As discussed earlier, various reproducibility criteria are used across datasets. Here we categorize these criteria as **Existence**, **Number Match**, **Name Match** and **Status Match**. **Existence** means that the artifact is reproducible if there is at least *one failing test* when running the buggy version. **Number Match** means that the artifact is reproducible if the *number* of observed failing tests (and passing tests, if required by the dataset) matches the number of tests reported by a given dataset in their reference data. **Name Match** requires the *names* of failing tests to match. In addition of fulfilling Number Match and Name Match, **Status Match** also requires *CI build status* to match. Note that all of these criteria are based on the assumption that the fixed version should never have failing tests.

Table 3.2 shows the *default* reproducibility categories per dataset. BUGS.JAR and BEARS only check Existence of failing tests. DEFECTS4J and GROWINGBUGS use Name Match,

and therefore also require Number Match and Existence. BUGSWARM adopts the most strict criterion Status Match that requires everything including CI build status to match.

## Experimental Setup

When running each artifact, we first check out the buggy version, build the project, run tests and gather results as *reproduced* logs. Then we checkout the fixed version and repeat the process of building, testing and collecting reproduced logs. Because BUGS.JAR only provides a `developer-patch.diff` instead of a fixed version, we apply the patch to the buggy version via `git apply` to generate the fixed version.

We comply with the environmental requirements of each dataset when specified. DEFECTS4J and GROWINGBUGS explicitly require the use of Java 8, and BUGSWARM includes the necessary Java versions in the Docker images of its artifacts. On the other hand, BUGS.JAR and BEARS do not specify a required Java version for their artifacts. We use Java 8 because (1) Java 8 is the earliest version that is still in life cycle at the time of conducting this study (Java 7 and earlier versions are deprecated and thus inaccessible), and (2) this study aims to report reproducibility from a user’s standpoint, and therefore it is not intended to debug individual artifacts. Note that there are artifacts from BUGS.JAR and BEARS that were originally built with Java 7 or older versions. We discuss the implications of unspecified Java versions in Section 3.4.

We use the scripts provided by DEFECTS4J, GROWINGBUGS, BUGSWARM and BEARS to run their artifacts. For BUGS.JAR, which does not provide scripts, we leverage the reference logs to infer the commands to build and run tests. Then we verify the correctness of the commands by comparing the test classes executed with those listed in BUGS.JAR’s reference logs. For all datasets, we parse the reference logs and the reproduced logs to extract test information, and in the case of BUGSWARM, Travis-CI build outcome. Then we calculate reproducibility, i.e., the percentage of artifacts in each dataset match their reference logs, and thus are still reproducible.

Table 3.3: Reproducibility of software defect datasets. Results for default criteria are shown in bold.

| Dataset     | # Artifacts | Reproducibility for Different Criteria |               |                    |                    |
|-------------|-------------|----------------------------------------|---------------|--------------------|--------------------|
|             |             | Existence                              | Number Match  | Name Match         | Status Match       |
| DEFECTS4J   | 864         | 837 (96.9%)                            | 837 (96.9%)   | <b>837 (96.9%)</b> | N/A                |
| GROWINGBUGS | 570         | 175 (30.7%)                            | 170 (29.8%)   | <b>170 (29.8%)</b> | N/A                |
| BUGS.JAR    | 1,158       | <b>308 (26.6%)</b>                     | 303 (26.2%)   | 303 (26.2%)        | N/A                |
| BUGSWARM    | 1,795       | 1,392 (77.5%)                          | 1,388 (77.3%) | 1,387 (77.3%)      | <b>1,239 (69%)</b> |
| BEARS       | 251         | <b>137 (54.6%)</b>                     | 134 (53.4%)   | 134 (53.4%)        | N/A                |

All datasets except for BUGSWARM explicitly exclude flaky artifacts. For BUGSWARM, we do not consider artifacts marked as “flaky.” Nevertheless, we run the artifacts of each dataset three times to rule out unknown flakiness. Note that we do not observe major differences across runs, and this section presents the results for the *first* run dated in July 2022.

## Results

Table 3.3 shows the reproducibility of each software defect dataset. Interestingly, the reproducibility under the category of Existence seems quite accurate to be representative. Matching number and/or name of tests just slightly decreases the reproducibility of all datasets. In fact, only 18 artifacts are reproducible under Existence but broken under Number Match or Name Match. We manually inspected all of them and found that 10 artifacts have missing expected failing tests when reproducing and 8 artifacts have unexpected failing tests. These artifacts still have observed failing tests but the number or name does not match the reference provided by datasets. On the other hand the category Status Match, only used by BUGSWARM, has a more significant impact on the overall reproducibility. While results for each criterion can be found in Table 3.3, the rest of this discussion focuses on the *default criterion* used by each dataset.

DEFECTS4J achieves the highest reproducibility of 96.9%. Among 864 artifacts, only 27 are broken. Note that maintainers of DEFECTS4J have deprecated 29 artifacts due to

behavioral changes introduced under Java 8. But we still examine reproducibility of all artifacts and only 27 of the 29 deprecated artifacts are broken in our run. Recall that DEFECTS4J is a handcrafted dataset that has taken more than 8 years to grow to 864 artifacts. The tedious and time-consuming human labor, among other things, has paid off in achieving a remarkable level of reproducibility.

As an extension of DEFECTS4J, GROWINGBUGS automatically mined 570 new artifacts without human intervention. However, the reproducibility of these auto-mined artifacts is considerably lower than that of the handcrafted DEFECTS4J artifacts, despite being newer. We find that 400 out of 570 artifacts are currently broken 15 months after their creation when using their default criterion Name Match, resulting in a reproducibility rate of only 29.8%. Using the least restrictive criterion Existence increases reproducibility slightly to 30.7%.

For BUGS.JAR, only 308 out of 1,158 artifacts are found to be reproducible, i.e., 26.6%. Despite running each artifact 10 times to ensure reproducibility at creation time, reproducibility of 850 artifacts faded away since the dataset creation in October 2017. Interestingly, despite that BUGS.JAR uses the most loose criterion Existence by default, 303 out of 308 reproducible artifacts remain reproducible when applying stricter reproducibility criteria. If we apply Number Match and Name Match, the reproducibility will be both 26.2%.

We find that 1,239 out of 1,795 artifacts are reproducible in BUGSWARM. The containerization strategy seems to contribute to a slightly higher reproducibility of 69% after 32 months since its creation, which is the highest among those datasets created automatically. Recall that BUGSWARM’s default reproducibility criterion is the most restrictive: Status Match. Less restrictive criteria Number/Name Match and Existence lead to a higher reproducibility of 77.3% and 77.5%, respectively. The difference in reproducibility is due to mismatches of the overall status of the Travis-CI build. There are no failing tests in the fixed version, and the number and names of failing tests in the buggy version match the reference logs. However, when reproducing the fixed version, the Travis-CI build status is

`failing` because some other CI steps fail after tests run. In this case, the artifact is deemed broken by the default reproducibility criterion of BUGSWARM, but reproducible for all other criteria.

Finally, the reproducibility of BEARS is 54.6% with 137 out of 251 artifacts being reproducible. After 46 months from creation, 117 artifacts have broken. Similar to BUGS.JAR, despite using the least restrictive Existence criterion by default, reproducibility remains almost the same when applying Number/Name Match, achieving a reproducibility of 53.4%.

Interestingly, while there is an overlap of projects from which some datasets mine artifacts, reproducibility at the project level varies significantly across datasets. For instance, both DEFECTS4J and GROWINGBUGS mine artifacts from JacksonDatabind, and both BUGSWARM and BEARS mine artifacts from `raphw/byte-buddy`. However, in DEFECTS4J, all 112 artifacts (100%) mined from JacksonDatabind are reproducible while in GROWINGBUGS, only 12 out of 39 artifacts (30.8%) are reproducible. Similarly, 351 out of 361 artifacts (97.2%) mined from `raphw/byte-buddy` in BUGSWARM are reproducible while none of the 5 artifacts in BEARS are reproducible.

Note that some BUGSWARM artifacts correspond to different build jobs of a same project version, where build jobs differ in the environment used to build (and test) a given project version. We grouped the 1,795 BUGSWARM artifacts by project version, which resulted in 1,152 groups from which 430 groups have more than one artifact. We found that in 67 out of 430 groups some build jobs are reproducible while others are broken. For example, `raphw-byte-buddy-202917181` and `raphw-byte-buddy-202917180` are build jobs of the same project version, but differ in reproducibility. Among the remaining 363 groups, in 262 all build jobs are reproducible while in 101 all build jobs are broken.

Our manual inspection of breakages revealed that artifacts from different datasets are broken due to similar reasons. Reasons for breakages include compile errors, unexpected test failures observed in the fixed version, and expected test failures in the buggy version not being observed. There are also common root causes leading to breakage. For example,

the deprecation of Java 7 caused artifact breakages for all datasets except for BUGSWARM, which leverages containerization to use the reference environment to run artifacts. Other root causes include missing compatible dependencies and deprecated insecure links. Section 3.3 will provide an in-depth investigation in the context of BUGSWARM.

**RQ2 Summary:** All datasets experience breakages, especially those created automatically. DEFECTS4J, the only handcrafted dataset, reaches the highest reproducibility of 96.9%, while BUGS.JAR has the lowest 26.6%. Despite adopting different criteria, reproducibility for a given dataset only varies slightly. On the other hand, artifacts from a same project can have different reproducibility across datasets.

### 3.3 A Case Study on Reproducibility

The goal of this study is to determine how software defect artifacts experience breakage through time, and how to ensure long-term reproducibility. Our study is threefold. First, we study the frequency at which software artifacts break. Second, we examine breakages to identify common root causes and fixes. Lastly, we discuss and evaluate strategies to prevent software breakage in the context of software defect datasets. We aim to answer the following research questions:

**RQ3** How often do software breakages occur?

**RQ4** What are root causes & fixes for software breakages?

**RQ5** How can we prevent software breakages?

**BugSwarm Dataset.** This case study is conducted on the dataset BUGSWARM [36, 351]. BUGSWARM is a software defect dataset designed to continuously grow. Upon our first adoption of BUGSWARM, the dataset included 1,941 Java artifacts from 120 different open-source projects hosted in GitHub that use Travis-CI, from which 146 are marked as “flaky.” BUGSWARM provides a Docker image for each artifact, which consists of the buggy and fixed versions of the source code, the exact build environment in which the code was first built, and a script to build the project and run existing tests. All the artifacts in BUGSWARM use



a build system. Among non-flaky artifacts, there are 1,640 artifacts that use the Maven build system, 69 that use Ant, and 86 that use Gradle.

We select BUGSWARM as the subject dataset for this study because: (1) it is an automatically created software defect dataset subject to software breakage, (2) its artifact containerization facilitates investigating and fixing breakages, (3) it provides software defects both on source code and configuration files, (4) its scale is relatively larger than others, and (5) it is in active development.

***Experimental Setup.*** We study all 1,795 non-flaky Java defect artifacts from BUGSWARM, all of which were reproducible when the dataset was created. At regular intervals, on average every 11.7 days over a 13-month period, we tested the reproducibility of the software artifacts in BUGSWARM. Each run of reproducibility test for all artifacts is referred as a *test suite*. We collected the results for a total of 36 test suites. The testing process consisted of (1) pulling the corresponding Docker images, (2) running the build scripts for both buggy and fixed versions, which download dependencies, build the projects, and run tests, and (3) comparing the reproduced logs against the reference logs, which are the Travis-CI historical build logs. We report reproducibility results as calculated by the criteria originally used by BUGSWARM—the status of the build (**passing**, **failing**, or **error**) is compared along with the number of passing & failing tests and the names of failing tests (if any), for both buggy and fixed versions. An artifact is deemed *reproducible* if all of these attributes match, otherwise the artifact is *broken*. Recall that this is the most strict reproducibility criterion discussed in Section 3.2.2.

### 3.3.1 Breakage Frequency

In this section we investigate how often software breakages occur. In particular, we first find how many of all artifacts have ever suffered from breakage during the 13-month study, and whether this is a one-time occurrence. Second, we investigate the overall frequency of breakage, e.g., whether artifacts gradually break on every test run, or there are specific

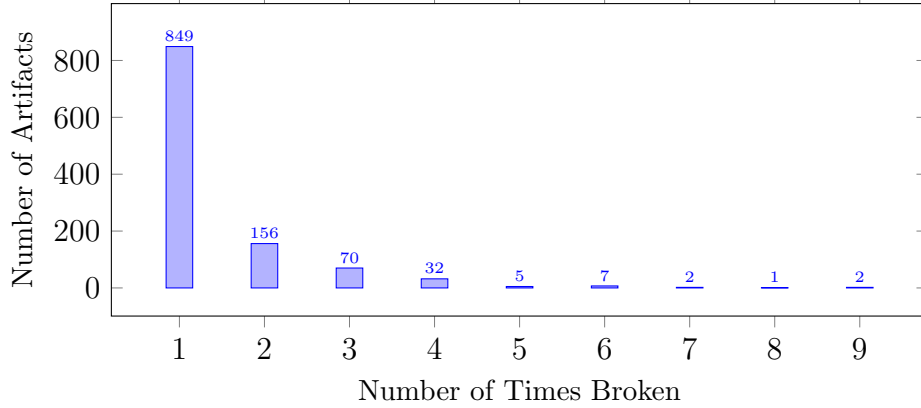


Figure 3.1: Count of artifacts with a number of breakages.

points in time in which large batches of artifacts break at once.

To understand the scale of breakage, we count the number of artifacts with breakages across all 36 test suites. The breakage count increments every time the status of an artifact changes from reproducible to broken. We find that 1,124 out of 1,795 artifacts (62.6%) broke at least once during the 13-month study. These artifacts belong to 86 out of 120 Java projects (71.6%). This shows that breakage is not a rare occurrence and that it affects a large majority of artifacts across different projects.

Figure 3.1 shows the distribution of artifacts in terms of the number of breakages reported during the study. A total of 275 (15.3%) artifacts broke multiple times with almost half of those breaking three or more times. Note that the number of artifacts decreases exponentially as the number of breakages increases. This is expected as broken artifacts remain broken until a patch is applied to fix the breakage.

On average, 430 (24%) artifacts remained broken in each test suite. All artifacts were reproducible when created, but as time passed they suffered from breakages for various reasons, which we describe later in Section 3.3.2.

Figure 3.2 shows the number of *newly reproducible* and *newly broken* artifacts identified in various test suites. A newly reproducible artifact in a given test suite is an artifact that was broken in the previous test suite, but that just became reproducible again because a patch was applied to fix the breakage. A newly broken artifact is an artifact that was reproducible

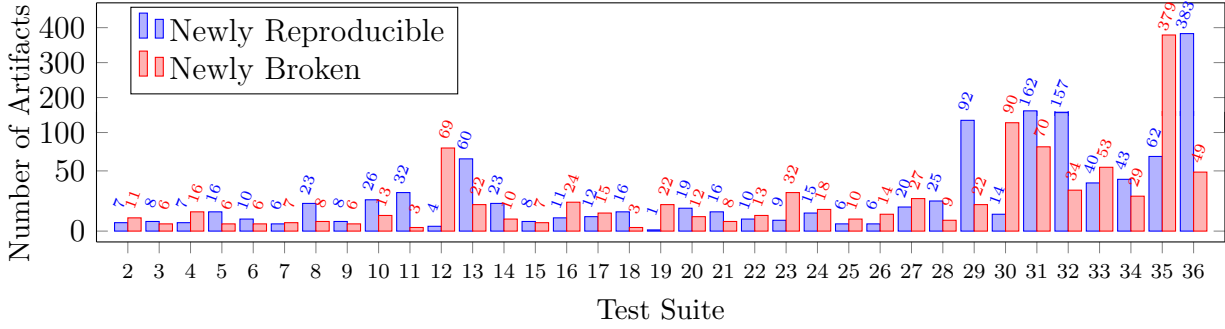


Figure 3.2: Number of newly reproducible and newly broken artifacts for each test suite.

Table 3.4: Root causes and patches for breakages.

| Category # | Root Cause                   | Patch                                         | # Patches Applied | Percentage |
|------------|------------------------------|-----------------------------------------------|-------------------|------------|
| 1          | Maven TLS Failure            | Update TLSv1.0 to TLSv1.2                     | 730               | 24.8%      |
| 2          | Unavailable PPAs             | Remove PPAs no longer available               | 679               | 23%        |
| 3          | Unavailable Ubuntu Release   | Change URLs for repository                    | 392               | 13.3%      |
| 4          | Insecure Link                | Change URLs using HTTP to HTTPS               | 339               | 11.5%      |
| 5          | Unavailable JDK Version      | Retrieve JDK version from official repository | 316               | 10.7%      |
| 6          | Unavailable Gradle Plugin    | Update URL of specific Gradle Plugin          | 158               | 5.4%       |
| 7          | Unavailable NodeJS Installer | Change URL to retrieve NodeJS installer       | 91                | 3.1%       |
| 8          | Incompatible NPM Package     | Pin NPM package version                       | 91                | 3.1%       |
| 9          | Unavailable XML              | Update URL to retrieve DTD files              | 83                | 2.8%       |
| 10         | Deprecated Checkstyle Link   | Replace deprecated checkstyle URL             | 69                | 2.3%       |
| 11         | Unexpected Test Failures     | N/A                                           | -                 | -          |

in the previous test suite, but it is now broken. For example, in test suite #36 in Figure 3.2, there are 383 previously broken artifacts that are fixed, but an additional 49 artifacts are now broken.

For the 36 test suites, the average of newly reproducible artifacts is 38, and 32 for newly broken artifacts. This suggests that the number of artifact breakages in the context of two consecutive test runs is relatively small, considering a total of 1,795 artifacts. Nevertheless, this is problematic as it demonstrates a need for high effort to maintain reproducibility; there are 32 new artifacts that experience breakage every 11.7 days on average. The maximum number of newly broken artifacts across all test suites is 379, and the maximum number of newly reproducible artifacts across all test suites is 383. The median for newly broken artifacts is 14, and for newly reproducible artifacts is 16. The above demonstrates the brittleness of artifact reproducibility and the challenge facing the maintenance of long-term reproducibility for an automatically created dataset.

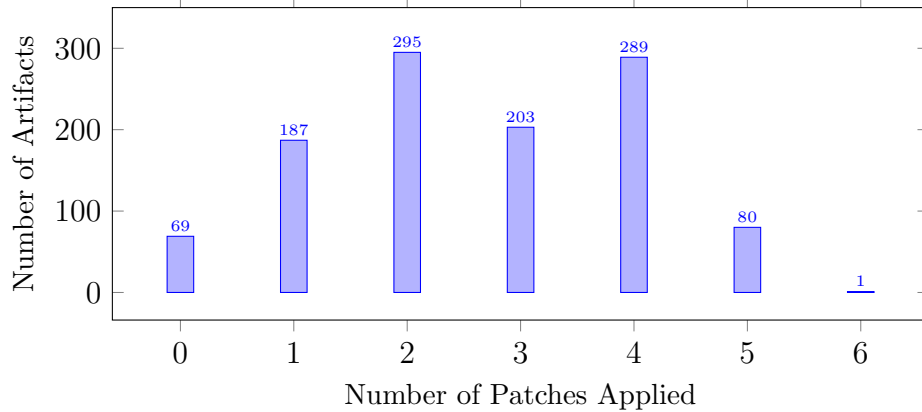


Figure 3.3: Count of artifacts with a number of patches.

**RQ3 Summary:** A total of 1,124 (62.6%) unique artifacts broke during our 13-month study. This includes artifacts from 86 (71.6%) different Java projects. For each test suite, 32 newly broken artifacts were observed on average. The rate at which artifacts break imposes a great challenge on the long-term maintainability of the dataset.

### 3.3.2 Breakage Root Causes and Fixes

Our 13-month study revealed a total of 1,606 individual instances of software breakage that affected 1,124 unique artifacts. We manually examined each breakage to identify root causes and potential fixes. Specifically, for each instance, we inspected the errors in the reproduced logs, and also referred to the configuration and source files when necessary. To minimize bias, three individuals worked independently to inspect each breakage instance to generate a consensus on its root cause.

Our manual investigation uncovered 11 root causes of breakage. Table 3.4 provides a brief description of the root causes and their patches. The patches can be categorized into: (1) system dependency and (2) project dependency. A system dependency patch fixes issues of dependency installation via APT. APT is a package manager for Linux that allows users to install, update, and remove packages retrieved from official package repositories such as Ubuntu Archive or from unofficial Personal Package Archives (PPAs). A project dependency patch modifies dependencies specified in build configurations. An important requirement for patches applied to BUGSWARM is that they do *not* make significant changes to the artifact.

These patches are the reason for the newly reproducible artifacts shown in Figure 3.2. Note that all patches are manually created but applied in an automated manner. Among 1,124 artifacts that break at least once, we applied 2,948 patches on 1,055 (93.9%) of them. Note that 868 out of 1,055 (82.3%) fixed artifacts required multiple patches. Figure 3.3 shows the distribution of artifacts based on the number of patches applied. The rest of this section discusses each root cause and fix strategy.

**Maven TLS Failure** This is the most common among all root causes, affecting 730 artifacts from 68 distinct projects. This issue is caused by the deprecation of TLSv1.0. The patch updates TLSv1.0 to TLSv1.2 and falls into the category of project dependency patch. This patch accounts for 24.8% of all the applied fixes.

**Unavailable PPAs** A total of 679 artifacts from 56 different projects were affected by unavailable PPAs. The breakage occurs when a previously accessible resource becomes unavailable. These particular resources are PPAs, i.e., Personal Package Archives, which are used on Ubuntu to package system dependencies. However, we found that these dependencies were actually not necessary for the artifacts to build successfully, and thus it is safe to remove the installation command that was causing the breakages. More specifically, the patch consists of removing the PPAs from the list of repositories that are retrieved when running `apt-get update`. This is considered as a system dependency patch, which corresponds to 23% of the fixes.

**Unavailable Ubuntu Release** Ubuntu 12.04 LTS's Precise Pangolin packages and releases were migrated to a new repository, making previous URLs used by the package managers to become stale. The patch consists of updating the APT's sources list to install the Ubuntu 12.04 LTS releases via the new URL. This affected 392 artifacts from 7 distinct projects and accounted for 13.3% of the fixes.

**Insecure Link** Maven central repository stopped the transfer of dependencies for HTTP protocol requests, instead requiring HTTPS. The transfer failed with a 501 return code resulting in unresolved dependencies. This affected 339 artifacts from 44 distinct projects. The fix was to replace URLs HTTP with HTTPS in affected URLs, and it accounted for 11.5% of the fixes.

**Unavailable JDK Version** This breakage occurred when artifacts relied on a third-party server to retrieve a specific version of the Java Development Kit (JDK), however the server was no longer accessible. The patch simply retrieves the missing JDK package from the official repositories. This patch is applied to 316 artifacts. Note that while the number of artifacts is large, they belonged to only 3 projects. Overall, this category represents 10.7% of the fixes.

**Unavailable Gradle Plugin, XML and NodeJS Installer** Unavailable Gradle Plugin is observed when plugins originally hosted in non-central repositories are no longer accessible. Similarly, Unavailable XML occurs when `.xml` files become inaccessible due to URL changes. Finally, Unavailable NodeJS Installer also refers to a stale URL. The installer was previously downloaded from an Amazon Simple Storage Server (S3) Bucket instead of the official NodeJS installer source URL. All these examples reveal the danger of relying on third-party sources to host required resources. All together, these accounted for 332 fixes on artifacts from 3 distinct projects, and represented 11.3% of all applied fixes.

**Incompatible NPM Package** This is caused by an incompatibility with the latest release of the NPM package. Our investigation found that NPM's semantic versioning was set to be greater than or equal to a certain version, e.g., `"my_dep" : ">1.0.0"`. As a result, a latest version was installed instead of the original version used by the project. The patch consists of pinning the specific package version used when the project was originally built. This affected 91 artifacts all from one project, accounting for 3.1% of the breakage fixes.

**Deprecated Checkstyle Link** These breakages occur when the link to retrieve the Checkstyle package is deprecated. Checkstyle is a development tool to examine the adherence of coding standards. The deprecation affected 69 artifacts from a project that retrieves the Checkstyle package during the build process. The patch simply updates the link of the Checkstyle package, accounting for 2.3% of the fixes applied.

**Unexpected Test Failures** Finally, 69 artifacts were broken due to unexpected test failures; the number of failing tests do not match the reference log. Our further investigation identified four sources of test failures: expired API key for tests, bad SSL certificate, timeout tests, and errors from the Gradle build system, for which we do not provide patches.

### Final Remarks for Root Causes and Fixes

Artifacts break for different reasons, and often experience multiple breakages over time. Our patches are applied to 1,055 artifacts in 2,948 fixes. Note that *Maven TLS Failure*, *Insecure Link*, *Unavailable Gradle Plugin* and *Deprecated Checkstyle Link* are patches that fix project dependencies, which are retrieved by the build system and account for 44% of all fixes. The rest (except for unexpected test failures) could be attributed to system dependencies. We discuss next how project dependencies represent low-hanging fruit for preventing future breakage.

**RQ4 Summary:** We identify 11 root causes of breakages and manually crafted patches to fix ten of them. We automatically apply 2,948 patches on 1,055 artifacts. 1,296 (44%) of the fixes are related to project dependencies.

### 3.3.3 Breakage Prevention

In this section, we discuss preventive strategies to ensure long-term reproducibility of software defect artifacts. First, we describe how we leverage popular build systems to cache project dependencies. This was inspired by the insight from our study on root causes and fixes (Section 3.3.2): 1,296 of all patches are applied to fix errors from dependencies retrieved by

build systems. We discuss the impact of dependency caching on the reproducibility of 20 additional test suites over a time period of 8 months. Finally, we discuss the need for full isolation of artifacts to ensure long-term reproducibility.

## Dependency Caching

To prevent BUGSWARM artifacts from breaking due to project dependencies managed by their build system, we add required dependencies for each artifact to the corresponding Docker image (recall that artifacts from BUGSWARM are instantiated as Docker containers). We refer to this process as *dependency caching*, which can be automated with the help of the build systems (e.g., Maven). The ultimate goal is to make artifacts independent from external repositories. Artifacts whose dependencies have been cached (also referred to as *cached artifacts*) run in the same way as the original artifacts, but no longer need access to an external central repository to download dependencies.

**Approach for Dependency Caching** We use the local repository configuration provided by the build system. We describe our process for Maven, which is similar for Gradle and Ant. We set a local repository path in Maven’s XML configuration file. This local repository is then used to automatically save dependency files downloaded from remote repositories when the build script invokes `mvn install` [53]. After downloading all dependencies, we configure the artifact to build offline with the `-o` option in `mvn install`. This option forces the project to build only from dependencies in the local repository. Note that Ant supports caching but does not have an offline mode. So we assume that the cached Ant artifacts are cached correctly as long as their build script does not break. We check for the existence of additional dependencies that may require caching in Section 3.3.3.

**Effectiveness of Dependency Caching** A total of 1,700 artifacts in BUGSWARM are reproducible after the patching process described in Section 3.3.2, and thus eligible for dependency caching, including 1,566 Maven artifacts, 71 Gradle artifacts, and 63 Ant artifacts.



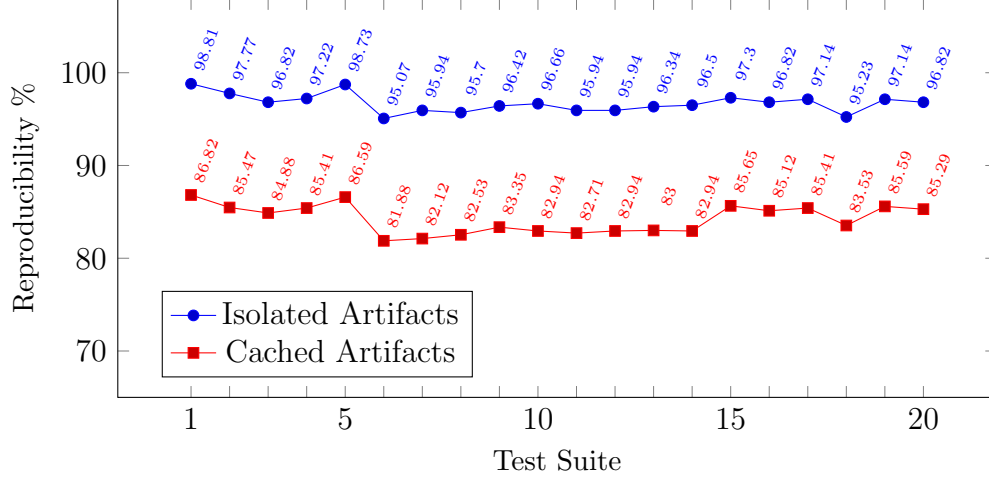


Figure 3.4: Reproducibility of cached and isolated artifacts.

We successfully cached all artifacts automatically. Cached artifacts are tested by running their build script using the build system’s offline mode to restrict their network access. Since dependency caching adds new files to the Docker container, the size of Docker images in BUGSWARM increases. However, to our surprise, the increase in size is rather small. For each artifact, after caching all its required dependencies, the size increase rate is 5.1% on average, and the 25<sup>th</sup>, 50<sup>th</sup>, 75<sup>th</sup>, and 90<sup>th</sup> percentiles of the size increase are 2.5%, 3.5%, 5.9%, and 12.2%, respectively.

We evaluated the reproducibility of cached artifacts across 20 test suites over a period of 8 months, subsequent to our initial study (Section 3.3.1), and a few weeks after the caching process concluded. The results are shown in Figure 3.4. While we observe a decrease in reproducibility, the reproducibility is mostly maintained in the 81% to 85% range with no maintenance effort. The reason for breakage is mainly due to dependencies that were not captured by our approach. Recall that we cache dependencies via Java build systems like Maven, but there are some artifacts fetching dependencies from other sources. Unexpected tests failures are the main reason for the variability across test suites. Some tests fail due to the temporary unavailability of remote services. For example, tests of `apache-dubbo-416755517` invoke the RPC service, which was unavailable during the runs of test suites #6 & #7 and restored for later test suites.

Table 3.5: Reproducibility without internet access.

| Description                               | Number | Ratio   |
|-------------------------------------------|--------|---------|
| Cached and reproducible artifacts         | 1,700  | 100.00% |
| Artifacts currently reproducible offline  | 1,257  | 73.9%   |
| <i>Not Reproducible – Internet Access</i> |        |         |
| Cached plugin accesses the internet       | 157    | 9.2%    |
| Internet access in unit tests             | 80     | 4.7%    |
| Build script accesses GitHub at runtime   | 68     | 4%      |
| <i>Not Reproducible – Caching Issues</i>  |        |         |
| Incompletely cached, missing dependencies | 51     | 3%      |
| Irregular dependency installation         | 32     | 1.9%    |
| Misc. runtime errors                      | 55     | 3.2%    |

## Artifact Isolation

Ideally, a cached artifact should be reproducible even without internet access because all its required dependencies are available in its Docker container. If an artifact is reproducible without access to external resources like the internet, then it should be reproducible in perpetuity. To verify if cached artifacts are completely reproducible, we tested the reproducibility of cached artifacts when disconnected from the internet. We found that only 920 out of 1,700 (54.1%) artifacts were reproducible.

We manually inspected the remaining 780 artifacts to identify the root cause. We identified three root causes that we were able to fix, bringing the number of successfully isolated artifacts to 1,257 (73.9%). The most frequent issue was build failure during the Travis initialization stage. This was observed for artifacts that use Travis’s temporary caching functionality, which uploads specific dependencies to the internet for later use [59], and thus failed in isolation. This initially affected 243 artifacts that are now reproducible in isolation after disabling Travis caching. The second reason was related to downloading packages via NPM from the internet. Note that our caching process only caches dependencies via build systems. Thus, without internet access, the build script is not able to retrieve NPM packages and thus fails. This affected 88 artifacts that are now reproducible. The remaining 6 artifacts had port issues that we fixed by enabling localhost within Docker containers.

A total of 443 (26.1%) cached artifacts are broken without internet. Table 3.5 shows two main categories: (1) *Internet Access*, artifacts that explicitly access the internet during

the build process, and (2) *Caching Issues*, artifacts that access the internet to upload or download files in ways that are not captured by our dependency caching techniques.

*Internet Access* accounts for 17.9% of cached artifacts. It is unusual to observe software artifacts that download content from hardcoded URLs at runtime because this practice creates test instability. Among these artifacts, calls to the internet were included in the tests, build scripts, or in the installation process of cached dependencies. These are not recommended developer practices for offline mode artifacts according to Maven’s documentation [53]. These artifacts would be much more difficult to reproduce in isolation, and each fix would be artifact specific, so we leave the process of mocking specific remote resources for future work. *Caching Issues* affect 8.1% of the artifacts. This is mainly due limitations of our caching technique. For example, 51 Ant and Gradle cached artifacts are missing system dependencies. Unlike the artifacts that explicitly make calls to the internet, these artifacts are likely to be reproducible with refinements to the dependency caching approach or manual caching of missing files.

Like cached artifacts, we evaluated reproducibility of isolated artifacts with 20 test suites run over 8 months. Figure 3.4 shows the reproducibility of isolated artifacts along with that the reproducibility of artifacts cached. Note that reproducibility of isolated artifacts stays above 95%. We inspected a few of the broken artifacts, and found that breakage is due to a timeout. In particular, this only affects artifacts belonging to one of two projects: `rinde-RinSim` and `spring-projects-spring-data-jpa`. Further investigation is required to determine the reasons for the timeout.

## Final Remarks

Note that BUGSWARM as of July 2022 included all the patches from Section 3.3.2, which were acknowledged and pushed to BUGSWARM’s main Docker repository. However, despite all the breakage fixes, reproducibility dropped to 69% due to further breakage. On the other hand, the official version of the dataset does *not* include yet our mechanisms for breakage

prevention. We applied these mechanisms to a copy of the dataset, which allows us to further evaluate the impact of dependency caching and artifact isolation. We demonstrate that breakage prevention contributes to maintaining high reproducibility after 8 months of 85.29% and 96.82% for cached and isolated artifacts, respectively, in comparison to 69% reported when no breakage prevention is in place. This is remarkable as the reproducibility is on par with that of handcrafted DEFECTS4J. Furthermore, while there is a raising concern that limitations on size and diversity of current defect datasets might introduce bias, the automated process of caching dependencies and isolating artifacts enables the continuous growth of a large-scale and diverse dataset with long-term reproducibility.

**RQ5 Summary:** Dependency caching is effective in preventing breakages. Isolation enables reproducibility even without internet. Reproducibility of 1,257 isolated artifacts stays above 95%, which is on par with handcrafted quality.

### 3.4 Threats to Validity

Our study on five Java datasets may not generalize to other languages, or even other Java datasets. However, we believe that our choice of language and datasets is representative; Java is the most popular language in fault localization and automated program repair, and we consider *all* available datasets that provide complete Java projects, which is a requirement to build and run the code.

The lack of runtime environment information for some datasets is another potential threat. In particular, BUGS.JAR and BEARS do not specify the Java version required to reproduce their artifacts, which increases the risk of incorrect execution. We contacted authors of BUGS.JAR and BEARS to request information about the runtime environment of their dataset, but we did not receive a response. Thus, to reduce this risk, we manually examined all artifacts in these datasets and identified 147 out of 1,158 artifacts from BUGS.JAR and 95 out of 251 artifacts from BEARS that require Java 7 or older. To measure the impact of this in our study, we set up a virtual machine with deprecated Java versions and ran the artifacts. Despite using the required deprecated Java versions, all artifacts were still broken.

The reason was the Maven TLS Failure discussed in Section 3.3.2. Finally, several artifacts across datasets belong to projects from the Apache Software Foundation, which may impose IP bans. We have ensured breakages do not occur because of this reason.

Despite all manual efforts, there is still room for human error when inspecting breakages. We minimize this risk by having multiple people do manual inspections following a uniform procedure. While the root causes of breakages are observed in all five datasets, the fixes we provide are specific to BUGSWARM’s breakages. It remains future work to ensure such fixes generalize to other datasets. Additionally, although all the fixes are applied in an automated manner, the process of creating patches remains manual. However, the number of broken artifacts will significantly decrease as new artifacts are successfully cached and isolated. Finally, our dependency caching approach heavily relies on the build systems. It remains future work to adapt this approach beyond the build systems used by BUGSWARM’s artifacts.

## 3.5 Related Work on Defect Dataset Reproducibility

**Understanding Build Failures** Prior studies have analyzed build failures based on 26.6 million builds at Google [312], 3412 builds from a commercial web application [194], and 2.6 million CI builds [85], finding that common build errors range from missing dependencies and other resources to test failures. Others have studied the *feasibility* of building software, such as the top 200 Java projects from GitHub [167], multi-language software packages [262], broken snapshots of Java Maven projects [353], and 7200 Java projects that use different build systems [334]. Unlike the above, we study builds that were reproducible *after* their commit time and track their reproducibility to present day. We find that the main reason for breakage is related to dependencies, which is touched on by [167, 194, 312, 353].

**Fixing Broken Builds** There are techniques that automatically fix broken builds. BuildMedic [238] fixes dependency-related broken Maven builds by applying simple fix strategies such as Ver-

sion Update or Dependency Deletion, which effectively fix 54% of the broken builds in 23 open-source projects. HireBuild [166] is a pattern-based patch generation approach for Gradle build scripts that fixes 45% of previously unseen broken builds. PyDFix [256] detects and then automatically fixes unreproducible builds due to dependency conflicts for Python projects. Vassallo et al. [356] presents BART, which provides build fix suggestions from StackOverflow. Deep learning approaches have also been introduced to fix compilation errors [247], to predict edits to fix a broken AST [343], and to fix broken Dockerfiles [168]. Similar to us, [166, 238, 356] use either common fix strategies or patterns that are empirically discovered. Furthermore, the fix for a broken build can be performed in the source files [247, 343, 356], or in the build files [166, 238, 256]. As the artifacts in our study were previously reproducible, the issue is likely related to build files. While BuildMedic and HireBuild can repair Java build-related issues, each is specific to one build system, while the builds in our study encompass three build systems.

**Studying Reproducibility** Different aspects of software reproducibility have been studied over the years, which include quality issues and reproducibility of Dockerfiles [115], reproducibility of 30 million CPAN (Comprehensive Perl Archive Networks) builds between 2011 and 2016 [461], reproducibility of past snapshots [241], reproducibility of GITHUB ACTIONS workflow runs [456], and reproducibility of specific kinds of bugs such as performance bugs [163], error propagation bugs [126], and concurrency bugs [382]. Furthermore, Frattini et al. [143] provide a bug taxonomy from the perspective of reproducibility with two categories: workload-dependent and environment-dependent. 18% of bug reports from MySQL Server are environment-dependent. Cavezza et al. [101] investigate environment-dependent failures reported for MySQL Server, and how altering the environment affects the reproducibility of those failures. There are also frameworks to *prevent* breakage by preserving software and its dependencies from source to execution [273, 345]. In this section we investigate the impact of software breakages over time for general software defects, propose fix

strategies for breakage and discuss preventive measures to ensure long-term reproducibility.

## 3.6 Reproducing GitHub Actions

Continuous Integration (CI) and Continuous Delivery (CD) play important roles in modern software development. They allow developers to configure and deploy a process to be automatically triggered by designated events (e.g., `git push`). In most cases, the CI/CD process will involve setting up the environment, building the code, and/or running tests. There are various popular CI/CD service providers, including TRAVIS-CI [58], GITHUB ACTIONS [43], and JENKINS [51].

The wide use of CI/CD has opened opportunities for the software engineering (SE) research community. Specifically, it has given researchers access to the build and testing processes of software projects at a large scale. Several research areas within SE, such as fault localization and automated program repair, are evaluated on large sets of successful and failed CI/CD builds. Furthermore, previous work has leveraged CI/CD to mine large scale code repositories and build software defect datasets [239, 351], to predict build outcomes [164, 165], or to automatically repair build scripts [166, 256].

To realize the full value of CI/CD from a research perspective, it is essential to be able to reproduce the build processes at scale whenever needed. Successful reproduction means the ability to run exactly the same steps in exactly the same environments and obtaining exactly the same outcomes as when the process was first completed (e.g., the code was pushed to GITHUB). Unfortunately, the process of reproducing builds is hampered by many challenges such as the unavailability of software dependencies [256, 351]. Previous work [351] proposed a methodology to mine and *reproduce* builds of GITHUB projects that use TRAVIS-CI by using TRAVIS-CI DOCKER images to recreate the exact build environments and TRAVIS-BUILD [57] to automatically create build scripts. While having reproduced thousands of builds, the success rate was extremely low. Most recently, reproducing TRAVIS-

CI builds has become even more challenging: following a change in business model, TRAVIS-CI historical logs for a majority of projects are no longer available, and DOCKER images capturing environments are no longer publicly accessible.

As mentioned earlier, other CI/CD services have gained popularity in the past few years. Among the existing CI/CD service providers, GitHub Actions [43] is one of the most popular due to its deep integration with GITHUB, which is the most popular software project hosting provider with more than 350 million code repositories [42]. GitHub Actions uses *workflow runs* to refer to build processes. However, to reproduce workflow runs for GitHub Actions, there are several challenges. First, GitHub Actions by default uses Azure VMs to host workflow runs so it is not easy to reconstruct the exact runtime environment while ensuring portability. Second, unlike TRAVIS-CI, GitHub Actions does not provide an official converter from workflow files to build scripts, thus a parsing process is required to run a workflow “offline.” To address these challenges, it requires (1) proper containerization techniques to reconstruct the runtime environment of a build, and (2) a sophisticated parser to convert the non-executable workflow file to executable build scripts.

In this section, we demonstrate ACTIONSREMAKER, a reproducer for GitHub Actions workflows. Given a desired workflow job to reproduce, ACTIONSREMAKER will first reconstruct the original runtime environment by building DOCKER containers with the information retrieved from the original workflow job. Then GITHUBBUILDER translates the workflow file into a build script. Finally, ACTIONSREMAKER packages the software repository and build script along with the required environment into a DOCKER image. To the best of our knowledge, ACTIONSREMAKER is the first tool designed to reproduce existing GitHub Actions workflow runs.

We evaluate ACTIONSREMAKER by examining its capability to reproduce fail-pass build pairs. Fail-pass pairs are consecutive builds where the first fails and the second passes, and have been used to construct software defect datasets such as BUGSWARM [351]. BUGSWARM is a software defect dataset mined from TRAVIS-CI builds, of which workflows are defined as



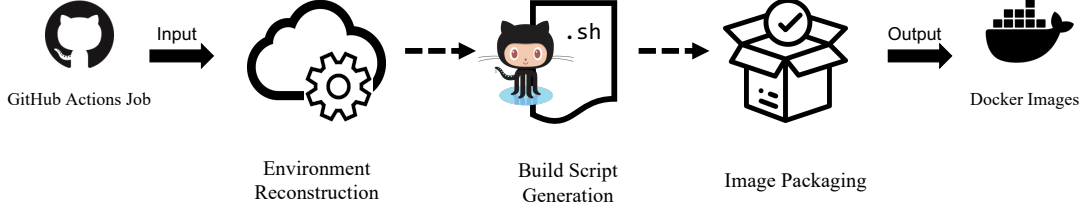


Figure 3.5: Steps of ACTIONSREMAKER.

.yaml files that can be translated to shell scripts by the official converter provided by TRAVIS-CI. We reuse the pipeline from BUGSWARM to *mine* fail-pass pairs from GITHUB projects. Among 180 fail-pass pairs mined from 67 projects, ACTIONSREMAKER successfully reproduced 130 (72.2%) of them, which is dramatically higher than the reproducibility reported by BUGSWARM (5.56%) when reproducing TRAVIS-CI builds. We also study and categorize the root causes of unreproducible pairs. The rest of this section presents background on GITHUB ACTIONS, the approach of ACTIONSREMAKER, and its evaluation.

### 3.7 Background

GitHub Actions is a CI/CD service provided by GITHUB. All configurations and steps are defined in .yaml formatted files in the .github/workflows directory of the repository, which are referred as workflow files. Each of workflows will contain events, jobs, steps, and runners configured by developers. Events are specific activities that trigger a workflow run, such as a *push* to a branch or a *pull request*. In a workflow run, there could be multiple jobs for different environments. Each job consists of a set of steps to be executed. Steps are the smallest portable unit in GitHub Actions. They can be either *predefined actions* or *custom actions* from GITHUB or developers. Runners are the virtual machines where jobs are executed. Developers can use matrix parameters to define a set of environments by combinations of different operating systems, programming languages, and compiler versions for runners. A workflow run triggered by an event executes all jobs defined in the workflow file. Finally, each job executes all steps defined in the job in a runner. The whole process is

referred as a *build*.

## 3.8 Approach

In this section, we describe ACTIONSREMAKER’s approach to reproduce GitHub Actions builds. As shown in Figure 3.5, ACTIONSREMAKER takes as input the desired GitHub Actions job, and outputs a DOCKER image in which the job can be easily reproduced. ACTIONSREMAKER first reconstructs the environment based on information retrieved from the original job. Then, GITHUBBUILDER generates a build script based on the workflow file. Finally, with the original environment, ACTIONSREMAKER packages the build script and software repository into a DOCKER image. We discuss limitations of ACTIONSREMAKER at the end of the section.

### 3.8.1 Environment Reconstruction

Given the job ID, ACTIONSREMAKER can retrieve the environmental information via the GITHUB API to reconstruct the original runtime environment. The reconstructed environment includes the operating system, software repository, predefined actions, and the original logs of the job.

For GitHub Actions, operating systems are specified in the workflow file with the `runs-on` option. Based on this value, ACTIONSREMAKER will determine the original operating system to construct with. However, if the users specify `ubuntu-latest` as the operating system version, ACTIONSREMAKER will parse the original log to get the correct operating system version because `ubuntu-latest` could vary based on the time when the original job was triggered. Next, ACTIONSREMAKER downloads the software repository with the exact revision that corresponds to the job. The GitHub Actions official runner requires developers to use the `actions/checkout` action at runtime to set up the repository. For ACTIONSREMAKER, it retrieves the project source code by cloning the project and runs `git reset` to reset the

project back to the target commit. Sometimes the target commit is not in the git history thus it is not resettable. In such cases `ACTIONSREMAKER` downloads the zip archive of the commit directly from `GITHUB`. `ACTIONSREMAKER` also retrieves the source code for all predefined actions based on the action name and version listed in the job’s workflow file. Moreover, since the goal of `ACTIONSREMAKER` is to reproduce existing workflow runs, it also downloads the original build logs from `GITHUB` API for future validation.

### 3.8.2 Build Script Generation

`GITHUBBUILDER`, a core component of `ACTIONSREMAKER`, automatically generates shell scripts from the workflow file for running a GitHub Actions job. `GITHUBBUILDER` retrieves additional workflow metadata like `actor` (the user that triggered the workflow run) and `head ref` (the source branch of the pull request) via the `GITHUB` API. It also retrieves runtime information like global environment variables, default shell, and working directory from the workflow file. If a job was triggered by a pull request (PR), the PR number is crucial for job reproduction since oftentimes the job will require information such as the title and head repository name of the PR. However, the PR number is not accessible from the API. To address this, `ACTIONSREMAKER` parses the original build log to extract the PR number, then uses the `GITHUB` API to fetch all PR-related information.

GitHub Actions workflow files contain a list of steps for a job. A step could be either a custom action (specified using the “run” field) or a predefined action (specified using the “uses” field). Regardless of action type, `GITHUBBUILDER` will extract metadata such as the step name, step number, the conditions to run the step, etc. For custom actions, `GITHUBBUILDER` will save the user-defined commands into a script file, then add a shell command that executes this script into a main build script. For predefined actions, `GITHUBBUILDER` supports JavaScript and composite actions. To reproduce those, `GITHUBBUILDER` first downloads the source code of the predefined action and its metadata file based on the version listed in the workflow file. Then, to reproduce JavaScript predefined actions, it

will use Node.js to run the main program specified in the action metadata file. Just like GitHub Actions, GITHUBBUILDER also passes arguments to JavaScript programs using a list of environment variable. For example, the `actions/setup-java` action requires users to specify the `java-version` option. So if we have `java-version: 17`, it will set the `INPUT_JAVA-VERSION` variable to 17. A composite action is similar to the workflow file; it contains a list of custom and predefined action steps, which can be grouped. To implement the composite action, ACTIONSREMAKER recursively generate the build scripts for this list of steps, similar to how it generates build commands from the steps list in the workflow file.

When generating build scripts, GITHUBBUILDER needs to handle *contexts* and *expressions* used in workflow files. Contexts are the collection of variables containing information about the workflow run, steps, environment variables, runner, etc., and expressions are the combinations of literal values, contexts, operators, and functions. For example, the `startsWith(searchString, searchValue)` function will return `true` if the `searchString` starts with `searchValue`. They are generally used in the conditional statements to control step execution. GITHUBBUILDER will replace static contexts (constant variables such as `github.repository`) with strings and replace dynamic contexts (non-constant variables such as `steps.<step_id>.outcome`) with environment variables. GITHUBBUILDER then converts expressions into lists of arguments to evaluate them at runtime.

### 3.8.3 Image Packaging

The final step is to build the DOCKER image. In this step, ACTIONSREMAKER generates a DOCKER file to specify a list of instructions for the image-building process. Some instructions include: adding a new user with proper permission, changing file permissions, adding the source code of the repository and pre-defined actions, and adding the build scripts. We also set the base image of the DOCKER file based on the job's operating system. Currently, ACTIONSREMAKER supports base images `ubuntu-18.04`, `ubuntu-20.04`, and `ubuntu-22.04`.

### 3.8.4 Limitations of ActionsRemaker

ACTIONSREMAKER does not support certain types of actions: (1) actions that invoke DOCKER are not supported since the runtime environment is constructed within a DOCKER container and running DOCKER in DOCKER will lead to security risks, (2) actions that require secret context are not supported since secrets are private and not accessible to the public, (3) `actions/checkout` actions with specified submodules is not supported only when the job is not resettable, since the downloaded zip archive does not contain the `.git` directory.

## 3.9 Evaluation

In this section, we evaluate ACTIONSREMAKER by measuring its capability to reproduce fail-pass pairs from open-source projects, i.e., we include real-world failures. We answer the following research questions:

**RQ1** How effective is ACTIONSREMAKER at reproducing real-world builds?

**RQ2** What are the root causes of unreproducible builds?

### 3.9.1 Experimental Setup

We reuse BUGSWARM’s pipeline for *mining* fail-pass pairs, which we extended to support GitHub Actions. In the case of GitHub Actions, fail-pass pairs are two consecutive job runs where (1) the first job comes from a failed build and the second comes from a passed build, (2) the two jobs are generated by the same GitHub Actions workflow file and have identical configurations and matrix parameters, and (3) the commits corresponding to each job are from the same branch. To verify reproducibility, we slightly modify BUGSWARM’s log analyzer to make it compatible with GitHub Actions log formats, so it can analyze and compare the reproduced log against the original log.

### 3.9.2 Effectiveness at Reproducing Real-World Builds

We use two sets of real-world fail-pass pairs to create our benchmark. The first set consists of manually-selected fail-pass pairs that make use of important and popular features of GitHub Actions. For instance, we chose pairs from the repository `Netflix/spectator` because its jobs dynamically modify environment variables when they run, and we chose pairs from `junit-team/junit5` because they use composite actions. Such features are expected to be handled by `ACTIONSREMAKER`. The manual selection yielded 87 fail-pass pairs from 11 repositories.

The second set is generated by mining GITHUB repositories with the adapted BUGSWARM pipeline. For a repository to be considered, it should (1) use GitHub Actions, (2) be written in Java, and (3) have at least 50 stars on GITHUB.<sup>1</sup> The above criteria yielded a set of over 3,600 repositories. To keep the benchmark at a practical size, we selected 75 of these repositories to be included in the benchmark. Then, we filtered out the pairs with inaccessible original logs, pairs with jobs that run on unsupported operating systems (Section 3.8.3), and pairs with unsupported actions (Section 3.8.4). This excluded 19 out of 75 repositories that have no suitable pairs to mine. Since we wanted to collect a diverse set of pairs while keeping the size of the benchmark manageable, we only kept the three most recent pairs from repositories that had more than 3 suitable pairs, and all pairs from repositories with more than one but less than 3 pairs. The process above generated a set of 93 fail-pass pairs from 56 repositories. Combining the two sets yields a benchmark of 180 fail-pass pairs from 67 repositories.

We provide each fail-pass pair in the benchmark as input to `ACTIONSREMAKER`, get the resulting `DOCKER` image, run the generated build script in the `DOCKER` container, and collect build outcomes. Then we use the adapted BUGSWARM analyzer to extract the number of tests run, passed, failed, and skipped, as well as the name of failed tests. We repeat the

---

<sup>1</sup>`ACTIONSREMAKER` is language agnostic, but we focused on Java because it is a popular language, and is supported by the BUGSWARM analyzer.

Table 3.6: Fail-pass pair counts for each type of unreproducible pair.

| Category # | Failure Reason      | Number | Percentage |
|------------|---------------------|--------|------------|
| 1          | Test Mismatch       | 23     | 12.8%      |
| 2          | Missing Requirement | 10     | 5.6%       |
| 3          | Dependency Error    | 8      | 4.4%       |
| 4          | Tool Failure        | 6      | 3.3%       |
| 5          | Other               | 3      | 1.7%       |
|            | Total               | 50     | 27.8%      |

process for the original logs and compare them against the reproduced outcomes. A fail-pass pair is reproducible only if the outcomes from both jobs match their original counterparts. Out of the 180 fail-pass pairs in the benchmark, 130 (72.2%) are reproducible. Out of the 67 repositories represented in our benchmark, there were 39 with all pairs reproducible, and 43 with at least one pair reproducible.

### 3.9.3 Root Causes of Unreproducible Builds

In total, 50 (27.8%) out of the 180 fail-pass pairs were not reproducible. Our manual inspection identified four reasons for unreproducibility, which are summarized in Table 3.6.

**Test Mismatches** 23 pairs failed to reproduce because their number of failed and passed tests did not match those listed in the original logs. The main cause for this is test flakiness. Flaky tests are tests that fail in some runs but pass in others. For example, the test `testSingleValueRandomJoin` from the Apache Lucene project failed in the original GitHub Actions workflow run but `ACTIONSREMAKER` does not always trigger the expected failure. While the source of flakiness can usually be attributed to non-determinism, it is often difficult to ascertain the exact root cause.

**Missing Requirements** 10 pairs failed to reproduce due to missing system requirements such as MySQL and the Android SDK. For purposes of practicality, our base images only contain a subset of the software installed on GitHub Actions’s official virtual machines, thus jobs using not included software will not be reproduced correctly.

**Dependency Errors** Some jobs rely on external services or data that are outside of our control. A total of 8 pairs failed to reproduce due to the inability to access Maven dependencies that are no longer available and external APIs.

**Tool Failures** Sometimes ACTIONSREMAKER fails to package a job properly; for instance a `git reset` might fail unexpectedly. A total of 6 pairs failed due to this reason.

There were also 3 unreproducible fail-pass pairs that did not fit neatly into any one of above categories. The unreproducible jobs in those pairs failed due to, respectively, the job not being run in a clean Git repository, a Maven plugin failing for unclear reasons, and a syntax error in a predefined action.

## 3.10 Related Work on Reproducing CI/CD Builds

***Mining CI Services for Dataset Creation.*** Among popular CI services, TRAVIS-CI in particular has been mined extensively by the SE community for different purposes. Related to dataset creation, TRAVISTORRENT [86] mines TRAVIS-CI to create a large-scale dataset of TRAVIS-CI builds that allows to easily access information from the build logs as well as trigger commits and aggregated CI project data. BUGSWARM [351] and BEARS [239] mine TRAVIS-CI to automatically create software defect datasets. To the best of our knowledge, ACTIONSREMAKER is the first tool that can be integrated into a pipeline that mines GitHub Actions to automate the process of *reproducing* builds.

***Reproducing and Fixing Builds.*** Prior work has studied the reproducibility of builds on DOCKERFILE [115], snapshots of Java projects [241], or Python projects [256]. In the scenario of CI services, BUGSWARM [351] reproduced TRAVIS-CI builds with a reported 5.56% success rate. Others have also proposed approaches to fix broken builds [164, 166, 455]. Unlike the above, this is the first study on the reproducibility of GitHub Actions builds.



**Triggering GitHub Actions.** Open source work [54] developed by the community provides functionality to trigger GitHub Actions workflow runs locally. However, the approach is designed to run a job that has *not* been triggered yet, not for reproducing *historical* workflow jobs. Unlike the above, ACTIONSREMAKER is the first tool targeting the reproduction of existing workflow jobs, and delivering a *reproducible* DOCKER image for future use.

## Chapter Summary

The two halves of this chapter address the decay and growth challenges of dataset infrastructure surfaced by Chapter 2. The reproducibility study shows that all five widely used Java defect datasets are prone to breakages, and that BUGSWARM in particular erodes over time with the majority of breakages traceable to dependency drift; dependency caching and artifact isolation preserve reproducibility at 95%. ACTIONSREMAKER shows that the field can keep up with growth via continuous integration: a substantial fraction of GitHub Actions fail-pass pairs reproduce offline, making CI mining a practical ongoing source of dataset growth. The breakage taxonomy and the ACTIONSREMAKER pipeline together inform the empirical evaluation substrate of Chapter 4, where we evaluate LLM-based program repair tools on real CI/CD failures.

This chapter is based on two papers: “On the Reproducibility of Software Defect Datasets,” co-authored with Cindy Rubio-González and published at the International Conference on Software Engineering in 2023; and “ACTIONSREMAKER: Reproducing GitHub Actions,” co-authored with Kevin Z. Guan, Robert M. Furth, and Cindy Rubio-González and published at the International Conference on Software Engineering, Demonstrations Track, in 2023. Both works were supported by NSF grant CNS-2016735. The artifact of the reproducibility study is available at <https://github.com/ucd-plse/on-the-reproducibility>, and ACTIONSREMAKER is available at <https://github.com/bugswarm/actions-remaker>.

## Part II

# Reliability

# Chapter 4

## Reliability in the LLM Era of Software Engineering

Large language models now sit at two complementary positions in software engineering, and both raise reliability concerns. This chapter addresses them in turn. The first half treats LLMs as opaque components inside software and develops a systematic methodology to debug them when their outputs are incorrect. The second half flips the roles: LLMs power program repair tools that act on software, and we measure how well the leading tools fix real-world continuous integration failures.

### 4.1 Delta Debugging for LLM-Integrated Systems

Across industries, Large Language Models (LLMs) are being incorporated rapidly into software systems to automate tasks that were previously infeasible. LLMs can act as oracles, answering complex questions over almost arbitrary inputs. Complex workflow steps can now be encoded as *instruction prompts* in natural language that describe the task at hand and how the output should be formatted. These prompts can then be appended with data in a variety of formats, such as documents or images, and sent as input to an LLM for processing.

As software engineers, we face the challenge of debugging such prompts. Specifically,

when an LLM produces an undesired output, how can we identify which parts of the input to focus on? Unlike traditional software where we can trace execution paths and inspect intermediate states, LLMs operate as black boxes with complex internal representations that are not directly accessible. Moreover, the sheer size of modern prompts, which can include lengthy documents, multiple examples, and detailed instructions, makes manual inspection and iterative refinement impractical.

LLMs are probabilistic by nature and, in most cases, inherently non-deterministic. Even with a *temperature* of zero, most of the popular models, such as Llama or Claude, still do not produce truly deterministic outputs. Debugging non-deterministic systems without specialized tools is notoriously difficult [248]. Software engineering research has produced many approaches for related challenges: debugging distributed systems [88], detecting and diagnosing Heisenbugs [259], debugging for concurrent systems [280], minimizing UI event traces [116], and debugging microservices [454]. These techniques compare inputs and executions that lead to “good” versus “bad” outcomes, and isolate patterns unique to the failures.

This approach of debugging by comparing passing and failing executions is known as delta debugging [417], which recently had its 25<sup>th</sup> anniversary [418]. While delta debugging has proven effective for traditional software, applying it to LLM-integrated systems presents unique challenges due to their black-box nature and the complexity of modern prompts.

Existing LLM explainability methods [235, 438] have limitations for systematic debugging. Feature attribution and attention visualization techniques require model internals and perturbation methods only observe input-output relationships. Chain-of-thought (CoT) prompting [375] shows step-by-step reasoning, but CoT explanations can be unfaithful [355], i.e., models may generate plausible explanations that do not reflect their actual reasoning.

In this section, we investigate whether delta debugging can be applied to identify relevant parts of an LLM input that lead to an undesired outcome without requiring access to model internals. Our investigation consists of three parts: First, we need to find a way to instrument

the LLM input in a way that allows us to extract from the output which parts of the input were considered relevant by the LLM. Second, we need to show that these parts of the input are indeed relevant and that removing them from the input changes the output (and conversely, that removing unrelated parts of the input does not affect the output). And lastly, we need to show that we can use these parts of the input for delta debugging to identify a compact subset that is sufficient to root-cause the issue and improve the input.

For the first task of extracting relevant parts of an LLM input from its output, we introduce the concept of *semantic markers*: we instrument the original input to the LLM by placing unique markers at regular intervals. For example, for textual input, we can place these markers to delimit every sentence. That is, for an input sentence like “LLMs are great. Debugging is hard.”, we may mark it as `<marker_1>LLMs are great.</marker_1>` `<marker_2>Debugging is hard.</marker_2>`. Then, we amend the original input with additional instructions to include these semantic markers `<marker_i>` in the chain-of-thought (CoT) [375] part of the output. We extract the *CoT trace* from the CoT portion of the output and verify that the markers indeed exist in the LLM input. This aims at tracing the data used in CoT and addressing its potential faithfulness problem by forcing the LLM to explicitly cite which input fragments it uses, without requiring model internals. Our evaluation in Section 4.3.1 shows that the LLMs reliably report these semantic markers as part of its CoT, and we can parse them as a trace.

For part two of our investigation, we need to understand if these CoT traces indeed represent parts of the input that were relevant to how the LLMs made their decisions. To that end, we conduct an experiment where our input to the LLM is a factual question with a ground truth. We extract the semantic markers from the output, and gradually remove parts of the original input. We show that removing parts of the input that are referenced in the traces of semantic markers indeed changes the output of the LLM compared to our ground truth (i.e., not just different wording of the answer, but a different conclusion compared to our ground truth). Conversely, we also confirm that removing parts of the input that are

not mentioned in the CoT trace does not change the output.

With the results of tasks one and two, we have a mechanism that allows us to extract traces of semantic markers from an LLM output. With such mechanism, we can apply delta debugging to the LLM prompts. Depending on where we are in the lifecycle of developing our LLM prompts, we envision different delta debugging applications. In the early stages of system development, we are not confident that our prompt instructions are factually correct and the system typically exhibits clear and deterministic failures caused by factual errors in the prompts. In this case, we will work with an oracle (e.g., a human expert or established ground truth) to determine if the LLM output yields the correct answer. If the output is deemed incorrect, we can first reduce the size of the LLM input by stripping off everything that was not mentioned in the trace of semantic markers, and then we conduct delta debugging via binary partitioning to identify the minimal subset of markers that, when removed, change the output from an incorrect answer to a correct or inconclusive answer.

As a second application of delta debugging for LLM prompts, we consider the case that in later stages of system development, the prompt is already mature and in pre-production but the system sometimes has subtle, non-deterministic issues caused by ambiguous prompt instructions. In this use case, we treat LLMs like probabilistic oracles, and we send the same input multiple times to observe occasional outliers if any. When there are outliers, we associate the outlier results with semantic markers by suspiciousness scoring delta debugging to identify the parts of the inputs that are responsible for inconsistent behaviors.

We present technical details of semantic markers in Section 4.2.1 and how we apply delta debugging with the aid of semantic markers in Section 4.2.2. We also present case studies in Section 4.3.2 where we follow the full development cycle of an AWS internal document processing system and demonstrate how delta debugging can be applied to effectively isolate a minimized region from the input that contains factual errors or poor instructions. In Section 4.3.3, we discuss that with the outcome of delta debugging, the system can be improved without human intervention.

In summary, this study makes the following contributions:

- **Semantic Markers for LLM Debugging and Explainability.** We introduce a novel technique for instrumenting and recovering unique markers from LLM outputs, enabling traceability from output elements back to specific input fragments.
- **Delta Debugging for LLM Inputs.** We adapt delta debugging to use semantic markers as a means of isolating minimal input fragments responsible for unexpected or incorrect outputs.
- **Experience Report on Industry Use Case.** We conduct experiments on a benchmark and an AWS internal LLM-backed document processing system to evaluate whether semantic markers can be reliably extracted, whether they accurately identify causal input fragments, and whether they enable systematic prompt debugging and improvement.

## 4.2 Approach

The goal of delta debugging is to find the minimal subset of the input responsible for the unexpected behaviors. In the scenario of LLM-integrated systems, unexpected behaviors include hallucinations or incorrect answers. In this section, we introduce our approach to debug LLM-integrated systems using a running example where we want to develop a service that can answer questions about AWS documentation. As developers, when we plan such a service, we would first get the relevant documentation text and put it into a text file, or a vector database (if we want to use RAG), and write some instruction prompts that put some general guardrails around how our service can answer, and which type of answers are out of scope. Then we would create some question-answer pairs of factual questions that we want to use for testing.

Assume now that while testing our service, one of our tests fails: When asked “After creating Lambda functions defined as container images, I need to use the Lambda API

to manage them. For this, do I need to create the function from the same account as the container registry in Amazon ECR?”, the system incorrectly responds “Yes” due to outdated information in the documentation context that we used for our service.

How do we debug this issue? And how do we identify this outdated part of the documentation in our input? There could be different reasons why the test failed: factual errors that cause deterministic failures, and ambiguous instructions that lead to inconsistent behavior. To enable systematic debugging of LLM inputs, we need a mechanism that provides visibility into which parts of the input context influence the model’s reasoning process. Traditional debugging approaches fail here because LLMs operate as black boxes, making it impossible to trace how specific input fragments contribute to the final output.

To address this challenge, our approach first injects unique semantic markers throughout the input context, then instructs the LLM to reference these markers during its chain-of-thought reasoning. This creates an explicit trace that maps each reasoning step to the input fragments that influenced it, providing the foundation for systematic delta debugging. We first describe how we use semantic markers to generate chain-of-thought traces in Section 4.2.1, then we demonstrate how to use such traces to run delta debugging in Section 4.2.2. We evaluate them in Section 4.3.

### 4.2.1 Trace Generation from Chain of Thought

Our approach generates chain-of-thought traces from the reasoning process by seeding semantic markers into the input context (in our running example, the AWS documentation and the instruction prompts) and amending the input with additional instructions that force the LLM to quote with these markers as references during its reasoning. The key insight is that by seeding unique, identifiable markers throughout the input, we can create a mapping between the LLM’s reasoning steps and specific portions of the original context, enabling precise traceability and explainability.

The trace generation process begins with semantic marker injection during input prepro-



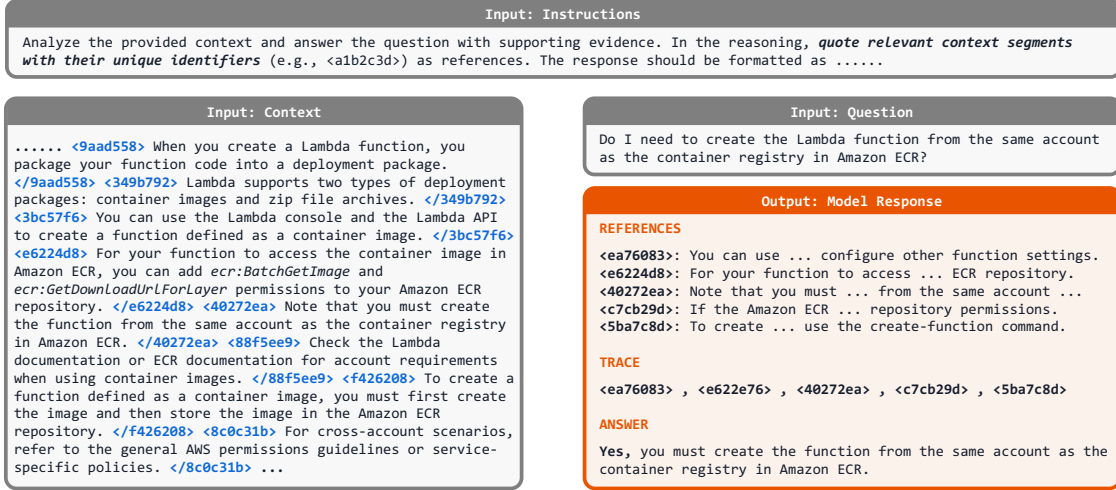


Figure 4.1: Semantic marker injection and chain-of-thought trace generation.

cessing. When processing documents such as PDFs or structured text, we parse the content using standard tools (e.g., PyMuPDF) to extract textual content. Then, each semantically meaningful unit (e.g., a sentence or a paragraph, depending on the context) is wrapped with a unique marker, preserving the original structure while adding traceability. As illustrated in Figure 4.1, after seeding semantic markers into the input (i.e., the AWS documentation and the instruction prompt), each sentence is wrapped with a unique identifier (e.g., <9aad558> ... </9aad558>). These semantic markers are generated by hashing the wrapped context elements, enabling efficient verification of whether the model accurately quotes the original text by simply re-hashing the quoted content. In practice, marker injection adds approximately 5% to 10% more input tokens, depending on the segmentation granularity.

Figure 4.1 shows how chain-of-thought traces are generated for our running example. Given the question “Do I need to create the Lambda function from the same account as the container registry in Amazon ECR?” and our input with seeded semantic markers, the model outputs its references, trace, and a final answer. The instruction prompt added by our approach explicitly instructs the model to quote relevant input segments using their unique identifiers in the reasoning process, generating a chain-of-thought trace (e.g., the highlighted trace <ea76083>, ..., <5ba7c8d> in the model output) that reflects the reasoning path. Rather than simply asking for conclusions, this prompting strategy requires the model to

justify each reasoning step by referencing specific marked segments from the input, which we refer to as *reference segments*. This creates a structured trace where each logical inference is anchored to identifiable document elements.

It is worth noting that in this example, the trace immediately provides debugging clues that were previously impossible to obtain due to the black-box nature of LLM systems. The trace reveals that segment <40272ea> containing the outdated “same account” requirement is referenced in the model’s reasoning, directly contributing to the incorrect answer. Without semantic markers, developers would have no visibility into which specific parts of the input influenced the model’s decision-making process. The resulting chain-of-thought trace can be used to map each reasoning step back to its source for post-validation, and enables systematic debugging of the decision-making process.

## 4.2.2 Delta Debugging with Semantic Markers

With the aid of the chain-of-thought traces generated through semantic markers, we propose a two-phase delta debugging methodology that systematically isolates minimal problematic input fragments. This approach handles both scenarios with and without ground truth through a unified framework that progresses from trace labelling to failure-inducing context isolation using delta debugging techniques.

### Labeling with Ground Truth

Delta debugging relies on ground truth that indicates whether a run is correct or incorrect. Given the input and output of a model run, we assign such labels using two complementary strategies, selected based on the availability and cost of ground-truth evaluation.

**Oracle-guided Labeling.** Oracle-guided labeling is used when ground truth can be reliably obtained. Ground truth may come from established references, from automated verification systems, or from human judgment when the test cases are manageable. Given a

question and LLM input with semantic markers, we record the model’s output and extract its chain-of-thought trace, then label the output as *correct* or *incorrect*. This approach ensures high label accuracy and is particularly effective during early development phases or for critical cases where precision is the top metric.

**Oracle-free Labeling.** Oracle-free labeling is used when ground truth is not available, such as in large-scale testing or when manual evaluation becomes impractical. In these cases, we leverage the model’s self-consistency by applying majority voting across multiple runs. We execute the same query multiple times (e.g.,  $N \geq 10$  runs) with identical context and question, collecting both final answers and reference segments from each run’s chain-of-thought trace. The most frequent answer is treated as the likely correct outcome, while minority answers are considered potential failures. This statistical approach mirrors established self-consistency techniques in LLM reasoning, where majority voting across multiple samples improves answer reliability. When vote distributions are ambiguous, additional runs are generated to establish clearer consensus, ensuring robust pseudo-ground truth for subsequent analysis.

The combination of oracle-guided and oracle-free labeling mirrors the evolution of real-world software development. Early in system development, developers can manually inspect a small number of runs to provide high-confidence labels and fine-grained feedback. As systems scale and queries grow more complex, manual evaluation becomes infeasible, and labeling naturally shifts toward automated oracle-free strategies and statistical analysis. This progression ensures effective debugging throughout the system lifecycle while balancing accuracy with scalability.

## Failure-Inducing Context Isolation

With labeled ground truth, we propose two delta debugging techniques to pinpoint the context segments that contribute to failures. These methods represent different reduction strategies within the delta debugging framework and can be used individually or in combi-

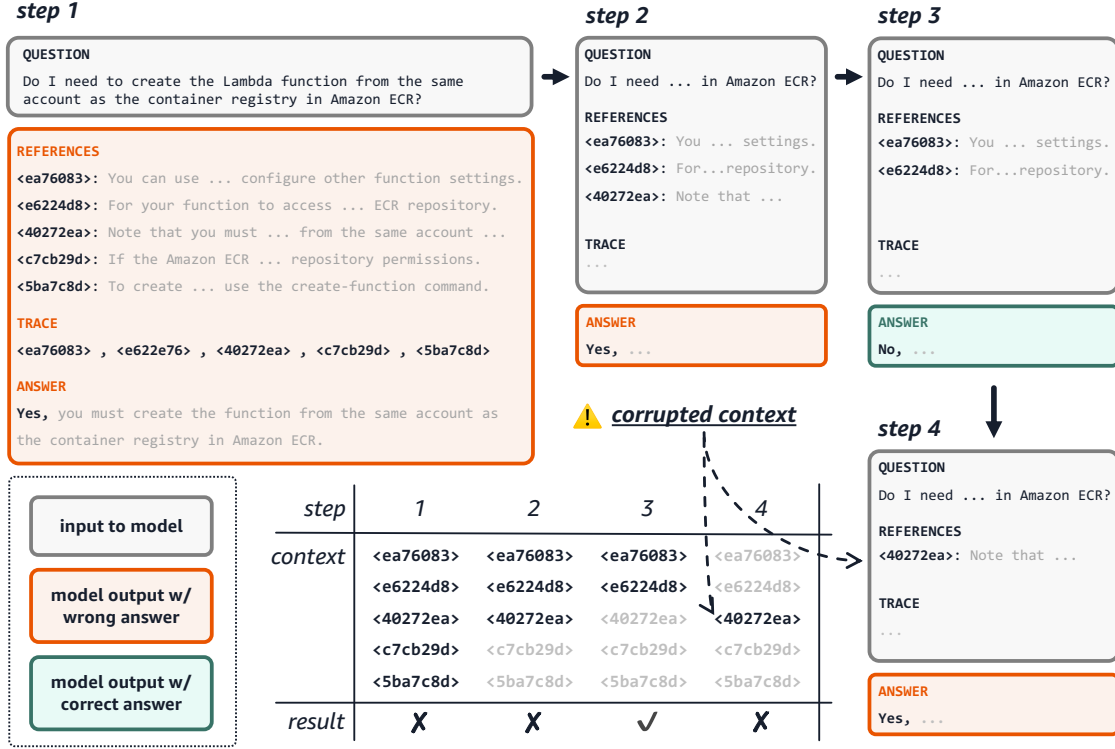


Figure 4.2: Delta debugging via binary partitioning.

nation depending on the specific debugging requirements.

**Binary Partitioning.** Binary partitioning represents the classical delta debugging approach, systematically testing subsets of context segments to identify minimal failure-inducing fragments through recursive partitioning. We adapt the ND3MIN algorithm, an extension of the traditional `ddmin` algorithm for handling non-determinism, to better align with the nature of LLM behaviors. For each subset test, we perform  $N$  repeated trials and require at least  $K$  failures out of  $N$  runs to classify a subset as failure-inducing. In this study, we set  $N = 10$  and  $K = 8$ . This threshold-based approach ensures statistical confidence in our outcomes, preventing false positives due to non-deterministic behaviors of LLMs.

To test subsets of context segments, we manipulate the chain of thought and guide the reasoning process by leveraging the semantic markers to constrain the model’s attention to a chosen subset of references. In the AWS documentation example, the initial chain-of-thought trace contains 5 reference segments: <ea76083>, <e6224d8>, <40272ea>, <c7cb29d>, <5ba7c8d>.

| <div>QUESTION</div> <div>Do I need to create the Lambda function from the same account as the container registry in Amazon ECR?</div> |     |     |    |     |     |    |     |     |    |     |                                               |
|---------------------------------------------------------------------------------------------------------------------------------------|-----|-----|----|-----|-----|----|-----|-----|----|-----|-----------------------------------------------|
| run                                                                                                                                   | 1   | 2   | 3  | 4   | 5   | 6  | 7   | 8   | 9  | 10  | score                                         |
| <2bf230a>                                                                                                                             | ●   | ●   | ○  | ●   | ●   | ○  | ●   | ●   | ○  | ●   | -1.00 ← - omission error vs. commission error |
| <eb27a8c>                                                                                                                             | ○   | ○   | ●  | ○   | ○   | ●  | ●   | ○   | ●  | ○   | +0.86                                         |
| <e187a92>                                                                                                                             | ●   | ●   | ○  | ●   | ○   | ○  | ●   | ●   | ○  | ○   | -0.71                                         |
| <7b9a5ce>                                                                                                                             | ○   | ○   | ●  | ○   | ○   | ●  | ○   | ○   | ●  | ○   | +1.00 ←                                       |
| <126c524>                                                                                                                             | ●   | ○   | ○  | ●   | ●   | ○  | ●   | ○   | ○  | ●   | -0.71                                         |
| <9cdcc69>                                                                                                                             | ●   | ●   | ○  | ●   | ●   | ○  | ●   | ●   | ○  | ○   | -0.86                                         |
| <0c4a4d9>                                                                                                                             | ○   | ○   | ●  | ●   | ○   | ○  | ○   | ○   | ●  | ○   | +0.52                                         |
| <d0a01aa>                                                                                                                             | ○   | ○   | ●  | ○   | ○   | ●  | ●   | ○   | ○  | ○   | +0.52                                         |
| answer                                                                                                                                | yes | yes | no | yes | yes | no | yes | yes | no | yes |                                               |

unique answer groups

● segment in trace

○ segment not in trace

Figure 4.3: Delta debugging via suspiciousness scoring.

<c7cb29d>, <5ba7c8d>. As illustrated in Figure 4.2, binary partitioning systematically tests subsets of these segments to isolate the failure-inducing fragment. For subset testing, we construct prompts with pre-filled partial responses containing only the subset of reference segments, allowing the model to complete the reasoning and give the final answer as a continuation. The algorithm recursively narrows down the search space, removing segments that do not contribute to the failure and further dividing those that do. This process continues until we identify a minimal subset where every segment is necessary for the failure. In our example, the process isolates segment <40272ea> containing the outdated “same account” requirement as the exact point from the input that is responsible for the faulty output. Removing this single segment causes the model to produce the correct answer, confirming its role as the failure-inducing segment.

**Suspiciousness Scoring.** This technique represents an alternative delta debugging strategy that leverages statistical analysis across multiple labeled traces to identify failure-inducing segments without explicit partitioning. Rather than systematically removing subsets as in classical delta debugging, this approach treats each context segment as a candidate for removal and uses correlation analysis to rank segments by their contribution to failures.

As illustrated in Figure 4.3, we aggregate traces from multiple runs, where ● indicates

segment presence and  $\circ$  indicates absence in each run’s chain-of-thought. In the AWS documentation example, when the context contains ambiguous instructions such as “Check the Lambda documentation or ECR documentation for account requirements”, the model produces inconsistent behavior across multiple runs. Some runs may reference segments about Lambda-specific requirements while others focus on general ECR permissions, leading to different conclusions. For each segment  $s$ , we define the suspiciousness score as:

$$score(s) = \frac{|R_f^s|}{|R_f|} - \frac{|R_p^s|}{|R_p|}$$

where  $R_f^s$  and  $R_p^s$  represent the subsets of failing and passing runs that reference segment  $s$ , respectively.

This scoring mechanism captures both commission and omission errors. Positive scores indicate segments that correlate with failure (commission errors), identifying problematic content that misleads the model when present. Strongly negative scores identify segments whose absence correlates with failure (omission errors), highlighting critical information that leads to incorrect outcomes when overlooked. For instance, segment `<7b9a5ce>` appearing only in failing runs yields  $score = 1$ , indicating strong failure correlation, while segment `<2bf230a>` present only in passing runs yields  $score = -1$ , suggesting its absence contributes to failure. The resulting scores produce a prioritized ranking of potentially failure-inducing inputs, directing developers toward a systematic and efficient investigation.

### 4.2.3 Development Lifecycle Integration

The example of AWS documentation demonstrates how our approach addresses different failure types throughout the software development lifecycle. Early development phases typically exhibit clear, deterministic failures caused by factual errors like the outdated “same account” requirement in segment `<40272ea>`, making them suitable for binary partitioning-based delta debugging. As systems mature, debugging challenges shift toward subtle, non-deterministic

issues caused by ambiguous instructions that create inconsistent behavior across multiple runs, requiring suspiciousness scoring for statistical pattern identification.

Both approaches leverage the same semantic marker infrastructure, creating a unified debugging framework that adapts to different failure characteristics. This complementary relationship transforms LLM debugging from ad-hoc manual inspection into a systematic methodology that scales with system complexity and development maturity.

## 4.3 Evaluation

We evaluate our approach by answering the following questions:

**RQ1 To what extent do generated chain-of-thought traces represent authentic**

**LLM reasoning?** Semantic marker-based traces must reflect genuine reasoning pathways to serve as reliable foundations of subsequent debugging or analysis. Establishing the authenticity and fidelity of these traces is essential before they can be trusted as reliable representations of LLM decision-making processes for debugging purposes.

**RQ2 How effective is delta debugging, based on the generated chain-of-thought**

**traces, in isolating input segments responsible for unexpected outputs?** Even if traces accurately represent reasoning processes, the practical utility of the approach depends on whether they can effectively guide the identification of problematic input segments. Demonstrating the effectiveness of trace-based delta debugging is crucial for validating the overall debuggability of LLM-based applications with our approach.

**RQ3 Can the outcome of trace-based delta debugging be used to systematically**

**improve the context quality?** Beyond fault localization, the ultimate value of our debugging approach lies in its ability to provide actionable insights for system improvement. This question evaluates whether the identified problematic input segments can guide systematic enhancements to context curation, input preprocessing, or retrieval mechanisms, thereby validating the practical impact of our approach.

**Benchmark.** We developed the *FamilyTree* benchmark, a controlled synthetic dataset that describes a customizable family across multiple generations. The family structure is described by only two direct relation types: *parent of* and *married to*, encoded as a formalized graph data structure where nodes represent family members and edges represent relationships.

The benchmark is accompanied by two automated engines: a context engine that generates natural language descriptions from the graph structure with precise definitions of a variety of relations, and a question engine that creates queries with controlled difficulty levels. The question engine generates questions requiring up to 5 hops of graph traversal, where each question automatically receives a difficulty score based on the number of hops (i.e., the number of facts required to be considered from the context for answering). This ranges from single-hop questions requiring direct relationship lookup (e.g., “who are the parents of Jack?”) to complex multi-hop questions requiring traversal of up to 5 edges (e.g., “who are the cousins of Jack?”).

This design fundamentally enables systematic evaluation with known ground truth, verifiable reasoning paths, and controllable complexity with the underlying graph data structure.

### 4.3.1 Fidelity of Chain-of-Thought Traces

To assess the reliability of our semantic marker-based approach, we evaluate two fundamental aspects: the accuracy of marker references and the authenticity of the generated chain-of-thought traces. These evaluations are essential to validate that our approach produces trustworthy debugging information.

#### Semantic Marker Reference Accuracy

We first investigate whether LLMs accurately reference the semantic markers when instructed to include them in their reasoning process. A key design feature of our semantic markers is their verifiability: each marker is generated by hashing the wrapped content, enabling



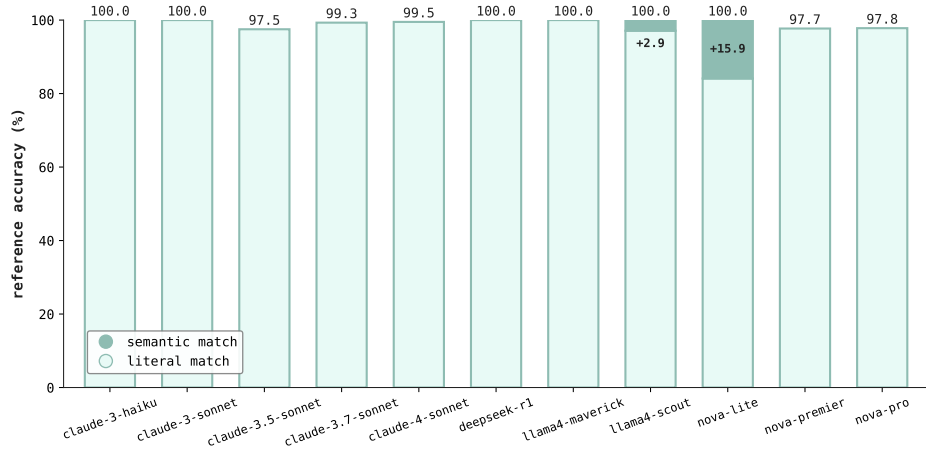


Figure 4.4: Reference accuracy of semantic markers.

efficient validation by re-hashing quoted segments and comparing them with the original marker strings.

We conducted experiments across 11 different models, with each model answering 12 questions from the FamilyTree benchmark, with each question repeated 10 times. This generated a total of 1,320 traces with marker-segment pairs across all responses, where each pair represents a semantic marker referenced by the model and its corresponding quoted text segment. For each marker-segment pair, we performed two types of validation: *literal matching*, which leverages our hash-based marker design by re-hashing the quoted text and comparing it with the original marker to verify exact character-by-character correspondence, and *semantic matching*, which uses a judge LLM to evaluate whether minor variations preserve the original meaning. Figure 4.4 presents the reference accuracy results across all evaluated models.

The results demonstrate near-perfect semantic matching accuracy (100%) for almost all models, indicating that LLMs reliably reference the correct content when instructed to use semantic markers. Some discrepancies between literal and semantic matching occur primarily with definitional content, where models may omit redundant prefixes while preserving the core meaning. For example, when the original text states “child-in-law: A is a child-in-law of B if A is the spouse of B’s child,” the model only quotes “A is a child-in-law of B if A is the

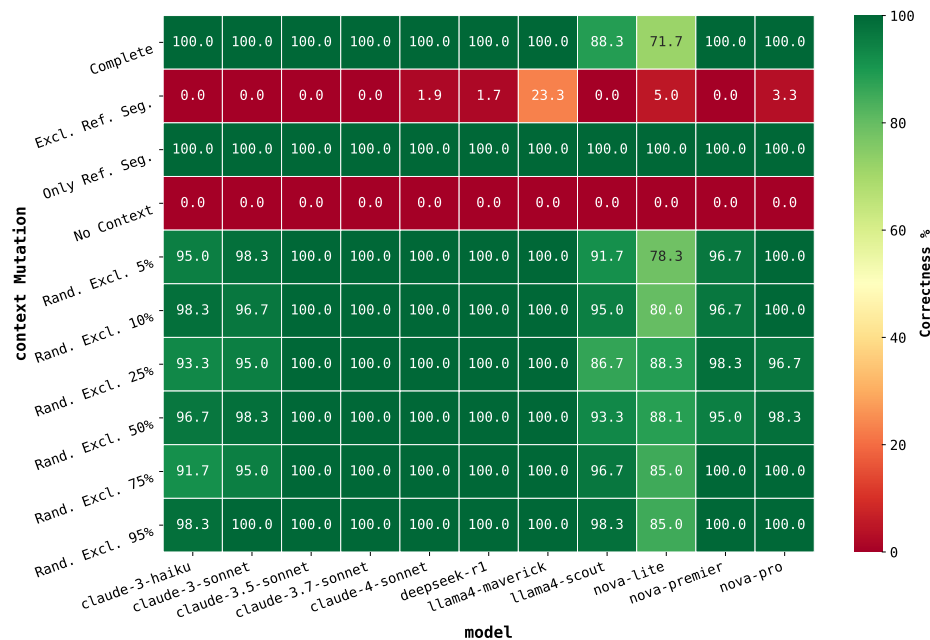


Figure 4.5: Mutation testing results for chain-of-thought trace authenticity validation.

spouse of B’s child,” omitting the definitional prefix while maintaining semantic accuracy.

## Chain-of-Thought Trace Authenticity

After verifying that models correctly reference individual markers, we must ensure that the complete chain-of-thought traces authentically represent the model’s reasoning process. To validate this, we designed a mutation testing experiment that systematically mutates the context provided to the model and observes the impact on answer correctness.

Our hypothesis is that if the generated traces authentically reflect reasoning dependencies, then: (1) removing segments referenced in the trace should impair the model’s ability to answer correctly, (2) providing only the traced segments should be sufficient for correct answers, (3) randomly removing unrelated segments should not affect performance as long as traced segments remain available, and (4) providing no context should result in poor performance, confirming that the model relies on the provided context rather than its own prior knowledge to answer the questions.

To test this hypothesis, we generated a set of multi-hop questions using our FamilyTree

benchmark, where answering each question requires the model to examine and reference multiple segments from the provided context. We mutate contexts that directly correspond to our hypothesis: we preserved the full context as a control condition, excluded segments referenced in the trace to test hypothesis (1), provided only the traced segments to test hypothesis (2), randomly excluded varying percentages (5%, 10%, 25%, 50%, 75%, 95%) of unrelated context while preserving traced segments to test hypothesis (3), and provided no context at all to test hypothesis (4). To ensure statistical robustness, we ran each question under each context mutation 10 times for each model and averaged the results.

Figure 4.5 presents the mutation testing results as a heatmap, where each cell represents the answer correctness under a specific context mutation for each model. Despite some minor deviations, the results strongly support our hypothesis: models consistently fail when traced segments are removed, succeed when only traced segments are provided, maintain performance when unrelated segments are removed, and show poor performance without any context, confirming their dependence on the provided information. The only outlier is llama4-maverick, which still successfully answered 23.3% of questions when the reference segments were removed. Our manual investigation found that llama4-maverick was able to find another, albeit longer, reasoning path in the family tree to reach the correct answer. This pattern confirms that the semantic marker-based traces accurately capture the reasoning dependencies required for correct answers, validating the authenticity of our trace generation approach.

### 4.3.2 Effectiveness of Delta Debugging based on Semantic Markers

To evaluate the effectiveness of our semantic marker-based delta debugging, we conducted two complementary case studies: a controlled laboratory experiment and a real-world industrial system evaluation.

## Lab-Controlled Case: FamilyTree Benchmark

We first validated our approach in a controlled setting using the FamilyTree benchmark to demonstrate systematic failure isolation under known conditions. We constructed a 42-segment context containing 16 family relation definitions and 26 direct relationships between family members, then introduced two types of failures to evaluate our debugging methodology.

In the first scenario, we planted factual errors in the relationship data while asking multi-hop questions that indirectly depended on the corrupted information. The LLM generated an initial chain-of-thought trace of 13 segments when producing incorrect answers. Binary partitioning-based delta debugging systematically reduced this to a 1-minimal set of 7 segments, where removing any single segment made the incorrect answer unreproducible. Critically, the minimal set correctly identified the planted erroneous facts despite the questions not directly querying the corrupted information, demonstrating the approach’s ability to trace indirect dependencies.

In the second scenario, we introduced ambiguous family relation definitions and applied oracle-free labeling across 10 runs without ground truth. The LLM produced inconsistent answers due to the poorly defined relations. Suspiciousness scoring analysis clearly identified the ambiguous definition segments as having the highest correlation with failures, with these segments appearing disproportionately in incorrect runs compared to correct ones.

Both scenarios validate that delta debugging reliably isolates failure-inducing fragments under controlled conditions, providing systematic evidence that the approach can distinguish between relevant and irrelevant input segments regardless of whether ground truth is available.

## Real-World Use Case: AWS Internal Document Processing System

We applied our approach at AWS to debug an internal service that is being developed for automating document auditing. The system is designed to cross-validate that contractual

documents are operationalized correctly in JSON configuration files. Since this is an internal system, we need to omit the exact nature of these contractual documents, but we will outline the process about how they are being used. A contract, once signed by all parties, is passed to an analyst (a domain expert with the relevant legal knowledge to understand the intent of the contract). The analyst then uses a purpose-built internal website to enter the terms of the contract one by one. The terms are then stored in JSON configuration files and later used to process the customer’s data. Contracts include certain identifiers, such as customer or product IDs, but also legal text that is not easily parsable by an automated system.

Our LLM-integrated auditing service integrates into this process to validate that the analyst accurately entered all contract terms into the system. To that end, our service takes three input components: (1) detailed instructions on how to conduct the audit with rules and special case handling, (2) contract text extracted from PDF documents, and (3) JSON-formatted data that was entered by the human analyst.

The instructions state that the LLM should return one of three values for each contractual term that it finds: “MATCH”, if the contract and the JSON both contain the term and agree on the details; “DIFFERENCE” if both contain the term, but disagree on the meaning, or “MISSING” if only one of both contains the term.

Our evaluation demonstrates how the two delta debugging approaches complement each other across different phases of system development, addressing distinct failure patterns that emerge as the system matures.

**Binary Partitioning for Clear Failures during Early Development.** During initial development phases, engineers used dummy contracts to test system performance and encountered a persistent deterministic failure. When processing test contracts, the system would incorrectly flag account IDs as “DIFFERENCE” despite manual verification confirming the account identifiers were indeed identical between the contract text and JSON data. This failure occurred consistently across multiple test cases, making it a clear blocker for system development. The manifestation was straightforward: given input containing matching

account information, the system would systematically produce incorrect mismatch reports, preventing progression to more complex testing scenarios.

The failing input contained 502 marked paragraphs of the input as relevant, encompassing the instruction prompt, contract text, and JSON data. The extensive length of the chain-of-thought trace made manual inspection impractical, as engineers could not efficiently identify which specific segments contributed to the incorrect behavior. When producing the incorrect “DIFFERENCE” output, the LLM generated a chain-of-thought trace referencing 21 segments consisting of instruction segments, contract text snippets, and JSON data excerpts.

We applied binary partitioning-based delta debugging to systematically isolate the failure-inducing fragments. The algorithm recursively partitioned the 21 referenced segments, testing each subset to determine whether it still produced the incorrect output. Through 28 systematic runs, the process eliminated segments that did not contribute to the failure while preserving those that did, ultimately converging on a minimal set of 3 segments: a paragraph from the instructions, one contract text snippet containing account information, and one corresponding data excerpt.

The isolated minimal set revealed the root cause: the instruction paragraph specified “12-digit account ID should strictly match” without accounting for common formatting variations such as hyphenated representations (e.g., “1111-2222-3333” vs “111122223333”). Armed with this precise diagnosis, developers corrected the instruction to accommodate formatting variations and added special case handling examples for hyphenated formats. After these targeted fixes, the account ID matching error was completely resolved, enabling the development team to progress to testing more complex contract scenarios.

**Suspiciousness Scoring for Subtle Failures when System Matures.** As the system matured and obvious deterministic failures were eliminated through iterative testing and fixing, engineers encountered a more subtle issue with contractual commitment terms, which are terms in the contract that specify a list of pairs of dates and monetary values.

The system would sometimes produce correct commitment values and sometimes fail, creating an inconsistent and unreliable behavior pattern. Unlike the previous clear failure, this issue was non-deterministic and difficult to reproduce consistently. Manual inspection revealed that the system would correctly extract commitment values from some test contracts while producing incorrect values for seemingly similar contracts, with no obvious pattern distinguishing successful from failed cases.

Since ground truth was not immediately available for all test cases and the failure was non-deterministic, we applied oracle-free labeling with suspiciousness scoring. We executed the same problematic input 10 times, collecting both final answers and chain-of-thought traces from each run. The multiple executions produced three distinct answer groups with different commitment values. Using majority voting, we identified the most frequent answer as the likely correct outcome, treating the minority answers as potential failures.

Suspiciousness scoring analysis computed correlation scores for each segment based on its appearance frequency in failing versus passing runs. The analysis revealed high commission error correlation for two specific segments: an instruction sentence stating “Locate commitment section in the main contract or in the contract amendments” and a contract segment containing actual spend commitment information from the main contract body. These segments appeared disproportionately in runs that produced minority (incorrect) answers compared to majority (correct) answers.

This statistical analysis provided the crucial insight that the model was encountering ambiguity when spend commitments appeared in both main contracts and amendments simultaneously. The instruction failed to specify precedence rules for such cases, leading to inconsistent model behavior depending on which information the model prioritized during reasoning. Understanding that the correct business logic requires prioritizing amendment information as it overrides main contract terms, developers added explicit precedence instructions and included special case examples demonstrating proper handling of conflicting spend commitment information. After these targeted improvements, the system achieved

stable and consistent spend commitment extraction across all test scenarios.

**Final Remarks.** The development of our internal Document auditing system demonstrates how binary partitioning and suspiciousness scoring address different debugging needs throughout the system development lifecycle. Binary partitioning proved essential during early development for isolating precise root causes of deterministic failures with clear manifestations and available ground truth. As the system matured and obvious bugs were resolved, suspiciousness scoring became valuable for identifying subtle patterns in non-deterministic behaviors where manual evaluation was impractical, and ground truth was not immediately available. Both approaches led to concrete improvements, like corrected rule descriptions and enhanced special case handling, that systematically improved system reliability and transformed debugging from manual inspection to methodical fault isolation.

### 4.3.3 Delta Debugging-Guided Improvement

The ultimate goal of debugging is not merely to locate the fault but to enable systematic improvement of the target systems. With the delta debugging outcomes from Section 4.3.2, we have seen that our approach can identify relevant LLM input segments; now we want to inspect if we can further use these segments for automatic repair of the input. We show how a tool like Amazon Q Developer CLI, an agentic LLM command tool used by developers at AWS, can be used to process the outputs of our delta debugging from the previous section to generate patches.

**Automated Factual Error Correction.** For the contract auditing system described in Section 4.3.2, the binary partitioning process had isolated a segment containing the faulty instruction about account ID formats, as well as two segments containing actual account ID data. Amazon Q Developer CLI received delta debugging outcomes from RQ2: (1) the full original input context, (2) the minimized result containing the 3 critical segments identified by binary partitioning, and (3) the incorrect “DIFFERENCE” output demonstrating the factual error. Without providing instructions or descriptions specific to the faulty segments



in this case, Amazon Q Developer CLI successfully recognized that the 3 segments constitute the minimal set that causes unexpected behavior, and identified that the instruction segment contains an overly restrictive comparison rule causing semantic equivalence to be incorrectly flagged as differences.

Amazon Q Developer CLI automatically generated a patch that corrected the logic from strict formatting to semantic equivalence. The fix required only two lines of modification of the instruction bit of the LLM input, totaling 36 tokens from the original input of 29,516 tokens, representing a modification rate of 0.12% while directly addressing the root cause identified by delta debugging.

The generated patch exhibited optimal characteristics: surgical precision with minimal modifications targeting only the problematic segment identified, semantic correctness that preserved audit functionality while fixing the logic error, and causal alignment that directly addressed the fault segment. The automated approach generated a minimal corrective patch without human intervention, demonstrating that delta debugging outcomes provide actionable insights for systematic system improvement and self-healing.

**Automated Instruction Enhancement.** We next applied the same improvement methodology to the commitment value inconsistencies identified in mature development scenario discussed in Section 4.3.2. The suspiciousness scoring had identified ambiguous instruction segments causing non-deterministic behavior in spend commitment extraction. We used these results to demonstrate automated prompt instruction enhancement.

Without any fixes, during the 10 runs of the same input with the same model, the model’s output could be divided into three groups with 3 different traces, where the most common trace appeared in 6 out of 10 runs. The suspiciousness scoring results had identified three highly influential segments contributing to reasoning inconsistency. The highest suspiciousness score for a segment is 0.70, indicating insufficient guidance for consistent decision-making.

For automated instruction enhancement, Amazon Q Developer CLI received (1) the orig-

inal input and (2) the segments with the top 5 suspiciousness scores. During its analysis, Amazon Q Developer CLI recognized that the instructions for handling spend commitments from main contracts versus amendments were ambiguous and proposed a patch (with 37 tokens) that elaborates the instructions with "When amendments modify the original contract terms, the amended values take precedence over the original contract values."

Post-improvement validation demonstrated significant stability enhancement. Running the same 10 experiments with the improved prompt achieved perfect trace consistency: all 10 runs produced identical execution traces and reached 100% output accuracy. The number of unique traces reduced from 3 to 1, and outlier runs were completely eliminated, confirming that the targeted improvements successfully resolved the underlying reasoning inconsistencies identified.

**Final Remarks.** Both examples demonstrate that delta debugging outcomes enable systematic system improvements through automated analysis and targeted modifications. The factual error correction leveraged binary partitioning results to achieve targeted fixes with only 0.12% modification of the input context. The prompt instruction enhancement used suspiciousness scores to eliminate reasoning inconsistencies entirely, reducing trace variance from 3 unique patterns to 1 consistent pattern across all runs. The patches proposed by Amazon Q Developer CLI are highly aligned with the developers' patches discussed in Section 4.3.2, except that the Amazon Q Developer CLI patches do not contain additional examples as it was not explicitly instructed to do so.

We are not able to collect or share representative data on how developers use delta debugging internally, we have anecdotal evidence that the experience aligns with what we discussed in this section. A promising observation that we leave for future work is that delta debugging integrated very well with our agentic developer tool Amazon Q Developer CLI. It is worth exploring if it is possible to provide generic delta debugging tools as Model-Context Protocol (MCP) servers to assist developers in other debugging tasks.

## 4.4 Related Work on Debugging LLM-Integrated Systems

Since its introduction 25 years ago [416, 418], delta debugging has been applied across various settings to isolate failure root causes by comparing passing and failing executions. Variants differ by the artifacts under reduction (inputs, execution traces, logs) and how many executions they require (e.g., automated reasoning based fault localization that requires only a single example, vs machine learning based approaches that need large quantities of logs), but share the goal of narrowing developer attention to the root cause of a failure.

**Debugging by instrumenting inputs.** Our approach instruments LLM context with semantic markers to enable principled debugging. Prior work has applied instrumentation and capturing tools for debugging: Beschastnikh et al. [88] enhance distributed-system logs to recover *happens-before* sequences of events for fault localization, JINSI [275] instruments Java applications to enable capture and replay of component-based systems for delta debugging, and others use capture-replay tools [298]. Our lightweight semantic markers specifically support ablations of knowledge fragments during debugging.

**Delta debugging across artifacts.** Beyond input values [416], delta debugging has been applied to diverse artifacts. Clapp et al. [116] propose a variant of delta debugging to minimize GUI event traces with the ability of handling non-determinism. Park et al. [280] present FALCON for debugging concurrent systems by monitoring data access patterns between passing and failing executions and Gong et al. [155] propose SNOWBOARD that localizes concurrency bugs in OS kernels by identifying shared memory usage between executions of generated tests. The CHESS tool by Musuvathi et al. [259] detects and diagnoses Heisenbugs [156] in multithreaded programs by taking control of the thread scheduler and to generate passing and failing interleavings for fault localization. Similarly, Zhou et al. [454] introduce an approach for debugging micro-service architectures through a dedicated

delta debugging controller. Our work adopts this *artifact-agnostic* approach, targeting *LLM prompting context and reasoning traces*.

**Refinements of delta debugging.** Delta debugging has been refined through semantic [294] and syntax awareness [337] to maintain input validity, hierarchical approaches [250] leveraging input structure, and probabilistic methods [361, 426] for prioritized testing. Our semantic markers similarly inject structure enabling fine-grained, semantics-aware reductions over long prompts.

LLMs introduce distinctive challenges: they expose probabilistic behavior, long contexts, and opaque internals, which makes direct application of traditional program-based fault localization challenging. Consequently, recent work emphasizes *explainability*, *input-output attribution*, and *prompt-level* analyses to understand and control model behavior.

**LLM-centric debugging.** Self-Debugging teaches an LLM to iteratively inspect and fix its own generated code using execution results and natural-language explanations [109]. Reflexion equips agents with verbal self-reflection and episodic memory to learn from prior failures across tasks [320]. *LDB* (LLM Debugger) segments programs into blocks and tracks the program state after each block throughout runtime execution [445]. *ChatDBG* provides a chat interface on top of traditional debuggers for programmers [212]. Compared to these, our approach is task-agnostic (beyond code), operates purely in the *prompt/CoT space*, and validates cited evidence with *delta debugging* to test causal necessity/sufficiency.

**Explainability and interpretability.** Surveys synthesize emerging techniques for LLM explainability and their limitations for reliable debugging [235, 438]. Gradient-based attribution [339] requires access to model internals, and model-agnostic approaches such as KernelSHAP [234] avoid internals but can be expensive for long inputs. Chain-of-thought (CoT) prompting [375] elicits step-by-step rationales, yet CoT explanations can be unfaithful to the true decision process [355], limiting their standalone value for debugging.

**Input-output attribution for LLMs.** Zhou et al. [453] score token influence for explanations in low-resource settings; Phukan et al. [287] analyze hidden-state similarities to identify which prompt segments are copied in contextual QA; TracLLM [372] proposes an informed search with denoising/ensembling for long-context attribution. Practical tools such as PIZZA [44] also adopt perturbation-based attribution. These methods help identify salient tokens or spans but stop short of testing minimal failure-inducing subsets.

**Comparison to our approach.** Our method combines CoT-style reasoning with input instrumentation and delta debugging. Semantic markers turn each context fragment into an addressable unit and *force* explicit citation during reasoning; the ensuing ablation tests provide a causal check on those citations. Compared to pure CoT, we verify faithfulness by testing whether cited fragments are necessary/sufficient for the answer.

## 4.5 Evaluating Large Language Model Tools on CI Failures

Large Language Models (LLMs) have revolutionized software engineering (SE) by enabling the automation of complex tasks such as fault detection, test generation, and automated code repair. LLMs and LLM-based tools have transformed how researchers and developers approach SE tasks, allowing them to tackle challenges traditional models struggle with. These tools are designed in various fashions, ranging from fixed, pre-defined steps [387] to agentic behaviors involving multi-step reasoning and planning [336].

However, evaluating the performance of LLMs and LLM-based tools on SE tasks remains a significant challenge. Existing benchmarks like SWE-Bench [187] and its extensions [415] primarily focus on GitHub issues, which do not fully capture the complexities of real-world software development processes. Additionally, these benchmarks do not automatically evolve to include new data, leading to potential data leakage when evaluating LLMs with a later

cutoff date than the benchmark release. Furthermore, these benchmarks lack sufficient details to fully replicate the original software failures, which limits their utility for reproducibility studies.

We introduce CI-BENCH, a unified framework that enables the evaluation of various LLM-based program repair tools to address the limitations of existing benchmarks. To harness the rich context from the software build process, CI-BENCH retrieves software build failures from the BUGSWARM [351, 456] dataset, comprising more than 3,500 reproducible CI/CD build failures containing complete context, such as the execution environment and logs that capture build failures (e.g., compilation errors, test failures, linting violations, misconfigurations, and environment-related issues), along with error messages, exceptions, and tracebacks. Such data is crucial for evaluating LLM-based program repair tools in real-world development and deployment scenarios. Furthermore, CI-BENCH also leverages the evolving nature of BUGSWARM to ensure evaluations include the latest data after an LLM’s cutoff date. Lastly, CI-BENCH reconstructs the original execution environment of each build failure, ensuring that evaluations faithfully mirror the real-world scenarios encountered by developers.

Given a targeted LLM-based program repair tool and a CI/CD build failure from BUGSWARM, CI-BENCH parses the build logs to extract information related to the failure. This information is then used to construct prompts for the targeted program repair tool. Additionally, using environmental information from BUGSWARM, CI-BENCH sets up a containerized environment that mirrors the original build process, allowing the tool under evaluation to run in the identical environment as the original build failure. With this setup, CI-BENCH replays the CI/CD build process and automatically determines whether the targeted tool successfully fixes the failure. To demonstrate the utility of CI-BENCH, we integrate three state-of-the-art LLM-based program repair tools: AGENTLESS [387], SWE-AGENT [403], and AUTOCODEROVER [434], with various underlying foundation models. However, CI-BENCH is not limited to these; it can be extended to support any program repair tool that

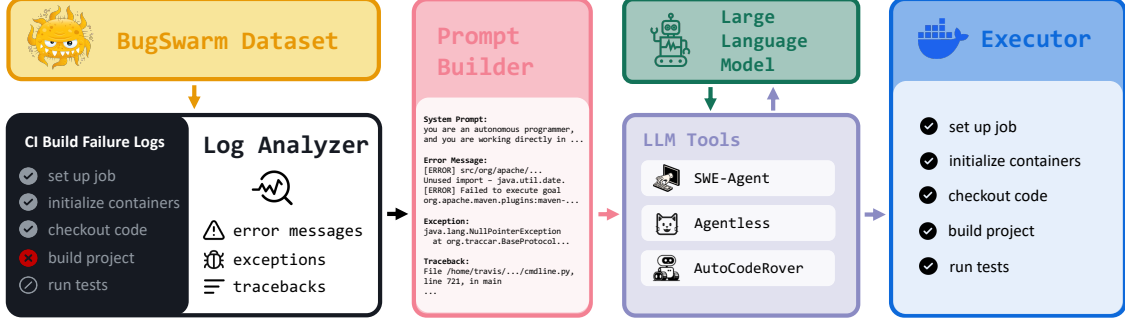


Figure 4.6: Components of CI-BENCH.

can process build failures.

We evaluate the three integrated LLM-based program repair tools (AGENTLESS, SWE-AGENT, and AUTOCODEROVER) on 100 real-world Java build failures from the BUGSWARM dataset (63 projects), measuring performance via plausibility and syntactic equivalence. For each tool, we assess the effectiveness of several foundation models, specifically GPT-4o, Claude-3.5-Sonnet, and Deepseek-V3. Our results demonstrate that AGENTLESS and SWE-AGENT significantly outperform AUTOCODEROVER, achieving average plausibility success rates of 28% and 30% respectively, compared to 11.3% for AUTOCODEROVER; peak performance for these tools reached 32%, 36%, and 13%. Manual verification against developer-provided fixes finds that AGENTLESS, SWE-AGENT, and AUTOCODEROVER produce on average correct patches in 25.3%, 28.3%, and 11.3% of cases, respectively. Notably, while AUTOCODEROVER had a lower success rate, all of its generated plausible patches were validated as correct.

With CI-BENCH, researchers and developers can more easily evaluate LLM-based program repair tools using any subset of CI/CD build failures from the BUGSWARM dataset and obtain performance metrics. CI-BENCH’s extensible design allows users to integrate new tools, models, and performance metrics seamlessly. The remainder of this section presents the technical details of the framework and a more detailed discussion of our evaluation results.

## 4.6 Overview of CI-Bench

CI-BENCH consists of several key components as shown in Figure 4.6. Our framework leverages BUGSWARM, as it remains the largest and most diverse dataset of reproducible CI failures available to date. The *Log Analyzer* parses the historical CI failure logs provided by BUGSWARM to identify key diagnostic information, such as error messages. The *Prompt Builder* transforms the extracted information into a generic prompt for the LLM-based program repair tools. These tools then incorporate this prompt into their system prompts and interact with the LLM to generate candidate patches. Finally, the *Executor* validates each patch by applying it to the failing program using the original execution environment, which is facilitated by BUGSWARM via Docker containers and the necessary build scripts. We further describe each component along with a running example using the BUGSWARM artifact `tananaev-traccar-64783123`.

**The BugSwarm Dataset** CI-BENCH enables LLM-based program repair tools to be evaluated on existing CI datasets. In this section, we select BUGSWARM, which is an established and maintained dataset built by mining consecutive failed-pass job pairs from two popular CI/CD services, GitHub Actions and Travis-CI. Each BUGSWARM artifact consists of a Docker image that includes the source code to reproduce the failure and the corresponding fix, scripts to build the code, and historical build logs from the original CI runs. A BUGSWARM artifact also encapsulates critical metadata such as the failed and passed commits and information about the CI system. Therefore, if artifacts from sources other than BUGSWARM are to be used, they must provide equivalent metadata so that the framework can seamlessly evaluate tools against them. The BUGSWARM dataset supports both Java and Python. Currently, it has 3,598 active artifacts; 2,245 are Java artifacts, and the rest are Python. In the rest of this section we will consider the BUGSWARM artifact `tananaev-traccar-64783123`. This artifact is mined from a Java GitHub repository named `traccar`, where a Null Pointer Exception occurs. We use this artifact to showcase our evaluation of



the SWE-AGENT tool.

**Log Analyzer** CI-BENCH’s first component is the *Log Analyzer*, which parses the failed build log of the artifact to extract error messages, exceptions, and tracebacks. This information is important because it usually includes the location of the failure such as file and line number, class name and method name. For our example, artifact `tananaev-traccar-64783123`, the Log Analyzer processes the failed build log and extracts various error messages, the Null Pointer Exception, and its tracebacks that indicate the location of the failure. This information serves as input to the *Prompt Builder*.

**Prompt Builder** The *Prompt Builder* creates a *generic prompt* from the error messages, exceptions, and/or tracebacks provided by the Log Analyzer to guide LLMs in code repair tasks. This prompt is the same for all the considered tools, and it also includes metadata such as the artifact name, its programming language, and its CI build system, which are essential for guiding the execution pipelines of certain tools. By design, the generic prompt remains simple yet effective, functioning as a structured problem description which each tool tries to repair. Each repair tool has its own *system prompt*: a predefined instruction that sets the overall task and its behavior. The generic prompt is inserted within each tool’s system prompt. The tools are originally designed to fix bugs using bug reports, not build logs, thus the need for our own *Prompt Builder*.

**Integrated LLM-Based Repair Tools** CI-BENCH is a unified framework that can plug in specific LLM program repair tools for benchmarking. To use the CI-BENCH framework, tools must meet the following requirements: (1) provide support for Java code repair, (2) be capable of processing sequential text input (required to process CI/CD build logs), and (3) generate patches. Currently, CI-BENCH has integrated three state-of-the-art LLM-based program repair tools, AGENTLESS [387], SWE-AGENT [403], and AUTOCODEROVER [434]. While these tools were originally designed for Python, we adapted them for Java programs.

```

protected Object decode( /* ... */ ) {
    /* ... */
} else if (type == MSG_LOGIN) {
-   if (channel == null) {
+   if (channel != null) {
        /* ... */
        response.writeByte(0x0A);
        channel.write(response, remoteAddress);
    }
    /* ... */
}

```

(a) Patch written by the developer.

```

protected Object decode( /* ... */ ) {
    /* ... */
} else if (type == MSG_LOGIN) {
    if (channel == null) {
        /* ... */
        response.writeByte(0x0A);
-       channel.write(response, remoteAddress);
+       if (channel != null) {
+           channel.write(response, remoteAddress);
+       }
    }
    /* ... */
}

```

(b) Patch generated by SWE-AGENT.

Figure 4.7: Patches for `tananaev-traccar-64783123`.

We also modified the tools to interact with BUGSWARM containers instead of relying on GitHub issue threads or source repositories. For example, we adapted the parsing and filtering logic to handle Java files and functions, and modified prompt templates to reflect Java syntax.

**Executor** The *Executor* validates the patches generated by the program repair tools. It accepts a patch file as input and applies it within a Docker container specifically configured with the artifact’s original dependencies. The *Executor* then replicates the build process by executing the exact commands from the original failed GitHub Actions job—typically involving a full project build and the execution of associated unit tests. A patch is deemed plausible if the repository builds successfully and all commands execute without error. In our running example, the patch generated by SWE-AGENT (Figure 4.7b) results in a successful build that passes all tests, and thus is considered plausible. However, this patch is not syntactically equivalent to the ground-truth fix provided by the developer (Figure 4.7a). We provide a detailed analysis of tool performance across various metrics in the next section.

## 4.7 Experimental Evaluation

This evaluation aims to determine how the LLM-based program repair tools perform on the CI failures. We vary the foundation models within the LLM-based program repair tools and assess the performance of each tool using different correctness metrics. To the best of

our knowledge, no prior work evaluates LLM-based repair tools on CI-failure data; thus, we report cross-tool and cross-model comparisons within CI-BENCH as reference points.

**Artifact and Model Selection** To select BUGSWARM artifacts for this evaluation, we adopt the issue selection criteria established by SWE-bench Lite [56]: (1) patches that do not include images, external hyperlinks, or references to specific commit SHAs and references to other pull requests or issues; (2) patches that contain one file edit; (3) patches with three or fewer edit hunks; (4) patches that do not create or remove files; (5) patches that are not for test files. From those artifacts, we selected 100 from 63 projects. This selection prioritized real-world patches that involved the smallest code changes, followed by those with the fewest edit hunks. This choice ensures that the evaluation focuses on patches with minimal edits, enabling a clearer assessment of the tool’s ability to repair issues that human developers typically resolve with small, localized modifications to the source code. However, CI-BENCH is not constrained by the nature and/or complexity of the required fixes. CI-BENCH supports evaluating tools on the full BUGSWARM corpus, though we selected only 100 artifacts here for a tractable evaluation. We chose three foundation models that have recently performed better on code generation tasks: GPT-4o, Claude-3.5-Sonnet, and Deepseek-V3.

**Correctness Metrics** We use two metrics to measure correctness: plausibility (PL) and syntactic equivalence (SYE). Plausibility is the measure of patches that result in a successful build of a BUGSWARM artifact. Syntactic equivalence refers to the token-level equivalence between the generated patch and the developer patch. CI-BENCH applies the generated patch and uses ANTLR [19] to tokenize the patched source for comparison with the developer-patched file. Furthermore, we additionally performed a manual inspection comparing the suggested patches against developer-provided patches.

**Evaluation Results** As shown in Table 4.1, we observe that all three tools are able to generate patches with all three foundation models. AGENTLESS and SWE-AGENT generate

Table 4.1: Results for 100 artifacts (#PG: patches generated, #PL: plausible, #SYE: syntactic equivalent, #CO: correct).

| Model                    | AGENTLESS |     |      |     | SWE-AGENT |     |      |     | AUTOCODEROVER |     |      |     |
|--------------------------|-----------|-----|------|-----|-----------|-----|------|-----|---------------|-----|------|-----|
|                          | #PG       | #PL | #SYE | #CO | #PG       | #PL | #SYE | #CO | #PG           | #PL | #SYE | #CO |
| <b>GPT-4o</b>            | 62        | 32  | 22   | 28  | 67        | 30  | 23   | 28  | 21            | 13  | 11   | 13  |
| <b>Claude-3.5-Sonnet</b> | 52        | 30  | 24   | 27  | 45        | 24  | 20   | 22  | 22            | 10  | 10   | 10  |
| <b>Deepseek-V3</b>       | 44        | 22  | 19   | 21  | 64        | 36  | 28   | 35  | 19            | 11  | 9    | 11  |

more patches than AUTOCODEROVER in general, with up to 62 and 67 patches generated, respectively. Our investigation shows that AUTOCODEROVER generates fewer patches due to the non-deterministic behavior of its patch agent, which sometimes fails to locate the exact code region for applying the patch, even when the patch is successfully generated by the foundational model. We have also investigated the reasons why AGENTLESS and SWE-AGENT do not generate patches for the remaining artifacts. For AGENTLESS, the main reason is that it fails to localize the bug location correctly, which is an essential step before generating a patch. And SWE-AGENT fails to generate patches due to the cost limit of the foundational model, which prevents it from generating a patch for every artifact.

Not all generated patches are plausible (PL) to make a successful build. SWE-AGENT generates up to 36 plausible patches across the three models, while AGENTLESS achieves a comparable number (up to 32), indicating their ability to effectively fix the issues that block the build process. On the other hand, AUTOCODEROVER generates up to 13 plausible patches, which is lower than the other two tools.

The trend persists when considering syntactic equivalence (SYE), i.e., when patches are plausible and also syntactically equivalent to the ground truth. AGENTLESS and SWE-AGENT consistently outperform AUTOCODEROVER, achieving up to 24 and 28 syntactically equivalent patches respectively across the models. Notably, Claude-3.5-Sonnet leads with 24 SYE patches under AGENTLESS, suggesting stronger alignment with the human-made fixes. On the other hand, Deepseek-V3 achieves 28 SYE patches under SWE-AGENT.

Furthermore, we additionally performed a manual evaluation of the generated patches to assess whether they genuinely addressed the underlying causes of build failures. Plausibility

was initially measured by checking if a previously failing repository could be built successfully after applying a tool-generated patch. However, in some cases, patches produced successful builds without resolving the actual exception or error in the source code. For example, a patch that simply modified or disabled failing test cases could yield a successful build, since success is defined only by all tests passing. To account for such cases, we conducted manual inspection of each patch to ensure that the observed build success reflected a meaningful fix rather than a superficial change.

Manual inspection is consistent with other metrics. AGENTLESS and SWE-AGENT clearly outperform AUTOCODEROVER, achieving success rates of up to 28% and 35%, respectively, compared to only 13% for AUTOCODEROVER. Specifically, GPT-4o produced 28 successful patches when combined with AGENTLESS, while Deepseek-V3 achieved 35 successful patches with SWE-AGENT.

Among the foundation models, GPT-4o and Claude-3.5-Sonnet exhibit comparable performance across all tools, with GPT-4o maintaining a slight lead in average plausibility. While Deepseek-V3 achieves the highest performance when paired with SWE-AGENT, it produces fewer plausible or syntactically equivalent patches when integrated with AUTOCODEROVER and AGENTLESS.

## 4.8 Threats to Validity

We evaluated three foundation LLMs, GPT-4o, Claude-3.5-Sonnet, and Deepseek-V3, on 100 BUGSWARM artifacts. Each of these models has its own training cut-off date, and it is possible that they were partially trained on GitHub data, though the specifics are not publicly known. This raises a valid concern for all real-world datasets used in program repair. However, CI-BENCH remains applicable regardless of whether a fix was pushed into the associated repository before or after a model’s training cut-off date, as it can be applied to any artifact in the dataset. This indicates the critical importance of evolving

datasets such as BUGSWARM, which continuously grow over time with new artifacts. For the manual inspection in our evaluation, we carefully compared each generated patch against the developer patch; however, some false positives may remain.

## 4.9 Related Work on Benchmarking LLM-Based Program Repair

SWE-Bench [187] is a benchmark of 2,294 real-world GitHub issues for evaluating code repair task. SWE-Bench Lite [56] is a subset of SWE-Bench, popular among SE research communities. Other benchmarks, such as Multi-SWE-Bench [415], expand the benchmark to support multiple programming languages beyond Python. Agentic tools like SWE-AGENT [403], AUTOCODEROVER [434], AIDER [15], and the non-agentic AGENTLESS [387] have shown significant performance on these datasets. However, these benchmarks face significant limitations: they often lack dataset diversity, fail to provide robust support for reproducibility, and are primarily restricted to GitHub issues [455, 459]. In contrast, BUGSWARM provides reproducible CI failures for both Java and Python, covering a broader range of repositories. CI-BENCH is a unified framework that leverages BUGSWARM artifacts to enable the systematic evaluation of LLM-based program repair tools on diverse and reproducible real-world CI failures.

## Chapter Summary

The two halves of this chapter cover the two directions in which large language models meet software engineering. Treating LLMs as opaque components, semantic markers paired with delta debugging isolate the offending prompt fragment without access to model internals; the methodology turns prompt debugging from an ad hoc practice into a systematic procedure. Treating LLMs as repair tools, CI-BENCH measures three state-of-the-art LLM-based

program repair tools end-to-end against real continuous integration failures and shows that the best tool fixes only about a third, with the remainder dominated by failures that require multi-step or domain-specific reasoning current tools do not yet support. Together, these results ground both directions of the interface between AI and software engineering and set up the next part of the dissertation, where we return to scalability, first in classical distributed systems (Chapter 5) and then in the AI infrastructure (Chapter 6) that increasingly carries the workload these chapters depend on.

This chapter is based on two papers: “Delta Debugging for LLM-Integrated Systems,” co-authored with Muhammed Numair Mansur, Martin Schäfer, Zeya Chen, Tancred Lepoint, and Willem Visser during an internship at Amazon Web Services, and published at the International Conference on Software Engineering, Software Engineering in Practice Track, in 2026; and “CI-BENCH: A Framework for Evaluating Large Language Model Tools on CI Failures,” co-authored with Raian Latif Nabil and Cindy Rubio-González, and published at the International Conference on Software Engineering, Demonstrations Track, in 2026. The CI-BENCH work was supported by NSF grants CNS-2016735, CNS-2346396, and CCF-2119348. CI-BENCH is available at <https://github.com/bugswarm/ci-bench>.

## Part III

# Scalability



# Chapter 5

## Detecting Scalability Faults in Distributed Systems

Given the constraints of Moore’s Law and Dennard scaling, coupled with the escalating demand for computing power, the last decade has seen unprecedented deployment scales: over 300,000 Cassandra nodes are hosting over 100 petabytes of data at Apple, tens of thousands of MapReduce/Spark jobs are running on the Hadoop clusters of X (Twitter), and trillions of Kafka messages are processed per day at LinkedIn [31, 52, 60]. With highly-scalable distributed systems [30], one can surpass the limitations of a single machine in meeting the increasing demand in computation and storage. To accommodate growth, distributed systems employ two fundamental scaling strategies: *scale-out* (horizontal scaling), which adds more nodes to the cluster, and *scale-up* (vertical scaling), which increases resources per node. While these systems are inherently designed to scale, their complexity makes them prone to *scalability faults*.

Scalability faults are defined as faults that manifest only when one or more system aspects (e.g., number of nodes, data volume, concurrent requests) increase beyond certain thresholds, but remain latent at smaller scales [211]. Scalability faults are orthogonal to traditional fault categories such as functional, performance, security, or concurrency faults. A scalability fault

```

    private void logDecommissioningNodesStatus() {
-   StringBuilder sb = new StringBuilder();
    for (DecommissioningNodeContext d : decomNodes.values()) {
+   StringBuilder sb = new StringBuilder();
        sb.append( /* decommissioning node status */ );
        /* ... */
        /* out-of-memory when size of decomNodes is large */
+   LOG.debug("Decommissioning node: " + sb.toString());
    }
-   LOG.info("Decommissioning Nodes: " + sb.toString());
    }

```

Figure 5.1: YR-6188 [13] : “... and yes, it does throw an OOM exception in case of large clusters”.

may exhibit symptoms similar to these categories (e.g., performance degradation, resource exhaustion, or race conditions), but is fundamentally characterized by its *scale-dependent manifestation*: the fault remains hidden during small-scale testing and only emerges under large-scale conditions. Revealing scalability faults requires large-scale deployment, which is difficult and expensive to achieve during the development and testing phases. As a result, scalability faults are often discovered in production environments [16–18, 38, 46, 47], leading to resource shortage, performance degradation, or even system crashes. Figure 5.1 shows an example of a scalability fault and its fix in Hadoop Yarn. In YR-6188 [13], developers notice that the unbounded design of a data structure and a large number (i.e.,  $\geq 2000$ ) of decommissioning nodes lead to a system crash due to an out-of-memory (OOM) error. The root cause is the combination of (1) the `for` loop whose number of iterations grows with the number of decommissioning nodes, and (2) an unbounded data structure `StringBuilder` whose size grows in every iteration of the loop. If the number of decommissioning nodes is large, the `StringBuilder` object will exhaust memory and trigger an OOM error. The fix is to move the `StringBuilder` object inside the `for` loop to leave it out of scope after each iteration, thus allowing the garbage collector to free up memory. This example illustrates the scale-dependent nature of scalability faults: the code functions correctly with a small number of decommissioning nodes but fails catastrophically when that number exceeds a threshold.

Table 5.1: Breakdown of faults in the empirical study.

| Dimension    | Examples                              | Cassandra | Hadoop | HBase | HDFS | Ignite | Kafka | MapReduce | Spark | Storm | Yarn | Total      |
|--------------|---------------------------------------|-----------|--------|-------|------|--------|-------|-----------|-------|-------|------|------------|
| Load         | # requests, RPCs, jobs                | 9         | 15     | 13    | 36   | 13     | 9     | 22        | 15    | 11    | 11   | 154        |
| Data         | # tables, files, dirs, or their sizes | 29        | 10     | 12    | 49   | 14     | 21    | 7         | 27    | 1     | 6    | 176        |
| Cluster      | # peers, datanodes, namenodes         | 29        | 5      | 9     | 17   | 17     | 14    | 2         | 1     | 2     | 2    | 98         |
| Failure      | # node failures                       | 0         | 2      | 2     | 3    | 1      | 1     | 4         | 1     | 1     | 1    | 16         |
| <i>Total</i> |                                       | 67        | 32     | 36    | 105  | 45     | 45    | 35        | 44    | 15    | 20   | <b>444</b> |

Unfortunately, scalability faults are not well understood. Previous work [211] has revealed their existence and provided a high-level overview of their potential impacts based on a small number of observations. However, many questions remain unanswered: Which aspects of a system cause scalability faults when scaled? Are there common root causes and/or code anti-patterns that induce scalability faults? Is it possible to systematically detect scalability faults without large-scale deployment? Each of these questions implies significant challenges, as scalability faults are often difficult to reproduce and diagnose unless the system is deployed at large scale.

To address the above questions, we conduct a comprehensive empirical study encompassing 444 scalability faults from 10 large-scale distributed systems. For each scalability fault, we manually inspect the issue report to identify (1) which system aspects are being scaled, and (2) the root causes and/or common anti-patterns that may induce scalability faults.

From (1), we report the aspects of the system that are subject to change in scale and significantly impact the system’s scalability with their growth. We refer to such aspects as **scalable system dimensions** [131, 132], which we categorize into four kinds: (i) *load*, the volume of work the system processes (e.g., #requests, RPCs, jobs); (ii) *data*, the volume of items the system stores (e.g., #tables, files, directories, or their sizes); (iii) *cluster*, the size of the deployment (e.g., #peers, datanodes, namenodes); and (iv) *failure*, the number of concurrent component failures the system must tolerate (e.g., simultaneously failing nodes during recovery). Table 5.1 lists examples of each scalable system dimension and their corresponding number of faults.

From (2), we identify four root-cause categories of scalability faults: (i) *compute*, execution-time overhead tied to scale; (ii) *unbound*, resource-consumption overhead tied to scale;

(iii) *bloat*, scalability-limiting data structure designs; and (iv) *logic*, logic flaws that surface only at large scale. Within these categories, we further identify 11 **anti-patterns**, i.e., concrete code or design forms that repeatedly induce scalability faults, with three or four anti-patterns per category, described individually in Section 5.1. Across these anti-patterns, we observe that **dimensional code fragments (DCFs)** play a fundamental role. A DCF is an iterative code fragment whose number of iterations is correlated with one or more scalable system dimensions. For example, the `for` loop in Figure 5.1 is a DCF whose iteration count grows with the number of decommissioned nodes. DCFs are common and often necessary in scalable systems, and are neither faults nor anti-patterns by themselves; they become fault-relevant only when associated with certain anti-patterns (e.g., the unbounded `StringBuilder` in Figure 5.1). In our study, DCFs underlie 67.6% of scalability faults in the *compute* and *unbound* categories; the remaining 32.4% in the *bloat* and *logic* categories involve other mechanisms.

Then, by formalizing the insights from our empirical study, we design and implement SCALELENS, a DCF-centered approach to detect scalability faults and identify their root causes while utilizing a single machine. SCALELENS consists of two main components: SCALEVIEW and SCALEPICK, which perform dynamic and static analyses, respectively. SCALEVIEW uses runtime instrumentation together with scaling workloads to collect execution traces. These traces are then automatically analyzed to determine correlations between scalable system dimensions and code fragments to identify DCFs and categorize their computational complexity. SCALEPICK performs call-graph analysis on the DCFs to identify anti-patterns and pinpoint fragments that lead to scalability faults.

Our evaluation of SCALELENS is two-fold. First, we evaluate SCALELENS on 55 previously known real-world scalability faults from Cassandra, HDFS, and Ignite, which are widely-used distributed systems. Our evaluation shows that SCALELENS is successful at fully detecting 36 (and partially detecting 2 more) out of 55 scalability faults, outperforming the baseline by  $4.2\times$ . Second, we evaluate the capability of SCALELENS to detect previously

unknown scalability faults. Specifically, we use SCALELENS to analyze the latest stable versions of Cassandra, HDFS, and Ignite, in which it detects a total of 334 DCFs with associated anti-patterns. After manual inspection, we find that all cases constitute problematic behaviors. We are in the process of reporting the new faults; so far 5 of reported DCFs have been confirmed by developers, and 4 more are under investigation.

In summary, this chapter makes the following contributions:

- We conduct the largest-to-date study on scalability faults, from which we derive the existence of DCFs and conclude common scalability anti-patterns (Section 5.1).
- We design and implement SCALELENS, a novel approach that combines dynamic and static analyses to detect scalability faults based on the synergy between DCFs and anti-patterns (Section 5.2).
- We evaluate SCALELENS on 55 previously known real-world scalability faults found in Cassandra, HDFS, and Ignite, showing the effectiveness of SCALELENS in detecting and precisely identifying root causes for 36 of them. Compared to the baseline, SCALELENS improves the DCF detection by  $4.2\times$  (Section 5.3.2).
- We demonstrate the ability of SCALELENS to identify 334 DCFs with associated anti-patterns in the latest stable versions of Cassandra, HDFS, and Ignite. We discuss their behavior and implications, as well as our ongoing process of reporting new faults (Section 5.3.3).

## 5.1 Empirical Study of Scalability Faults

This section presents our empirical study on scalability faults, with a discussion of their root-cause categories and common anti-patterns.

## Methodology

### System Selection

We considered 10 large-scale distributed systems: Cassandra [20] (CA), Hadoop [21] (HA), HBase [22] (HB), HDFS [23] (HD), Ignite [24] (IG), Kafka [25] (KF), MapReduce [26] (MR), Spark [27] (SP), Storm [28] (ST), and Yarn [29] (YR). We chose these systems because (1) they are open-source with public and highly organized JIRA issue tracking systems, (2) they are typically deployed at scale and (3) the selection covers the most common categories of distributed systems, including databases, file systems, and parallel computing.

### Issue Selection

We collected all (over 199K) issue reports from the JIRA issue tracking systems of the target systems from 2007 to 2024. Our initial attempt to use keywords to automatically identify reports describing scalability faults was unfruitful due to the wide variety of scalability problems and the lack of specific keywords (detailed in Section 5.4.2). Therefore, we performed a two-pass manual inspection. In the first pass, we read the title and description of the reports to identify bug reports that involve the increase in the scale of any aspect of the system. In the second pass, we performed a more detailed review of the candidate bug reports by examining the aspects being scaled when symptoms are observed. We then categorized those into four dimensions: *load*, *data*, *cluster*, and *failure*. This process involved 8 individuals and spanned over one year, leading to the identification of 444 scalability faults, detailed in Table 5.1.

### Root-Cause and Anti-Pattern Analysis

We manually analyzed the 444 scalability faults to understand the nature of each fault and its root cause. This process consisted of examining the contents of the report, discussions, source code, and available patches/pull requests. To ensure reliability and minimize bias, we

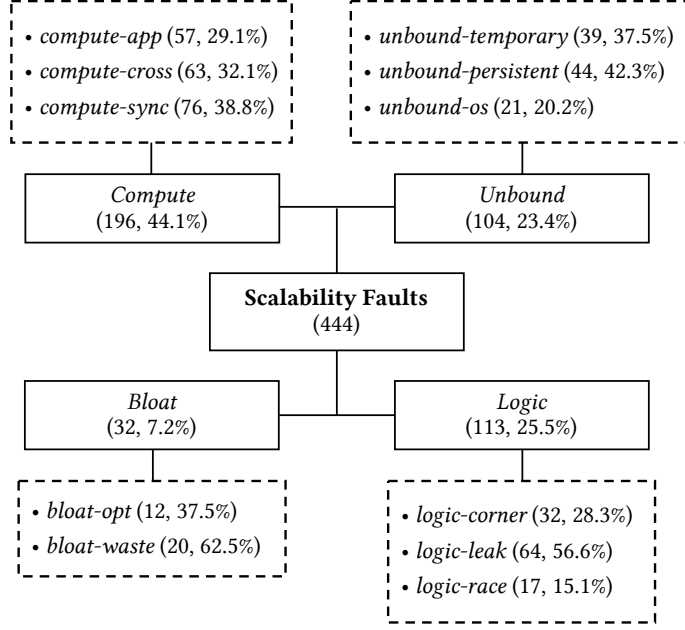


Figure 5.2: Scalability fault anti-patterns.

employed a multi-stage adjudication process. First, each issue was independently reviewed and tagged by two authors, who categorized it by root-cause category (compute, unbound, bloat, or logic) and its associated anti-pattern. Second, a third author reviewed all cases where the initial taggers disagreed or expressed uncertainty. Finally, all authors participated in group discussions to resolve remaining conflicts and validate the final categorization. All analyzed issue reports are publicly available in the respective JIRA repositories, and we provide supplementary material including the complete set of bug reports analyzed in this study, detailed tagging guidelines, and a full catalog of issue categorizations to support reproducibility and enable further research.

Our analysis has revealed four **root-cause categories** of scalability faults, *compute*, *unbound*, *bloat*, and *logic*, and 11 **anti-patterns** grouped under them, as shown in Figure 5.2. Across these anti-patterns, a recurring structural property stands out: many of them involve code fragments that iterate (either implicitly or explicitly) a number of times that grows with the scale of the system. We refer to these as **dimensional code fragments (DCFs)**. DCFs are not themselves part of the taxonomy; instead, they are a cross-cutting property

|                                                                                                                                                                                                                                                                                           |                                                                                                                                                                                                                      |                                                                                                                                                                                                                                                      |                                                                                                                                                                                                                                            |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> 1 void unwindEvicts(Partition[] all){ 2   // for every partition 3   for(Partition o : all){ 4     // for every version 5     for(Partition i : o.versions()){ 6       if(i.isExpired()) { 7         i.markForRemoval(); 8       } 9     } 10  } 11 } </pre> <p>(a) compute-app</p> | <pre> void updateOwner(Partition[] ps){   // for every partition   for(Partition p : ps){     // cross-system call     ZKClient.send(p.id, p.owner);   } } </pre> <p>(b) compute-cross</p>                           | <pre> void updateTokens(Token[] ts){   // holds a lock   writeLock();   // for every token   for(Token t : ts){     if(!cachedTokens.contains(t)){       cachedTokens.add(t);     }   }   writeUnlock(); // release } </pre> <p>(c) compute-sync</p> | <pre> void loadCache(Domain[] ds){   for(Domain d : ds){     // bring all into memory     Entity[] es = d.load();     for(Entity e : es){       // accumulate in cache       cache.put(e);     }   } } </pre> <p>(d) unbound-temporary</p> |
| <pre> 1 void processRequest(Request r){ 2   // execute 3   Result re = r.execute(); 4   if(re.isDone()){ 5     // add without checking 6     // size or memory usage 7     responseQueue.add(re); 8   } 9 } </pre> <p>(e) unbound-persistent</p>                                          | <pre> void processRequest(Request r){   Result re = r.execute();   // one connection per request   Socket connection = \     new Socket(r.ip, r.port, 10);   replyTo(connection, rs); } </pre> <p>(f) unbound-os</p> | <pre> class IndexSummary {   // Long[] is preferable   List&lt;Long&gt; pos = new ArrayList&lt;&gt;();   // byte[][] is preferable   Map&lt;Byte, Byte&gt; keys = new Map&lt;&gt;();   // ... } </pre> <p>(g) bloat-opt</p>                          | <pre> class StatsMetadata {   // heavy object   byte[] min = new byte [1000000];   byte[] max = new byte [1000000];   // ... } </pre> <p>(h) bloat-waste</p>                                                                               |

Figure 5.3: Code samples for the *compute*, *unbound*, and *bloat* faults. From (a) to (h), they are based on IG-8681, KF-5642, CA-5456, YR-7147, CA-15013, HA-15696, CA-5506, and CA-15400, respectively.

shared mainly by the *compute* and *unbound* anti-patterns. Below, we describe the root-cause categories and anti-patterns in turn, drawing on DCFs where the anti-pattern’s behavior is directly tied to scale-dependent iteration.

### 5.1.1 Compute Faults

*Compute* faults are related to bottlenecks caused by the increase in size of one or more scalable system dimensions (e.g., *#partitions* in Figure 5.3a). It is the largest category with 196 fault reports, representing 44.1% of the total, and includes 3 anti-patterns:

- **compute-app** faults, with 57 reports, account for 29.1% of this category and represent cases in which bottlenecks are caused by DCFs in performance critical paths. Even when the computations inside the DCF are not considered costly, the complexity (commonly  $\geq 2$ ) produces notable performance degradation. For example, in IG-8681 [10] (Figure 5.3a), `unwindEvicts` is invoked every time a remote Ignite command is processed. This method contains a quadratic DCF of  $O(P^2, P = \# \text{ partitions})$  (highlighted at lines 3 and 5). Since remote command execution is part of the critical path, invoking this method is reportedly problematic.



- **compute-cross** faults, with **63** reports, account for **32.1%** of this category and represent cases in which bottlenecks are caused by DCFs containing cross-system (e.g., methods from pluggable components, external clients, or IO operations) method calls. When cross-system method calls are performed inside DCFs, their computational cost gets amplified [37, 401] by the size of the related dimension, reportedly leading to severe performance degradation. For example, in [KF-5642](#) [11] (Figure 5.3b), `updateOwner` contains a linear DCF that iterates over each partition to update the ownership for a Kafka topic. Each iteration triggers synchronous communication with a cross-layer component (Zookeeper, highlighted at line 5). As the number of partitions grows, the number of synchronous messages increases and the execution time of the whole operation becomes dependent on the network speed and the load on the external component.

- **compute-sync** faults, with **76** reports, account for **38.8%** of this category and represent cases in which synchronization bottlenecks are caused by DCFs protected by global locks. Such fault anti-patterns have been observed previously [224, 333] in centralized architectures such as HDFS and include both cases in which the locks wrap DCFs and cases in which the locks are held inside the DCF, the former being the most frequent. For example, in [CA-5456](#) [4] (Figure 5.3c), the method `updateTokens` is invoked every time *range movements* are performed to maintain consistency when a Cassandra cluster’s topology is changing. The operation contains a linear DCF (line 4) and is performed while holding a global lock (highlighted at line 2). Other threads, in particular the ones that handle membership changes, have to wait until the whole operation finishes. As the number of tokens grows, the execution time of the code block between lines 4 and 8 grows too, reportedly causing a negative impact on elasticity.

### 5.1.2 *Unbound* Faults

*Unbound* faults are related to unconstrained resource consumption caused by the increase in size of one or more scalable system dimensions (e.g., *#entities/domains* in Figure 5.3d). This

category, with **104** fault reports, represents **23.4%** of the total and includes 3 anti-patterns:

- **unbound-temporary** faults, with **39** reports, account for **37.5%** of this category and represent cases in which temporary memory allocations, such as stack-level data structures, grow without bounds in response to the growth of one or more dimensions, reportedly causing out-of-memory errors and performance degradation due to frequent garbage collection. This anti-pattern can be intuitively related to the lack of buffering or paging when loading data from external sources (e.g., files or databases) [41, 108]. For example, in [YR-7147](#) [14] (Figure 5.3d), the method `loadCache` is invoked every time the component *Timeline Server* [61] of Yarn is started and contains a quadratic DCF of  $O(D * E, D = \# \text{ domains}, E = \# \text{ entities})$ . The cache is loaded from disk (highlighted at line 7), but the whole file is materialized in memory in one method call (highlighted at line 4), causing the component to run out of memory when the related files get too big. As reported, the component is killed and it can be restarted only if there is more memory available or the caching feature is disabled.

- **unbound-persistent** faults, with **44** reports, account for **42.3%** of this category and represent cases in which long-lived data structures, such as inbound/outbound message processing queues or multi-purpose caches, grow without bounds in response to the growth of one or more dimensions, reportedly causing out-of-memory errors. Note that the difference between this and the previous category is related to object lifetime, purpose and expected functionality: While the previous refers to temporary allocations typically used as containers/helpers, this category refers to persistent allocations that, accompanied by the necessary logic, are supposed to accommodate the growth of one or more dimensions but fail to do so due to issues in said logic. For example, in [CA-15013](#) [2] (Figure 5.3e), the Cassandra request processor contains a DCF of  $O(R, R = \# \text{ requests})$  that accumulates reportedly large responses in a persistent queue (highlighted at line 7). If the responding threads (which consume those objects) cannot keep up, the queue grows without bounds and the node runs out of memory.

- **unbound-os** faults, with **21** issues, account for **20.2%** of this category and represent cases in which OS resource allocations, such as threads and file descriptors, grow without bounds in response to the growth of one or more dimensions potentially exhausting those resources. These faults are uncommon in newer versions since the unbounded increase of threads and/or sockets is a known issue and is typically targeted in early fixes. For example, in [HA-15696](#) [6] (Figure 5.3f), an encryption-related HDFS component contains a DCF of  $O(R, R = \# \text{ requests})$  that creates a single **Socket** (with a corresponding file descriptor) every time a request is processed. Even if these connections are short-lived, the default idle-timeout is set to 10 seconds (highlighted at line 5). As reported, an explosive increase in the number of requests ends up in failures (the system runs out of file descriptors).

### 5.1.3 *Bloat* Faults

*Bloat* faults are related to data structures with scalability-limiting design [251, 265, 331], causing unexpected increases in memory consumption at larger scales. This category, with **32** fault reports, represents **7.2%** of the total and includes 2 anti-patterns:

- **bloat-opt** faults, with **12** issues, account for **37.5%** of this category and represent cases in which data structure design includes “space-time trade-offs” [124, 424] that, albeit intended to improve performance, obviate memory constraints at larger scales. For example, in [CA-5506](#) [5] (Figure 5.3g), Cassandra nodes utilize a data structure called **IndexSummary** that uses the collections framework [49] to store internal indexes. As the number of live **IndexSummary** instances grows, a 2× per-element overhead in positions (highlighted at line 3) and a 10× per-element overhead in keys causes 70% memory bloat in a cluster with billions of rows.

- **bloat-waste** faults, with **20** reports, account for **62.5%** of this category and represent cases in which data structure design includes unnecessarily large preallocated fields (e.g., fixed arrays or buffers), that obviate memory constraints at larger scales. Note that the difference between this and the previous category is related to functionality: The previous

focuses on allocations related to more complex techniques (such as indexing) which require a much more fine-grained design, while this category focuses on unnecessarily large (and often unused) space being preallocated (i.e., *wasted*). For example, in [CA-15400](#) [3] (Figure 5.3h), Cassandra creates one `StatsMetadata` instance for each *BigTableReader*, an internal data structure whose size is correlated to the load dimension. The former declares two fixed-size 1MB buffers (highlighted at lines 3 and 4) that, according to developers, are frequently not used in their entirety. When under high load, many live *BigTableReader* instances lead to high memory usage and eventually end in out-of-memory errors, affecting not one but several nodes in the cluster.

#### 5.1.4 *Logic Faults*

*Logic* faults are related to logic issues that become visible at large scales, such as corner cases, memory or resource leaks, and data races. This category, with **113** fault reports, represents a surprising **25.5%** of the total and includes 3 anti-patterns:

- **logic-corner** faults, with **32** issues, account for **28.3%** of this category and represent cases in which faulty logic (e.g., wrong choice of datatypes) is problematic at larger scales but harmless at smaller scales. For example, in [ST-3256](#) [12], a large Storm topology is used to communicate with custom terminals in a client-server fashion. The communication is handled by *worker* threads, which are uniquely identified across the whole deployment using a numeric 16-byte unsigned integer. Due to this, no more than 32K workers can be instantiated concurrently, which imposes a severe limit to parallelism (in the reporter’s words, “... *is a disaster for large scale computing*”).

- **logic-leak** faults, with **64** issues, account for **56.6%** of this category and represent cases in which memory, OS resources, or storage leaks at larger scales. For example, in [HD-13039](#) [8], HDFS *erasure coding* component exhibits a classic OS (file descriptor) leak: `createReader` is invoked when creating a `Reader` object on a per-request basis. However, a connection is created but never closed in the case of file creation failures. This leads to

extended downtimes (no more file descriptors available) when many of these failures happen.

- **logic-race** faults, with **17** issues, account for **15.1%** of this category and represent cases in which race conditions are observed at larger scales (e.g., when a large number of peers join the cluster at the same time), causing effects that range from data inconsistencies to system failures. For example, in [HA-16385](#) [7], a 53K-container heavily loaded Hadoop HDFS cluster is dealing with multiple node failures at the same time. In the middle of this failure storm, a race condition related to the calculation of the number of live peers triggers an assertion failure that ends up killing the HDFS *namenode*.

### 5.1.5 Key Insights

Our study has confirmed that scalability faults are challenging to detect. We identified four main categories of scalability faults, from which *compute* and *unbound* stand out; more than half of all faults (67.6%) fall into these categories, which are caused by the combination of DCFs and anti-patterns. From them, we observe the following scalability anti-patterns:

- *compute-app*: DCFs placed in performance-critical paths.
- *compute-cross*: DCFs with API or IO operations.
- *compute-sync*: DCFs protected by global locks.
- *unbound-temporary*: DCFs with memory allocations.
- *unbound-persistent*: DCFs that grows size of objects.
- *unbound-os*: DCFs that consumes OS resources.

Based on these observations, identifying DCFs and associated anti-patterns is essential to detect scalability faults. We also observe that:

- **The nature of DCFs** can be **explicit** or **implicit**. Explicit DCFs are **for** or **while** loops directly visible in the source code of the system or its related libraries, as shown in

Figure 5.3a-d, and implicit DCFs refer to cases where no loops are directly present, but the number of method invocations increases with one or more system dimensions as shown in Figure 5.3e-f. Due to the nature of DCFs, especially implicit ones, they are difficult to detect without runtime information.

- **DCF**s are not necessarily problematic by themselves, as they represent basic building blocks of distributed systems. For example, for *compute*-related DCFs, we observe that they become problematic only when present in performance sensitive paths, make external API or IO calls, or are protected by global locks. Therefore, detecting DCFs alone is not sufficient and identifying associated anti-patterns is essential to accurately detect scalability faults.

- **DCF complexity matters**, but it is not a strong indicator of faults by itself. For example, high-complexity functions could be considered acceptable in background paths but problematic in foreground or user-facing paths.

In addition to *compute* and *unbound* faults, we also observe *bloat* and *logic* faults. *Bloat* faults are related to inefficient data structure designs that lead to unnecessary memory overhead as the system scales, while *logic* faults are logic issues that become visible when the system is deployed at larger scales. Unlike *compute* and *unbound* faults, these two categories do not involve DCFs, and may require memory footprint and concurrency analyses for their detection. Together, they account for a smaller portion of the scalability faults (32.4%).

In the next section, we describe our approach, SCALELENS, to detect *compute* and *unbound* scalability faults. We leave further analysis of *bloat* and *logic* faults as future work.

## 5.2 Our Approach: ScaleLens

We present SCALELENS, an approach for detecting scalability faults in large-scale distributed systems. SCALELENS has two major components, illustrated in Figure 5.4: (1) SCALEVIEW that performs *dynamic* analysis to identify DCFs, and (2) SCALEPICK that performs *static*

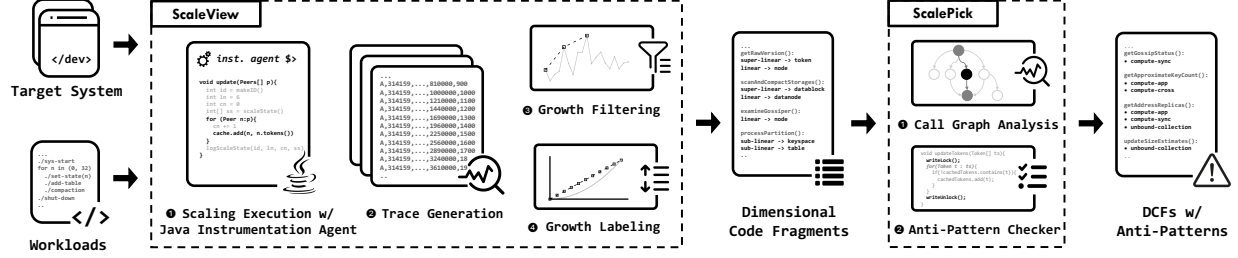


Figure 5.4: SCALELENS workflow.

analysis to detect anti-patterns associated with DCFs.

### 5.2.1 ScaleView

The goal of SCALEVIEW is to list all DCFs for a given system and their relationships with the system dimensions. Such information can help developers to *view* all potential scalability bottlenecks from a scaling perspective prior to actual production deployment.

Given the source code of a target system and a set of scaling workloads, SCALEVIEW (1) instruments the code to obtain execution traces, (2) parses and analyzes the execution traces to discover execution trends, (3) filters out noise to identify real growth, and (4) labels the DCFs according to their relationships to the system dimensions.

### Scaling Workloads and System Dimensions

The first step is to write a **scaling workload**: a script that systematically increases one or more scalable system dimensions (the *scalable* aspects of interest) while invoking the target system’s APIs. This process is manual and requires familiarity with deployment and operational practices of the target system. It is a one-time effort and can be reused across different versions of the system (implications are discussed in Section 5.4.3).

Figure 5.5 shows an example workload scaling the number of tables in a Cassandra cluster. The maximum number of tables is set on line 3, and the `for` loop on line 4 adds one table at a time. On lines 6 and 7, a table is created and data is inserted via `cqlsh`, the database operation interface provided by Cassandra. On line 8, compaction is run via

```

1  # Cassandra: scaling up tables and run compaction
2  ./setup-node && ./start-node
3  MAX_TABLES=128
4  for (( i = 1; i <= $MAX_TABLES; i++ )); do
5  ./set-scale-state $i
6  ./cqlsh -e "CREATE TABLE (...)" # create a new table
7  ./cqlsh -e "INSERT INTO (...)" # insert dummy data
8  ./nodetool (...) compact (...) # run compaction
9  ./sleep
10 done
11 ./shut-down

```

Figure 5.5: Scaling workload example.

`nodetool`, the system management utility provided by Cassandra. Compaction is optional, and can be replaced with other system operations (e.g., snapshot, scrub, repair, etc.) based on the interest of the user of SCALEVIEW.

The function `set-scale-state` on line 5 of Figure 5.5 is an API provided by SCALEVIEW for reporting the current **scale state** of the workload: the numeric value(s) of the dimension(s) being scaled at that moment of execution (here, the current number of tables, e.g., 17 after the 17th loop iteration). SCALEVIEW uses this reported value to correlate each observed iteration count of a code fragment with the scale of the target dimensions. A multi-dimensional scale state is also supported: for example, a workload that scales tables and rows simultaneously would call `set-scale-state` with a pair of values, such as (17, 10000) once the 17th table contains 10000 rows.

## Runtime Instrumentation

After scaling workloads are triggered, SCALEVIEW monitors the relationship between the current size of the dimensions and the number of iterations performed by a code fragment. Such monitoring is done by using bytecode-level runtime instrumentation [48, 50]. Based on the insights from Section 5.1, SCALEVIEW considers both explicit and implicit iterations.

- **Explicit.** As shown in Figure 5.6a, once a loop is detected by SCALEVIEW in the source code of the target system, SCALEVIEW adds a unique ID, line number, and a counter variable to the loop. Library loops are also instrumented but in this case the unique ID



|                                                                                                                                                                                                                                              |                                                                                                                                                                                                                             |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> 1 void update(Peers[] p) { 2   int id = makeId(); 3   int ln = 5, cn = 0; 4   int[] ss = getScaleState(); 5   for(Peer n : p) { 6     cn += 1; 7     cache.add(n, n.tokens()); 8   } 9   logScaleState(id, ln, cn, ss); 10 } 11 </pre> | <pre> void add(K k, V[] v) {   int id = callerId();   int ln = callerLine();   int cn = 0;   int[] ss = getScaleState();   for(V val : v) {     cn += 1;     map.put(k, val);   }   logScaleState(id, ln, cn, ss); } </pre> |
| (a) Application                                                                                                                                                                                                                              | (b) Library                                                                                                                                                                                                                 |

Figure 5.6: Runtime instrumentation. Code in grey is instrumented.

corresponds to the ID of the caller, as shown in Figure 5.6b. This is done to maintain the scope of reporting within the target system. Finally, for both types of loops, SCALEVIEW retrieves the scale state information with `getScaleState()` and logs the scale state along with the counter variables with the `logScaleState()`.

- **Implicit.** SCALEVIEW captures implicit iterations by capturing the relationship between the number of invocations of each method and the current state of the scaling dimensions.

## Trace Generation

The combination of scaling workloads and instrumentation produces traces that capture runtime information about loop executions. Specifically, a tuple  $\langle type, methodID, line\#, execID, \#iterations, SS \rangle$  is recorded for each loop execution. In the tuple,  $type \in \{A, L, I\}$  represents the type of the loop: application, library, or implicit.  $methodID$  and  $line\#$  record the method and location of the loop, respectively.  $execID$  is a unique identifier for each execution of the loop, and  $\#iterations$  is the final iteration count of a given loop execution.  $SS$  represents the scale state in an array with one or more numbers for one- or multi-dimensional executions, respectively. For example, tuple  $\langle A, 314159, 64, 265358, 810000, 900 \rangle$  describes the loop execution with ID 265358: an execution of the application loop located in the method with ID 314159 on line 64 iterated a total of 810000 times when the number of nodes (i.e., the scale state) reached 900, respectively.

## Growth Filtering

The goal of growth filtering is to detect DCFs by identifying iterative code fragments with a positive correlation between the number of iterations and the target scaling dimensions, and discarding the rest. This is done with heuristics based on the growth patterns shown in Figure 5.7. Besides the trivial clear flat (Figure 5.7a) and clear growth (Figure 5.7b) patterns, we also observed noisy growth (Figure 5.7c) and noisy flat (Figure 5.7d) patterns. SCALEVIEW performs a 3-step empirical-driven filtering process: **(1)** only keep the data points that are larger than the maximum of all the previous data points (i.e., the dashed boxes in Figure 5.7), **(2)** remove the code fragments with less than 10% of data points retained after the first step (e.g., Figure 5.7a and Figure 5.7d), and **(3)** keep the code fragments whose correlation to the scaling dimension is high (i.e.,  $\geq 0.9$ ). Such a filtering process excludes both clear and noisy flat patterns.

## Growth Labeling

Finally, SCALEVIEW categorizes DCFs based on their previously filtered growth trends as (1) super-linear, (2) linear, or (3) sub-linear. To do so, we select a set of theoretical complexity models (e.g.,  $I = O(D^2)$  is super-linear,  $I = O(D)$  for linear, and  $I = O(\log D)$  for sub-linear, where  $I$  is the number of iterations and  $D$  is the scale of dimension) and use similarity measures [77] to detect the closest theoretical model and assign the corresponding label to the fragment. We consider that this simple labeling can be useful for developers to estimate the urgency or priority of the issue. For example, a super-linear (e.g.,  $O(N^2)$  or above) code fragment intuitively brings more negative impacts than a linear (e.g.,  $O(N)$ ) fragment under the same condition.

### 5.2.2 ScalePick

DCFs reported by SCALEVIEW represent potential scalability bottlenecks in the system, but as discussed previously, not all DCFs are necessarily problematic. Thus, given the

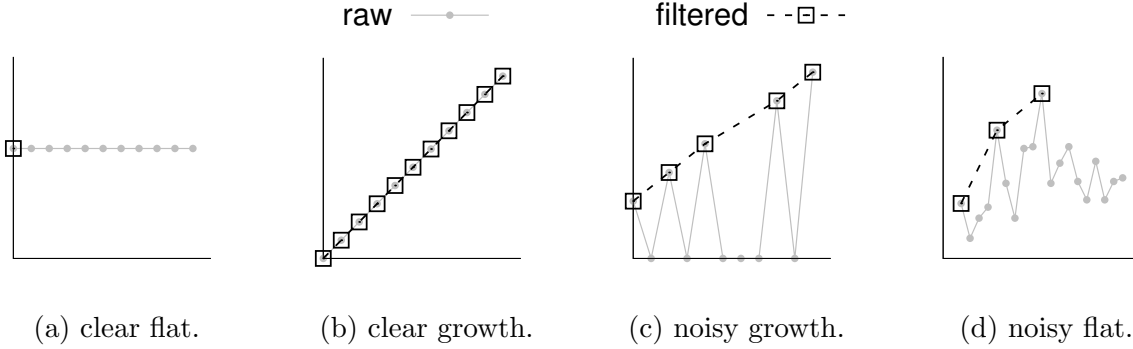


Figure 5.7: Growth patterns.

source code of a target system, as well as the DCFs found by SCALEVIEW, SCALEPICK (1) performs call graph analysis from DCFs, and (2) uses static checkers to identify associated anti-patterns, if any.

SCALEPICK incorporates a set of simple static checkers that identify the first 6 anti-patterns from our empirical study using CODEQL [39], and all checkers are applied to each DCF to flag associated anti-patterns. SCALEPICK’s checkers use **program slicing** [92], a technique that computes the set of statements that may influence (backward slice) or be influenced by (forward slice) a given statement. We apply slicing over the call graph of the target system: a backward slice from a DCF gathers the methods that can reach it, while a forward slice from a DCF gathers the methods and the allocation, I/O, or synchronization sites it may reach. Each checker below inspects a specific slice and flags the DCF if the slice contains a checker-specific sink. Details of the checkers are discussed below.

- The **compute-app** checker detects DCFs in performance or operational critical paths. Those code paths could be user-facing or foreground depending on the testing requirements. To detect this anti-pattern, we perform backward slicing [92] from the DCFs over the call graph to check reachability from any specified user-facing or foreground API calls. Reachable DCFs are flagged as *compute-app*.

- The **compute-cross** checker finds external API or IO operations within DCFs. Such operations could be read/write interactions with disk, network, or database, which are often performed in a synchronous fashion. To detect this anti-pattern, we perform forward slic-

ing [92] from the DCFs over the call graph to find DCFs associated with external API or IO invocations. Such DCFs are then flagged as *compute-cross*.

- The **compute-sync** checker finds instances where lock contentions may be caused by DCFs holding a global lock. Such lock contentions typically happen in foreground/background interactions. To detect this anti-pattern, we perform both backward and forward slicing from the DCFs over the call graph. If a DCF holds a global lock that is also used by other components, then it is flagged as *compute-sync*.

- The **unbound-temporary** checker finds unbounded memory allocations whose lifetime is local to the method that contains the DCF. We compute the forward slice from the DCF and check whether the slice contains a memory-growth sink on a *method-local* target, i.e., an object scoped to the method: (i) a fresh allocation (e.g., a constructor call for a Java collection class such as `ArrayList`, `HashMap`, or `StringBuilder`; a Java array creation `new T[n]`; or an external buffer-returning API whose return value is not released in the slice), or (ii) a size-growing call (e.g., `append`, `add`, `put`) on a mutable container declared in the containing method (and hence unreachable after the method returns). If either holds, the DCF is flagged as *unbound-temporary*.

- The **unbound-persistent** checker detects unbounded growth of long-lived objects reachable from a DCF, i.e., objects whose lifetime outlives the enclosing method. We compute the forward slice from the DCF and check whether it contains a size-growing call on a *non-local* target: an instance field, a static field, or a mutable object passed by reference (e.g., a shared queue, map, or cache). If such a call is in the slice, the DCF is flagged as *unbound-persistent*.

- The **unbound-os** checker identifies DCFs that consume system resources, such as file descriptors, sockets, or threads, without proper management. Such resources are often limited and could be exhausted when the system scales. To detect this anti-pattern, we perform forward slicing from the DCFs over the call graph to see if the DCF consumes system resources. If so, the DCF is flagged as *unbound-os*.

Table 5.2: Evaluation results.  $T_{DCF}$  denotes the type of reported DCF, where  $E$  stands for explicit and  $I$  for implicit. In  $DCF$  columns for both SCALELENS and SFIND,  $\times$  denotes that the related DCF was not detected and  $\checkmark$  denotes positive results. In *anti-pattern* column under SCALELENS,  $\times$  denotes the anti-pattern is not detected given the detection of DCF, and empty cell means not evaluated because the DCF is not detected.

| Issue Information |              |                 | SCALENS      |              | SFIND    | Issue Information |              |                 | SCALENS      |              | SFIND        | Issue Information |              |                 | SCALENS      |              | SFIND        |
|-------------------|--------------|-----------------|--------------|--------------|----------|-------------------|--------------|-----------------|--------------|--------------|--------------|-------------------|--------------|-----------------|--------------|--------------|--------------|
| ID                | $T_{DCF}$    | Anti-Pattern    | DCF          | Anti-Pattern | DCF      | ID                | $T_{DCF}$    | Anti-Pattern    | DCF          | Anti-Pattern | DCF          | ID                | $T_{DCF}$    | Anti-Pattern    | DCF          | Anti-Pattern | DCF          |
| CA-19534          | $I$          | $u$ -persistent | $\checkmark$ |              | $\times$ | CA-14660          | $E \ \& \ I$ | $c$ -sync       | $\checkmark$ | $\checkmark$ | $\times$     | HD-16100          | $E$          | $c$ -sync       | $\checkmark$ | $\checkmark$ | $\checkmark$ |
| CA-19477          | $E \ \& \ I$ | $c$ -cross      | $\checkmark$ | $\checkmark$ | $\times$ | CA-14239          | $E$          | $u$ -persistent | $\checkmark$ | $\checkmark$ | $\times$     | HD-15621          | $I$          | $u$ -persistent | $\times$     |              | $\times$     |
| CA-19412          | $E \ \& \ I$ | $u$ -persistent | $\checkmark$ | $\checkmark$ | $\times$ | CA-14096          | $E$          | $u$ -persistent | $\checkmark$ | $\checkmark$ | $\times$     | HD-15406          | $E$          | $c$ -sync       | $\checkmark$ | $\checkmark$ | $\times$     |
| CA-19336          | $I$          | $u$ -temporary  | $\checkmark$ | $\checkmark$ | $\times$ | CA-13993          | $I$          | $c$ -cross      | $\checkmark$ | $\checkmark$ | $\times$     | HD-14859          | $E$          | $c$ -app        | $\checkmark$ | $\checkmark$ | $\checkmark$ |
| CA-19107          | $E \ \& \ I$ | $c$ -app        | $\checkmark$ | $\checkmark$ | $\times$ | CA-13948          | $E$          | $c$ -sync       | $\checkmark$ | $\checkmark$ | $\checkmark$ | HD-14854          | $E$          | $c$ -sync       | $\times$     |              | $\times$     |
| CA-18773          | $E \ \& \ I$ | $c$ -app        | $\checkmark$ | $\checkmark$ | $\times$ | CA-13923          | $E \ \& \ I$ | $c$ -sync       | $\checkmark$ | $\checkmark$ | $\times$     | HD-14771          | $E$          | $c$ -cross      | $\times$     |              | $\times$     |
| CA-18546          | $E \ \& \ I$ | $u$ -persistent | $\checkmark$ | $\checkmark$ | $\times$ | CA-13569          | $I$          | $u$ -persistent | $\times$     |              | $\times$     | HD-14657          | $I$          | $c$ -sync       | $\checkmark$ | $\checkmark$ | $\times$     |
| CA-17787          | $I$          | $u$ -temporary  | $\checkmark$ | $\checkmark$ | $\times$ | CA-13299          | $I$          | $u$ -persistent | $\times$     | $\times$     | $\times$     | HD-14613          | $E$          | $c$ -app        | $\checkmark$ | $\checkmark$ | $\checkmark$ |
| CA-17691          | $E \ \& \ I$ | $c$ -sync       | $\checkmark$ | $\checkmark$ | $\times$ | CA-13215          | $E \ \& \ I$ | $c$ -app        | $\checkmark$ | $\checkmark$ | $\times$     | HD-14370          | $E$          | $c$ -cross      | $\times$     |              | $\times$     |
| CA-17342          | $E \ \& \ I$ | $c$ -app        | $\checkmark$ | $\checkmark$ | $\times$ | CA-13065          | $E$          | $c$ -sync       | $\times$     | $\times$     | $\times$     | HD-14366          | $E$          | $c$ -sync       | $\checkmark$ | $\checkmark$ | $\checkmark$ |
| CA-16380          | $E \ \& \ I$ | $c$ -sync       | $\checkmark$ | $\checkmark$ | $\times$ | CA-12281          | $I$          | $c$ -sync       | $\times$     | $\checkmark$ | $\times$     | HD-14201          | $I$          | $c$ -cross      | $\times$     |              | $\times$     |
| CA-16261          | $I$          | $u$ -persistent | $\times$     |              | $\times$ | CA-12245          | $E$          | $c$ -sync       | $\times$     | $\times$     | $\times$     | HD-14171          | $E$          | $c$ -app        | $\checkmark$ | $\checkmark$ | $\checkmark$ |
| CA-16201          | $E$          | $u$ -temporary  | $\checkmark$ | $\checkmark$ | $\times$ | CA-11748          | $I$          | $u$ -persistent | $\times$     |              | $\times$     | HD-13821          | $I$          | $c$ -sync       | $\times$     |              | $\times$     |
| CA-15364          | $I$          | $c$ -app        | $\checkmark$ | $\checkmark$ | $\times$ | CA-10654          | $E$          | $c$ -sync       | $\times$     | $\times$     | $\times$     | HD-13768          | $E$          | $c$ -sync       | $\times$     |              | $\times$     |
| CA-15141          | $E \ \& \ I$ | $c$ -sync       | $\checkmark$ | $\checkmark$ | $\times$ | CA-9882           | $E \ \& \ I$ | $c$ -sync       | $\checkmark$ | $\checkmark$ | $\checkmark$ | HD-13692          | $E$          | $u$ -persistent | $\checkmark$ | $\checkmark$ | $\checkmark$ |
| CA-15013          | $I$          | $u$ -persistent | $\times$     |              | $\times$ | HD-17097          | $E$          | $c$ -sync       | $\checkmark$ |              | $\times$     | IG-14076          | $E \ \& \ I$ | $u$ -persistent | $\checkmark$ | $\checkmark$ | $\times$     |
| CA-14855          | $E \ \& \ I$ | $u$ -persistent | $\checkmark$ | $\checkmark$ | $\times$ | HD-17096          | $E$          | $u$ -temporary  | $\checkmark$ | $\times$     | $\checkmark$ | IG-12189          | $I$          | $c$ -cross      | $\checkmark$ | $\checkmark$ | $\times$     |
| CA-14840          | $I$          | $u$ -persistent | $\checkmark$ | $\times$     | $\times$ | HD-16989          | $I$          | $c$ -sync       | $\checkmark$ | $\checkmark$ | $\times$     | IG-12087          | $I$          | $c$ -sync       | $\checkmark$ | $\checkmark$ | $\times$     |
| CA-14747          | $I$          | $u$ -persistent | $\times$     |              | $\times$ |                   |              |                 |              |              | Total        | 55                |              |                 | 38           | 36           | 9            |

To demonstrate these checkers in practice, consider the for-loop DCF from [YR-6188](#) [13] (the motivating example from [Chapter 5](#)). In this example, a `StringBuilder` declared within the method containing the DCF is updated via repeated calls to `append`, which is reachable from the DCF. Because the buffer’s size grows with every iteration, the unbound-temporary checker identifies this as a “size-growth sink” under case (ii). This detection aligns with both the manual tagging of [YR-6188](#) [13] and SCALEVIEW’s runtime data, which shows the buffer growing linearly with the number of decommissioning nodes.

### 5.3 Evaluation of ScaleLens

This evaluation aims to answer the following questions:

- RQ1** How effective is SCALELENS in identifying previously known real-world scalability faults, and how does it compare to the baseline?
- RQ2** Can SCALELENS detect previously unknown scalability faults in the latest stable versions of distributed systems?

### 5.3.1 Experimental Setup

#### Benchmark

As far as we know, there is no existing benchmark that targets scalability faults in distributed systems. Hence, we build our own based on the scalability faults we collected for our empirical study (Section 5.1). From the 444 scalability faults, we focus on those reported for Cassandra, HDFS, and Ignite, the three systems with the most scalability faults (217 combined, as shown in Table 5.1) and the most up-to-date documentation. Then, we (1) determine if the system’s versions related to the faults are deployable and runnable, and (2) select the faults that are within the scope of SCALELENS, i.e., *compute* and *unbound* faults. The steps above result in 55 real-world scalability issues, as listed in Table 5.2.

#### Baseline

The most closely related work to SCALELENS is SCALECHECK [333], an approach designed to discover scalability faults. SCALECHECK has two components: SFIND and STEST. Among the two, SFIND is responsible for detection, exposing *scale-dependent loops* through program analysis. Such loops are a narrower form of DCFs. Since both SFIND and SCALEVIEW identify code fragments with scale-correlated iteration counts from source code and workloads, we use SFIND as the baseline for SCALEVIEW. Technical details of SCALECHECK and a head-to-head comparison with SCALELENS are provided in Section 5.5.

#### Workloads

We implemented 8 workloads for Cassandra, 3 for HDFS, and 2 for Ignite. In those, we scaled a total of 8 dimensions (*nodes*, *tokens*, *tables*, *rows* for Cassandra, *datanodes*, *datablocks*, *files* for HDFS, and *rows* for Ignite) and exercised a total of 6 APIs (*snapshot*, *compaction*, *repair* for Cassandra, *snapshot*, *snapshotDiff* for HDFS, and *query* for Ignite). The dimensions and APIs are selected based on their popularity, while SCALELENS users can select any

dimension or API of interest. Each workload comprises 80 – 100 lines of code in shell scripts and is reutilized for every version we test. The same workloads are given to SFIND and SCALEVIEW.

## Environment

All experiments are conducted on a single machine with 40-core Xeon Platinum 8380 CPU and 256 GB of RAM. All experiments, including the benchmark, are packaged into Docker images and are publicly available to ensure the reproducibility of the results.

### 5.3.2 Effectiveness and Comparison with Baseline

The goal of RQ1 is to evaluate the effectiveness of SCALELENS in detecting known scalability faults, and to provide a comparison with the baseline SFIND.

We evaluate SCALELENS on our benchmark of 55 real-world scalability faults drawn from our empirical study. For each fault, we examine whether SCALEVIEW detects the corresponding DCF, and whether SCALEPICK identifies the correct anti-pattern associated with that DCF. As summarized in Table 5.2, SCALEVIEW detects the correct DCFs for 38 out of 55 faults, including both explicit and implicit iterations. For each detected DCF, SCALEVIEW also correctly identifies the associated system dimensions (not shown in Table 5.2 due to space constraints). We found that all 17 cases in which SCALEVIEW fails to detect DCFs are related to multi-threading. For example, [HD-13768](#) [9] is a scalability fault where all threads are forced to wait for the slowest to complete, a scenario that requires deeper concurrency analysis. SCALEVIEW is unable to detect such instances whose root cause relates to the multi-threading strategy.

On top of the DCFs detected by SCALEVIEW, SCALEPICK successfully identifies anti-patterns for 36 out of 38 cases, demonstrating high effectiveness. In the remaining 2 cases, the DCFs are enclosed in inner classes implementing the `java.lang.Runnable` interface. Such classes are usually designed for multi-threading, being submitted to thread pools for

executors, and are not handled by the call graph analysis in SCALEPICK. We leave as future work the detection of scalability faults related to multi-threading.

We also compare SCALELENS and SFIND in their capability to detect DCFs (SFIND does not detect anti-patterns). As shown in Table 5.2, SFIND identified DCFs for only 9 out of the 55 faults, all of which involve explicit iterations. Compared to SCALELENS, SFIND detected 29 fewer DCFs. Even restricting our comparison to explicit iterations, SCALELENS still outperforms SFIND by detecting 18 more DCFs.

There are two reasons why SCALEVIEW, the dynamic analysis component of SCALELENS, outperforms SFIND in DCF detection. First, SFIND is designed to monitor the state of the system by performing heap measurement, which makes it unable to detect scale-dependent data structures that are not persistent when dimensions scale. In contrast, SCALEVIEW directly instruments the source code to obtain DCFs. Second, by design, SFIND only considers explicit application iterations, whereas SCALEVIEW takes into account both explicit application or library iterations and implicit iterations.

In addition to detecting more faults, SCALELENS provides richer diagnostic information. While SFIND only reports the presence of a scale-dependent loop, SCALELENS reveals (1) the system dimension associated with the DCF, and (2) the anti-pattern underlying the DCF. These insights are crucial for developers to understand the root cause of scalability faults.

**RQ1 Summary:** SCALELENS detects 29 more (4.2×) DCFs than the baseline, and fully identifies the DCFs and anti-patterns for 36 out of 55 previously known scalability faults. SCALELENS also reveals richer diagnostic information, such as the correlated system dimensions.

### 5.3.3 Finding Unknown Scalability Faults

In RQ1, we discussed the effectiveness of SCALELENS in detecting previously known scalability faults included in our empirical study. The goal of RQ2 is to evaluate the effectiveness of SCALELENS in detecting previously unknown scalability faults not present in our study.



Table 5.3: Ablation analysis of SCALELENS.

|                             | Cassandra  | HDFS       | Ignite     |
|-----------------------------|------------|------------|------------|
| # Fragments                 | 68,966     | 74,737     | 190,059    |
| # Fragments + Anti-Patterns | 7,823      | 16,982     | 12,467     |
| # DCFs                      | 251 (0.4%) | 106 (0.1%) | 342 (0.2%) |
| # DCFs + Anti-Patterns      | 129 (0.2%) | 68 (0.1%)  | 137 (0.1%) |

For that, we focus this evaluation on the latest stable versions of Cassandra (4.1.0), HDFS (3.4.0), and Ignite (2.16.0).

### Ablation Analysis & Manual Inspection

SCALELENS combines SCALEVIEW and SCALEPICK. To assess the contribution of each component, we analyzed the results from running SCALELENS on the latest stable versions of Cassandra, HDFS, and Ignite, as detailed in Table 5.3.

Without the analysis of SCALELENS, all loops and methods in the codebase are potential DCFs, leading to a large number of code fragments: 68,966 in Cassandra, 74,737 in HDFS, and 190,059 in Ignite.<sup>1</sup> Simply applying SCALEPICK to detect static code anti-patterns results in 7,823 to 16,982 code fragments, still making manual inspection impractical.

SCALEVIEW significantly reduces the number of code fragments of interest from 68,966 to 251 for Cassandra, representing 0.4% of the whole system. Out of the 251 DCFs pinpointed, SCALEPICK further identifies 129 that are associated with anti-patterns. Similarly, SCALEVIEW reduces the number of code fragments from 74,737 to 106 for HDFS, and from 190,059 to 342 for Ignite, with 68 and 137 DCFs containing anti-patterns, respectively. To confirm the correctness of the DCFs, we manually inspected all DCFs by plotting the iteration count to verify the increase of computational overhead along with the increase of the scale of the flagged dimension. For the DCFs with anti-patterns, we located the anti-patterns in the source code and confirmed their existence.

<sup>1</sup>Loops represent explicit iteration while a method called multiple times would capture implicit iteration.

## Scalability Faults in the Wild

SCALELENS identifies 129, 68, and 137 DCFs associated with anti-patterns in the latest versions of Cassandra, HDFS, and Ignite, respectively. From them, we found that 28 DCFs from Cassandra and 7 from HDFS are related to (de)serialization operations. Notably, all of them are (1) labeled as having linear growth to the dimension, and (2) identified with the *compute-cross* anti-pattern, which is correct as such operations are known to be IO-intensive due to their purposes. Such (de)serialization operations are often considered as necessary costs in distributed systems and are often unavoidable without major redesigns. As such, we exclude them from reporting. For the remaining DCFs, we manually confirmed that all of them exhibit problematic behavior and should be reported to developers.

To contribute to the effort of reporting previously unknown scalability faults, we have reported 27 DCFs with anti-patterns to the developers of Cassandra and HDFS. So far, the developers have confirmed problematic behavior for 5 of them and are investigating 4 others. The rest are pending review and none have been marked as invalid. We are in the process of reporting the rest of the scalability faults.

**RQ2 Summary:** By combining dynamic and static analyses, SCALELENS enables efficient detection of scalability faults. SCALELENS has detected previously unknown faults in the latest stable versions of Cassandra, HDFS, and Ignite, resulting so far in 27 reports to developers, with 5 confirmed and 4 under investigation.

## 5.4 Threats to Validity

### 5.4.1 Construct Validity

The central constructs of our work are *scalability fault*, *scalable system dimension*, *DCF*, and *anti-pattern*. We defined each at first use and anchored our definitions in prior literature where possible: scalability fault follows Leesatapornwongsa et al. [211], and scalable system dimensions follow the scalability characterization framework of Duboc et al. [131, 132]. Alternative operational definitions are possible. To guard against inconsistency, we applied the

resulting taxonomy uniformly across all 444 faults and released the full catalog of labeled issues for independent audit.

We use iteration count as the observable proxy for scale-dependent execution cost. However, some scale-sensitive costs do not manifest as loops, such as a recursive descent whose depth correlates with a dimension, or a chain of conditional branches whose path space grows with cluster size. To reduce the risk of missing such cases, SCALEVIEW tracks not only explicit iteration but also implicit iteration, where the number of method invocations across homogeneous components grows with the system’s scale (Section 5.2.1).

Last, a construct-validity concern is the fidelity of the manually authored scaling workloads. A workload is intended to drive one or more dimensions to a large value while exercising representative APIs; a badly designed workload may fail to do so. We mitigate this by requiring workloads to report their current scale state through a `set-scale-state` API provided by SCALEVIEW (Section 5.2.1), which gives SCALEVIEW a runtime signal for whether the intended dimension is actually being scaled, and by releasing the full set of workloads used in our evaluation for independent inspection.

### 5.4.2 Internal Validity

Our empirical findings rest on the manual selection and categorization of 444 scalability faults from over 199K issue reports. This manual process can introduce bias. We mitigate this with a multi-stage adjudication protocol: every issue is independently reviewed and tagged by two authors, a third author reviews every disagreement or uncertainty, and unresolved conflicts are settled in group discussion. We further mitigate the threat by publishing the full set of tagged issues and the tagging guidelines.

We also considered keyword-based selection as an alternative to this manual process. A broad keyword search over the 199K issues, using terms such as “scale”, “scalability”, “cluster”, “size”, “degradation”, “contention”, “out of memory”, “performance”, “regression”, and “impact”, returned 10,046 candidates, of which manual inspection revealed a 99% false-

positive rate. Backward validation against the manually selected 444 faults showed that 78% of them contain none of those keywords. Automating selection by keyword would therefore have both over-reported and under-reported scalability faults, and the manual process was the only reliable option available.

A further internal-validity concern is the inherent nondeterminism of dynamic analysis: the JIT compiler, garbage collector, and operating system scheduler introduce run-to-run variation in SCALEVIEW’s recorded iteration counts. We mitigate this by filtering growth trends across multiple scale states rather than relying on single-point measurements (Section 5.2.1), so that the correlation between iteration count and scale state is robust to local noise.

### 5.4.3 External Validity

The 10 systems in our empirical study (Section 5.1) are all Java-based and span databases, file systems, compute frameworks, streaming platforms, and coordination services. This covers the most common categories of open-source distributed systems, but our findings may not transfer to non-JVM distributed systems (e.g., C++, Go, or Rust) or to application domains with substantially different scaling profiles, such as HPC-style tightly coupled compute.

SCALELENS has two components with different generalization properties. SCALEVIEW instruments code at the JVM bytecode level and is therefore applicable to any Java-based system without source modification. SCALEPICK, in contrast, depends on static call-graph analysis, which is accurate for code with conventional control flow (explicit method calls, inheritance, explicit locks) but is less precise on event-driven, callback-heavy, reflective, or executor-dispatched code, where the call graph is incomplete or statically unresolvable. In our evaluation, for instance, two anti-pattern false negatives trace to `Runnable`-based executor code, which our analysis does not follow into the thread pool. In practical terms, SCALELENS is most effective on Java batch, storage, and database systems such as Cassandra, HDFS, and Ignite, where control flow and synchronization relationships are largely

recoverable from the source. Extending SCALELENS to event-driven architectures would likely require hybrid static-dynamic call-graph recovery, which we leave as future work.

SCALELENS’s detection performance additionally depends on the coverage of the supplied scaling workloads. A DCF is detected only if the relevant dimension is exercised, so workloads that omit a dimension will not surface DCFs that depend on it. This is a generalization threat rather than a tool bug. Users of SCALELENS can mitigate it by authoring workloads that jointly scale the dimensions they care about, and by reusing workloads across versions; in our evaluation the same workloads were reused across 7 versions of Cassandra, 5 of HDFS, and 3 of Ignite at no additional cost.

For the tool evaluation, our benchmark comprises 55 scalability faults drawn from Cassandra, HDFS, and Ignite, which together account for 217 (48.9%) of the faults in our study. This gives substantial coverage but is not exhaustive: the remaining seven studied systems and fault categories outside *compute* and *unbound* are not represented in the benchmark. We additionally explored using the 10-bug benchmark from SCALECHECK [333], but those systems (Cassandra 1.1.x and 1.2.x, HDFS 2.0.0) were released before April 2013 and are no longer buildable due to outdated dependencies; we therefore report numbers only against our own benchmark.

Finally, the taxonomy itself carries an external-validity concern. The 4 root-cause categories and the 11 anti-patterns emerged from 444 faults in 10 systems, so additional systems or application domains, particularly outside the Apache Java ecosystem, may surface different categories or anti-patterns.

#### 5.4.4 Conclusion Validity

SCALEVIEW finds DCFs as system facts: a DCF is by construction an iterative code fragment whose iteration count correlates with a scaled dimension, so SCALEVIEW produces no false positives for DCF identification. SCALEVIEW may, however, miss DCFs whose dimension is not exercised by the supplied workload, a source of false negatives already discussed above.

SCALEPICK flags a DCF with an anti-pattern only when its slice contains a checker-specific sink (Section 5.2.2). We manually verified every flagged anti-pattern on our benchmark and observed no false positives. The two anti-pattern false negatives on known faults both trace to Runnable-based executor code, tying back to the static-analysis limitation noted under External Validity.

Our headline effectiveness numbers, 36 fully detected and 2 partially detected out of 55 known scalability faults and 4.2× more DCFs than SFIND, are reported against this benchmark. They support our claims about SCALELENS’s detection capability on the studied fault categories and systems, but should not be read as universal guarantees across arbitrary distributed systems. The 334 DCFs with anti-patterns reported on the latest stable versions of Cassandra, HDFS, and Ignite were individually verified by manual inspection, so the positive count is reliable, but the proportion that developers will ultimately confirm as problematic is still evolving as reports progress through upstream review.

## 5.5 Related Work

SCALECHECK [333] discovers scalability bugs in large-scale distributed systems via two components: SFIND, a program analysis tool that identifies scale-dependent loops (those whose number of iterations grow alongside system-scale data structures), and STTEST, a set of colocation techniques that emulate a real-scale cluster on a single machine. SCALELENS differs from SCALECHECK in three key ways. First, SCALEVIEW generalizes scale-dependent loops into DCFs by capturing implicit iteration, even when no explicit loop exists in the source. Second, while SFIND tracks heap-allocated collections to find loops, SCALEVIEW instruments bytecode directly; this allows it to detect DCFs involving transient data structures that a heap trace would miss. Finally, whereas SCALECHECK relies on STTEST to exercise the identified loops through cluster emulation, SCALELENS provides SCALEPICK, a set of static checkers to identify anti-patterns known to incur in scalability issues when combined

with DCFs. These approaches are complementary: SCALEPICK offers immediate root-cause analysis, while STEST reproduces problematic symptoms at runtime.

Other tools also follow this symptom-oriented emulation approach [160, 237, 369, 420]. For example, EXALT [369] aims to emulate hundreds of HDFS nodes in a single machine to observe I/O-related increases in processing time, while DIECAST [160] aims to emulate large networks using a single machine and observe how networking speed affects processing time. In contrast, SCALELENS does not require explicit symptoms nor emulation techniques as it detects unknown scalability faults based on a combination of DCFs and anti-patterns.

Other works utilize extrapolation [189, 203, 319, 449, 452] to project system behavior (e.g., execution time) at scale using smaller-scale measurements. PATTERNMINER [319] identifies scalability bottlenecks in centralized distributed systems (e.g., HDFS) by detecting repeating behavior and noting that such behavior repeats at larger scales. VRISHA [449], SCALANA [189], AUTOMADED [203] and [452] model the scalability of HPC applications combining smaller-scale behavior observations with techniques such as canonical correlation analysis, performance graphs, stack trace analysis and machine learning, respectively. SCALELENS differs in that it requires no specification of symptoms/behaviors, is architecture-agnostic, and focuses on distributed systems.

Finally, other works present techniques for detecting memory bloat [96, 183, 213, 251, 265], resource leaks [193, 314, 359, 392, 393], thread contention [65, 67, 215, 224, 411, 422, 450], chatty I/O [37, 78, 288, 402], and performance anti-patterns [104, 108, 214, 269, 270, 286, 316, 323, 324, 327, 352, 437]. We consider all these works orthogonal to SCALELENS, as they are limited to design flaws regardless of system scale. SCALELENS, however, focuses on scalability faults unique to large-scale distributed systems, identifying anti-patterns whose negative impact remains latent at small scales but manifests at large scales due to growth in one or more system dimensions.

## Chapter Summary

Scalability faults in distributed systems are not arbitrary. The empirical study in this chapter shows that the majority follow a dimensional code fragment plus anti-pattern signature, where a code fragment whose execution count grows with one of four scaling dimensions (load, data, cluster size, or failure rate) is paired with one of a small set of specific anti-patterns. SCALELENS operationalizes that signature: it locates dimensional code fragments via lightweight dynamic analysis on small inputs, then matches them against the anti-patterns via static analysis. Across the known faults we used for evaluation, SCALELENS detects substantially more dimensional code fragments than the memory-monitoring baseline. Applied to the latest stable releases of Cassandra, HDFS, and Ignite, it surfaces hundreds of latent issues that we confirmed by manual inspection. [Chapter 6](#) extends this perspective to a different class of system, modern LLM-serving infrastructure, where similar fault patterns appear together with new scaling dimensions specific to AI workloads.

This chapter is based on the paper “Understanding and Detecting Scalability Faults in Large-Scale Distributed Systems,” co-authored with Goodness Ayinmode, Cesar A. Stuardo, Haryadi S. Gunawi, and Cindy Rubio-González. The work was supported by NSF grants CCF-2119348 and CCF-2119184. The empirical data and analysis are available at <https://github.com/ucd-plse/scalability>, and SCALELENS is available at <https://github.com/ucd-plse/scalelens>.



# Chapter 6

## Toward Scalable LLM Serving: Bottlenecks and New Dimensions

Large language models (LLMs) have rapidly become the back-end of production services that span search, coding assistance, customer support, and agentic workflows. In these settings, the model is only one component of a larger online system, and deploying an LLM as a production service requires an *LLM serving infrastructure*: the software stack that accepts requests, schedules them into batches, manages KV-cache memory, routes traffic across replicas, and coordinates GPU workers across one or more nodes.

As deployments scale, this infrastructure is stressed along several dimensions at once: concurrent requests grow, contexts become longer, multimodal payloads increase per-request state, and model execution spans more GPUs and nodes. These pressures shift the reliability problem from the model to the serving system itself. A deployment can use a correct model and still fail because a router performs an  $O(N)$  scan over replicas, a cache is never reclaimed, a kernel assumes a small batch size, or an inter-node transfer path was never tested beyond a few ranks. These are not ordinary functional bugs; they are failures whose trigger is scale.

*Scalability faults* are latent at small scale and manifest only when an operational dimension grows past a threshold [211, 333, 458]. This definition makes the fault class difficult to

detect and diagnose. Unit tests, single-machine integration tests, and short-duration load tests often execute the relevant code paths below the threshold at which the fault becomes observable. Moreover, each fault is tied to a *scaling dimension*: the operational parameter whose growth triggers the failure. The same code can be correct at one operating point and incorrect at another, with no change in the code itself.

Prior work leaves this fault class poorly characterized for LLM serving infrastructures. Bug studies of classical JVM-era distributed systems (e.g., CbsDB [159], TaxDC [210], and crash-recovery analyses [146]) predate the LLM era and primarily study storage, data-processing, and coordination paths. LLM serving inherits some of these dimensions, such as client count, cluster size, data volume, and cumulative request history [159, 458], but it also introduces GPU memory hierarchies, KV-cache lifecycle, kernel grid limits, multimodal inputs, and multi-node model parallelism. Recent studies of bugs in DL frameworks and LLM inference engines [179, 227] catalog the full bug population, but they do not condition on scale. Consequently, they do not explain which scaling dimensions trigger LLM-serving faults, what symptoms operators observe, which root causes dominate, or which faults are fixable in practice.

To fill this gap, we characterize scalability faults across three architecturally diverse systems chosen to span the LLM serving spectrum: Ray Serve [254], a general-purpose serving framework focused on replica orchestration and routing; vLLM [202], a paged-attention inference engine focused on GPU execution and KV cache management; and SGLang [442], a structured-generation runtime that combines RadixAttention prefix sharing with multi-node prefill-decode disaggregation. We mine 56,456 issue reports across these three repositories, apply two-pass manual inspection with independent double-coding by two authors, and identify 109 scalability faults. We classify each fault along four orthogonal axes, organized by the following research questions:

**RQ1 Scaling Dimensions** (Section 6.3). Which scaling dimensions trigger scalability faults in LLM serving infrastructures, and how does the dimension profile differ across

architectural roles?

**RQ2 Symptoms** (Section 6.4). What externally observable symptoms accompany scalability faults, and how does the symptom profile differ by architectural role?

**RQ3 Root Causes** (Section 6.5). What recurring root-cause patterns explain scalability faults, and how do those patterns relate to scaling dimensions and symptoms?

**RQ4 Fix Strategies** (Section 6.6). How are scalability faults fixed in practice, what fix patterns recur, and which faults remain stubbornly open?

The analysis yields three connected findings. First, scalability faults bifurcate into two architecturally disjoint regimes that we call the *two-plane pattern*. Ray Serve faults concentrate in control-plane scaling: management operations,  $O(N)$  scans over global state, single-actor proxy hot paths, and unbounded telemetry queues. vLLM and SGLang faults concentrate in data-plane scaling: GPU memory lifecycle, kernel boundary conditions, and inter-node communication assertions. The bifurcation persists across all four axes (dimensions, symptoms, root causes, and fixes), and the two profiles are nearly disjoint at the categorical level even though all three systems serve the same workload at the same operating point. This means that testing strategies inherited from either classical distributed systems or inference-engine kernel testing alone are structurally incomplete for the full LLM serving stack.

Second, these faults are difficult to repair once exposed. 54.1% have no code-level resolution at the time of analysis, and the unfixed rate is concentrated in the most common root causes (Memory Mismanagement 82.6%, Distributed System Fault 66.7%). The unfixed rate is comparable across systems (Ray Serve 47.9%, SGLang 67.9%, vLLM 51.5%), indicating that the difficulty is structural to the fault class, not specific to any single maintainer community.

Third, despite this overall repair difficulty, one root-cause category is unusually preventable. Configuration & Validation Defects, though only the third most frequent root cause (15.6%), admit the simplest patches and have a comparatively high fix rate (47.1%):

a one-line formula correction (e.g., `max()` replaced with `min()` when computing available KV slots), a configurable parameter exposed in place of a hard-coded constant, or a startup assertion that rejects an unsupported feature combination. Because this category is distributed evenly across all three systems, systematic startup validation is a high-leverage intervention rather than a system-specific concern. In short, LLM-serving scalability faults are architecturally patterned, hard to repair after deployment, and partly preventable through validation.

These findings motivate a substantive shift in how scalability faults are tested and repaired in LLM serving infrastructures. The two-plane pattern argues that test infrastructure should explicitly target both planes (cardinality sweeps for the control plane; soak, multi-node, and boundary-value tests for the data plane). The unfixed-rate concentration argues that memory lifecycle and distributed coordination need first-class engineering attention, not the incremental patching they receive today. The leverage of Configuration & Validation Defects argues that a fail-fast validation discipline at engine startup is the single change with the largest expected reduction in production scalability faults.

In summary, this chapter makes the following contributions:

- We conduct the first empirical study of code-level scalability faults in LLM serving infrastructures, deriving a four-axis taxonomy from manual analysis of 109 real-world faults across Ray Serve, vLLM, and SGLang.
- We identify a *two-plane pattern* in which faults bifurcate into a control-plane regime in the serving framework and a data-plane regime in the inference engines, with near-disjoint profiles across all four axes.
- We surface cross-axis correlations that prescribe the repair approach (e.g., Concurrency Bug → Concurrency Hardening in 76.9%) and identify Configuration & Validation Defects (15.6%) as the highest-leverage repair target.
- We release a dataset of 109 LLM-serving scalability faults, including manual labels and

code-level fixes where applicable.

## 6.1 Background

LLM inference is autoregressive: a Transformer [357] predicts one token at a time, conditioned on the prompt and previously generated tokens. The process unfolds in a compute-bound *prefill* phase that consumes the prompt in a single forward pass, followed by a memory-bandwidth-bound *decode* phase that emits one token per step [63, 408, 448]. A serving stack connects user requests to GPU execution through a pipeline of software components. Requests first enter a frontend, which receives HTTP or RPC calls from clients, and then pass to a router, which chooses the model replica or worker that should handle each request. Before execution, a scheduler groups requests into batches and decides whether there is enough KV-cache memory to admit more tokens. The admitted batches then run on GPU worker processes, each of which executes the model on one or more GPUs. When a model spans multiple GPUs, these workers exchange data over fast links within a machine (such as NVLink) or over network fabrics across machines (such as RDMA). Because these stages run concurrently, they communicate through message queues, remote procedure calls, or shared object stores, often asynchronously so that one layer can continue working while another is waiting. Continuous batching, KV caching, and prefix caching sustain throughput across heterogeneous concurrent requests. Throughout the paper, we distinguish the *control plane* from the *data plane*. The control plane manages serving metadata and orchestration, such as deployments, replicas, routing handles, autoscaling decisions, and metrics. The data plane executes inference requests, including scheduling, GPU-kernel execution, KV-cache allocation, and inter-GPU or inter-node data movement.

We focus on three systems whose source repositories together capture the architectural spectrum of the LLM serving stack.

**Ray Serve** [254] Ray Serve is a general-purpose serving framework on the Ray distributed runtime. It models the stack as a graph of *actors*, long-lived Python processes that own state, and handles replica management, routing, dynamic batching, autoscaling, and backpressure at the framework level. Its scalability faults primarily arise in the *control plane*: the management of replicas, deployments, and routing handles, not the GPU forward pass.

**vLLM** [202] vLLM is an LLM inference engine built around *PagedAttention*, a paging-inspired KV-cache allocator that reduces fragmentation and enables flexible memory sharing across concurrent sequences. Its architecture splits the inference engine into an HTTP API process, an engine-core process (scheduler and KV manager), and per-GPU worker processes connected via ZMQ. Its scalability faults primarily arise in the *data plane*: GPU memory, kernel execution, and request scheduling.

**SGLang** [442] SGLang is a structured-generation runtime built around *RadixAttention*, which organizes KV caches in a radix tree to enable prefix reuse, and a Rust-based router for production deployment. It additionally supports prefill–decode disaggregation, in which prefill and decode stages run on separate GPU sets that exchange KV-cache state over an inter-node fabric. Because these distributed serving mechanisms are built into the runtime, SGLang’s data-plane faults include KV-transfer and inter-node coordination paths in addition to GPU memory and kernel execution.

## 6.2 Study Methodology

Our methodology follows established empirical software engineering guidelines [302] and applies open-coding principles [332] to derive a taxonomy of scalability faults from real-world issue reports.

**System Selection** We selected three widely deployed LLM serving systems: Ray Serve (RS), SGLang (SG), and vLLM (VL). Three criteria motivated this selection. First, the three systems are among the most widely adopted open-source LLM serving systems, with large user bases and extensive production deployments across industry. Second, they cover complementary architectural layers of the serving stack. Third, all three have active development communities with substantial issue histories on GitHub, providing a rich corpus of real-world scalability reports suitable for empirical analysis.

**Issue Collection and Filtering** We collected all issue reports and pull requests from the GitHub repositories of the three target systems, from the inception of each repository through January 31, 2026, totaling 56,456 reports. We then applied system-specific label and keyword filters to retain only bug-related reports and exclude unrelated items such as feature requests, documentation updates, and CI tasks. Because Ray Serve is a component of the broader Ray project, we first filtered its repository to issues tagged with the `serve` label or containing `[Serve]` in the title. For Ray Serve, we selected issues labeled `bug`, `enhancement`, `stability`, or `performance`, because scalability limitations in serving frameworks are frequently filed as enhancement requests rather than bugs. For vLLM, we selected issues labeled `bug` or `performance`. For SGLang, whose issue tracker uses labels sparsely, we selected issues labeled `bug` or whose titles contain bug-related keywords (e.g., “fix,” “crash,” “OOM,” “hang,” “error,” “leak”). This filtering step yielded 16,756 candidate reports.

**Manual Inspection** We performed a two-pass manual inspection of the 16,756 candidate reports. In the first pass, we read the title and description of each report to identify issues that involve growth in any operational scaling dimension of the system, such as concurrency, request volume, prompt length, replica count, parallelism degree, or cumulative request history. In the second pass, we performed a more detailed review of the first-pass candidates by examining the discussion threads, linked pull requests, referenced source code, and reproduction snippets. We categorized each candidate as a scalability fault when its manifestation

Table 6.1: Summary of issue collection pipeline per system.

| System       | Total Issues | After Filtering | Scalability Issues |
|--------------|--------------|-----------------|--------------------|
| Ray Serve    | 5,882        | 1,655           | 48                 |
| SGLang       | 17,706       | 7,176           | 28                 |
| vLLM         | 32,868       | 7,925           | 33                 |
| <b>Total</b> | 56,456       | 16,756          | 109                |

depended on the growth of one or more dimensions defined in Section 6.3, and we excluded candidates whose failure was independent of scale. This process identified 109 scalability faults across the three systems, summarized in Table 6.1.

**Pattern Analysis and Adjudication** We analyzed the 109 scalability faults to characterize each along four orthogonal axes: scaling dimension, symptom, root cause, and fix strategy. This process consisted of examining the full content of each issue report, discussion threads, source code, available patches, and linked pull requests. To minimize subjective bias, we employed a multi-stage adjudication protocol. First, two authors independently labeled each fault along all four axes. Second, the two authors compared labels for every fault. For cases where authors disagreed or expressed uncertainty, the two authors met to discuss the case in light of the evidence in the issue report and source code, and converged on a single label by consensus. A third author was consulted for cases where consensus could not be reached after discussion. This adjudication procedure follows the protocol used in prior empirical bug studies of distributed systems [159, 458] and DL frameworks [179, 227].

## 6.3 Scaling Dimensions

To understand which scaling dimensions most commonly trigger scalability faults in LLM serving infrastructures, we categorize each of the 109 faults by the operational parameter whose growth beyond an operational threshold causes the fault to manifest. Through open coding of the issue reports, we derive a taxonomy of six scaling dimensions that cover all faults in our dataset. Their distribution across the three systems is shown in Table 6.2.



Table 6.2: Distribution of scalability faults by scaling dimension.

| Scaling Dimension      | Ray | SGL | vLLM | Total | %     |
|------------------------|-----|-----|------|-------|-------|
| Request Load           | 18  | 14  | 14   | 46    | 42.2% |
| Data Scale             | 5   | 6   | 9    | 20    | 18.3% |
| Serving Infrastructure | 18  | 0   | 0    | 18    | 16.5% |
| Temporal Accumulation  | 2   | 2   | 7    | 11    | 10.1% |
| Cluster & Parallelism  | 2   | 5   | 1    | 8     | 7.3%  |
| Model Complexity       | 3   | 1   | 2    | 6     | 5.5%  |
| <b>Total</b>           | 48  | 28  | 33   | 109   | 100%  |

**D1: Request Load** The number of concurrent requests, the request arrival rate, or the scheduling batch size grows. This dimension captures growth in queue depth, in-flight sequences, and batched-token count. For example, in [SGL-4020](#), a 500-concurrent-request stress test against an 8-GPU H200 deployment leaves seven GPUs saturated while one sits idle, and requests soon return empty 200 responses. Request Load contributes the largest share of faults in every system (18 in Ray Serve, 14 in vLLM, 14 in SGLang), but the manifestations differ: proxy bottlenecks and routing-level races in Ray Serve, KV-cache exhaustion and scheduler-level hangs in vLLM, and KV-transfer channel and request-pool races in SGLang.

**D2: Data Scale** The volume of data associated with individual requests grows. This includes prompt and context length (in tokens), the count and resolution of multimodal inputs (images, video, audio), output token length, and raw payload size in bytes. Unlike Request Load, Data Scale increases the cost of each request in compute and memory. As a concrete example, in [vLLM-32680](#), the engine hangs at a 256K-token video-understanding context because the encoder cache is sized smaller than the actual number of video tokens; `num_new_tokens` returns zero on every scheduling step and the decoder stalls indefinitely. The 20 Data Scale faults include 9 in vLLM, 6 in SGLang, and 5 in Ray Serve. In inference engines, input growth pushes GPU memory or kernel limits; in Ray Serve, the same dimension manifests as serialization overhead and gRPC message-size limits.

**D3: Serving Infrastructure** The number of software-level management entities in the serving framework’s control plane grows: deployments, applications, replica actors, routing handles, and telemetry streams. Equivalently, this dimension captures control-plane management scale. This dimension is architecturally distinct from Cluster & Parallelism (D5): it concerns the overhead of orchestrating and routing to service instances, not the physical distribution of a model’s computation. Consider [Ray-45793](#): the controller’s `_get_live_deployments` function iterates over every live deployment to filter for one application, saturating a CPU core when many applications are running. All 18 Serving Infrastructure faults originate from Ray Serve; neither vLLM nor SGLang exposes the controller, proxy, deployment-handle, or replica-scheduler abstractions involved in this dimension.

**D4: Temporal Accumulation** The cumulative number of requests processed over the system’s lifetime grows, and resources that should be transient accumulate instead. Unlike Request Load, which captures instantaneous concurrency, Temporal Accumulation captures faults that emerge only after sustained operation. For example, in [vLLM-31577](#), GPU VRAM usage climbs monotonically while serving a Whisper model and never returns to baseline between requests, eventually causing OOM despite no single request exceeding the per-step memory budget. Of the 11 faults in this dimension, 7 originate from vLLM, reflecting the complexity of its memory management (encoder caches, KV blocks, Triton JIT caches); the remaining 4 are split between SGLang (2) and Ray Serve (2).

**D5: Cluster & Parallelism** The number of physical devices (GPUs, nodes) grows, or a model’s execution is partitioned across more parallelism modes (tensor, pipeline, data, or expert parallelism; prefill–decode disaggregation). This dimension captures faults that arise from topology-dependent assumptions in communication libraries, initialization protocols, and inter-device synchronization. Unlike D3, this dimension concerns how a single model’s execution is partitioned across hardware. For instance, in [SGL-5596](#), deploying DeepSeek-R1 on 16 nodes (128 H200 GPUs) with shared-experts fusion fails during model loading

while the same configuration works on 4 nodes (32 GPUs). Of the 8 Cluster & Parallelism faults, 5 originate from SGLang, reflecting its emphasis on prefill–decode disaggregation and multi-node distributed execution; the remaining faults occur in Ray Serve (2) and vLLM (1).

**D6: Model Complexity** The model’s architectural complexity or the number of concurrently active model variants grows: experts in a Mixture-of-Experts (MoE) architecture, concurrent LoRA adapters, or multiple distinct models on shared hardware. [vLLM-32915](#) illustrates this: enabling LoRA with `--max-loras=10` causes the engine to fail at initialization with OOM; vLLM pre-allocates GPU memory proportional to the configured LoRA capacity rather than to the adapters actually in use. The 6 Model Complexity faults span all three systems and frequently arise from combinatorial interactions with other dimensions (e.g., expert parallelism amplifies kernel grid-limit pressure).

**RQ1 Summary:**

- Request Load is the universal scaling trigger across all three systems, accounting for 42.2% of faults with comparable per-system shares (18 in Ray Serve, 14 in vLLM, 14 in SGLang); concurrency is the one dimension whose fault profile is architecturally insensitive.
- Other dimensions cluster sharply by architectural role: Serving Infrastructure is exclusive to Ray Serve (100%), Temporal Accumulation concentrates in vLLM (63.6%), and Cluster & Parallelism concentrates in SGLang (62.5%); each system’s dominant non-Request-Load dimension matches the subsystem most exposed to growth (control-plane management, GPU memory, or multi-node coordination).
- Model Complexity is the rarest dimension (5.5%) but consistently produces the highest-severity faults: most are initialization-time failures (LoRA pre-allocation OOM, MoE kernel boundary violations) that prevent the engine from starting at all.

## 6.4 Symptoms

We classify each scalability fault by its primary externally visible symptom. Open coding yields seven symptom categories, whose distribution across systems is shown in Table [6.3](#).

Table 6.3: Distribution of scalability faults by symptom category.

| Symptom Category          | Ray | SGL | vLLM | Total | %     |
|---------------------------|-----|-----|------|-------|-------|
| Resource Exhaustion       | 5   | 10  | 9    | 24    | 22%   |
| Liveness Failure          | 8   | 5   | 5    | 18    | 16.5% |
| Control-plane Degradation | 15  | 2   | 0    | 17    | 15.6% |
| Throughput Degradation    | 10  | 0   | 6    | 16    | 14.7% |
| Hard Failure              | 5   | 7   | 4    | 16    | 14.7% |
| Latency Inflation         | 3   | 1   | 5    | 9     | 8.3%  |
| Correctness Violation     | 2   | 3   | 4    | 9     | 8.3%  |
| <b>Total</b>              | 48  | 28  | 33   | 109   | 100%  |

**S1: Resource Exhaustion** GPU or CPU memory is exhausted at startup or at runtime, memory usage grows without bound, or allocated memory fails to return to baseline after requests complete. The category includes both acute OOM events and chronic leaks. In [SGL-13940](#), free GPU memory keeps decreasing while serving Qwen3-VL-8B-Instruct-FP8 image requests; intermediate tensors from the vision encoder are not fully released, and the process eventually crashes with CUDA OOM. Resource Exhaustion is the most frequent symptom (22%) and is dominated by inference engines (10 in SGLang, 9 in vLLM); in Ray Serve its 5 instances are control-plane leaks rather than GPU memory pressure.

**S2: Liveness Failure** The system stops making forward progress: the server hangs, requests stall, health endpoints become unresponsive, or a queue blocks indefinitely. Unlike crashes or fatal errors, the process remains alive but ceases to produce useful output. For instance, in [vLLM-30343](#), under high load the API server hangs and even `/health` stalls because a per-token `detokenize_incrementally` call starves the event loop. Liveness Failure faults are split nearly evenly across systems (8 in Ray Serve, 5 each in vLLM and SGLang), arising from architecturally unrelated pathways: cancellation races and unbounded queues in Ray Serve, event-loop starvation and kernel-level hangs in vLLM, and KV-transfer deadlocks in SGLang.

**S3: Control-plane Degradation** The serving framework’s management plane becomes the bottleneck: controller CPU saturates, control-loop iterations slow to seconds, metrics-push tasks accumulate unboundedly, log volume floods storage, autoscaling decisions mis-

behave, or deployment-management operations take disproportionately long. For example, in [Ray-45792](#), the Serve controller exceeds 100% CPU utilization because `get_deployment_statuses` iterates over every deployment to obtain status for a small subset. Of the 17 Control-plane Degradation faults, 15 originate from Ray Serve and 2 from SGLang; vLLM exhibits none.

**S4: Throughput Degradation** The system remains responsive but cannot scale: the request rate stops increasing as replicas increase, token throughput drops sharply, GPUs sit idle while requests queue, or adding hardware fails to improve aggregate capacity. The defining feature is a *sublinear or flat scaling curve*: the system is live, but end-to-end throughput does not grow with resources. As a concrete example, in [Ray-43549](#), the proxy actor becomes the throughput ceiling because each request triggers three remote calls to replicas. Throughput Degradation is concentrated in Ray Serve (10 of 16) and vLLM (6 of 16); SGLang exhibits none.

**S5: Hard Failure** The server process crashes, a CUDA kernel triggers an illegal memory access, an assertion fires, or the OS terminates the process (e.g., OOM killer on host memory, segmentation fault). Unlike Resource Exhaustion, the immediate manifestation is process termination or a fatal exception, not a gradual memory climb. Also unlike Liveness Failure, the process does not remain alive while stalled. [SGL-7545](#) illustrates this: a CUDA kernel crashes when the effective batch size exceeds a GPU-kernel grid limit of 65,536. Hard Failure is concentrated in SGLang (43.8%), reflecting kernel and distributed-execution boundary conditions that cross only at large scale.

**S6: Latency Inflation** Per-request latency degrades with scale: time-to-first-token (TTFT) increases, inter-token latency (ITL) increases, end-to-end response time climbs, or clients begin timing out. Unlike Throughput Degradation, which is measured at the system level, Latency Inflation is measured at the individual-request level. In [Ray-46428](#), sending a single

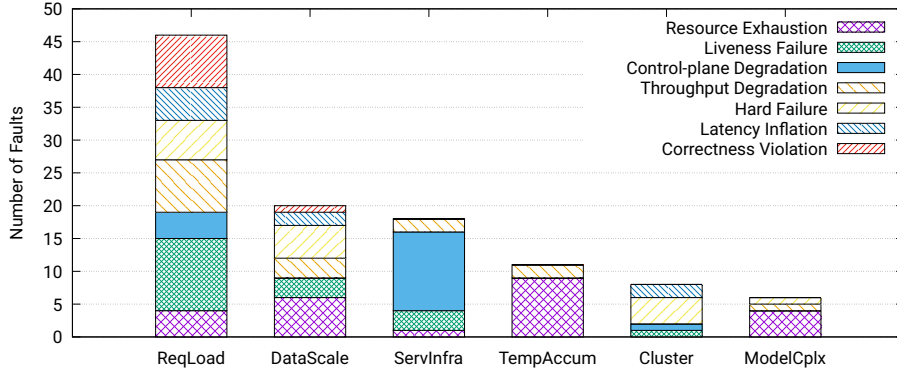


Figure 6.1: Distribution of symptom categories across scaling dimensions.

1 MB payload through Serve regresses end-to-end latency from 3.4s to 4.6s; the overhead is in `Router._resolve_deployment_responses`, where nested `pickle.dump/pickle.load` calls scale linearly with payload size. Latency Inflation accounts for 9 of 109 faults.

**S7: Correctness Violation** The system returns a response, but the response is wrong: outputs are corrupted, non-deterministic across identical inputs, truncated or empty, or the streaming protocol is violated. The category also includes semantic contract violations such as generating far more tokens than requested or returning stale cached results. For example, in [SGL-17850](#), chunked prefill batches assign the same `req_to_token_pool` slot to two concurrent requests; the scheduler’s write of KV cache indices overwrites data being read by the GPU in the forward pass. Correctness Violation accounts for 9 of 109 faults.

Across dimensions (Figure 6.1), Request Load is the most heterogeneous trigger, with 46 faults spanning all seven symptoms. The other dimensions are more concentrated: Temporal Accumulation maps mostly to Resource Exhaustion (9 of 11), Serving Infrastructure maps to Control-plane Degradation (66.7%), and Cluster & Parallelism maps mainly to Hard Failure (4 of 8).

Table 6.4: Distribution of scalability faults by root cause category.

| Root Cause Category               | Ray | SGL | vLLM | Total | %     |
|-----------------------------------|-----|-----|------|-------|-------|
| Memory Mismanagement              | 4   | 6   | 13   | 23    | 21.1% |
| Control-plane Bottleneck          | 16  | 2   | 0    | 18    | 16.5% |
| Configuration & Validation Defect | 8   | 4   | 5    | 17    | 15.6% |
| Request Processing Bottleneck     | 11  | 1   | 4    | 16    | 14.7% |
| Concurrency Bug                   | 7   | 4   | 2    | 13    | 11.9% |
| Kernel & Accelerator Defect       | 1   | 4   | 8    | 13    | 11.9% |
| Distributed System Fault          | 1   | 7   | 1    | 9     | 8.3%  |
| <b>Total</b>                      | 48  | 28  | 33   | 109   | 100%  |

**RQ2 Summary:**

- Resource Exhaustion is the most frequent symptom (22%) and is concentrated in inference engines (79.2% from vLLM and SGLang); Ray Serve exhibits Resource Exhaustion through control-plane leaks rather than GPU memory pressure.
- Liveness Failure is the only symptom uniformly distributed across systems (44.4% Ray Serve, 27.8% vLLM, 27.8% SGLang); the same external symptom emerges from unrelated pathways (cancellation races, event-loop starvation, KV-transfer deadlocks).
- SGLang exhibits zero Throughput Degradation faults: it either works or fails catastrophically (crash, hang, OOM), without the gradual performance erosion seen in Ray Serve and vLLM.
- The symptom profile bifurcates along the two-plane pattern: Ray Serve faults concentrate in control-plane symptoms (31.2% Control-plane Degradation, 20.8% Throughput Degradation), while vLLM and SGLang faults concentrate in data-plane symptoms (Resource Exhaustion, Hard Failure, Liveness Failure; 65.6%).

## 6.5 Root Causes

We classify each scalability fault by its root cause: the code-level or design-level reason the fault exists. The same scaling dimension can trigger different causes, and the same cause can manifest as different symptoms. Open coding yields seven root-cause categories, whose distribution across systems is shown in Table 6.4.

**R1: Control-plane Bottleneck** A management-plane component (e.g., controller, proxy, metrics aggregator, deployment manager) sits on a hot path whose cost grows with the number of managed entities (deployments, handles, replicas, applications). Concrete mechanisms include  $O(N)$  scans over global state, metrics push tasks that accumulate unboundedly,

hot-path logging, and sequential lifecycle operations that serialize parallelizable work. For example, in [Ray-45777](#), metrics-push tasks accumulate on the Serve controller because every `DeploymentHandle` pushes its own metrics; the controller CPU saturates as the number of handles grows. Of the 18 faults in this category, 16 originate from Ray Serve; the two SGLang instances arise from health-check bypass and abort logging.

**R2: Request Processing Bottleneck** The request-handling pipeline contains a stage whose cost grows faster than throughput capacity: a centralized proxy that must touch every request, CPU-bound per-token operations (detokenization, serialization), routing decisions that require multiple remote calls, or missing backpressure that allows unbounded queue growth. In [vLLM-30343](#), the engine’s `detokenize_incrementally` routine runs on every generated token; under high load the event loop stalls and even `/health` hangs. Of the 16 faults in this category, 11 originate from Ray Serve (single-threaded proxy, missing backpressure, hot-path routing), with the rest in vLLM (4) and SGLang (1).

**R3: Memory Mismanagement** GPU or CPU memory is allocated but never reclaimed (leak), grows without bound (unbounded cache), fails to return to baseline between requests (deferred GC), or is provisioned proportionally to configured maxima rather than actual demand (over-allocation). Consider [vLLM-32789](#): the encoder output cache grows without bound because cache-free attempts miss entries whose reference counts never reach zero. Of the 23 Memory Mismanagement faults, 13 originate from vLLM, reflecting the complexity of its KV cache, encoder cache, Triton JIT cache, and LoRA preallocation paths.

**R4: Concurrency Bug** Concurrent execution exposes a race condition, an unsafe interleaving, or a state-consistency violation that is benign at low parallelism but triggers faults at scale. Concrete mechanisms include cancellation races, LRU mutations that race with concurrent lookups, buffer overwrites where a scheduler writes to a slot the GPU is still reading, and request-ID collisions that misroute responses. For instance, in [Ray-59837](#), a



model-multiplexing deployment uses `self.models.pop(model_id)` followed by reinsertion to update LRU order; under concurrent requests, another async task can observe the key as missing and trigger a duplicate model load. The 13 faults split between Ray Serve (7), SGLang (4), and vLLM (2).

**R5: Kernel & Accelerator Defect** A GPU kernel or accelerator backend has a boundary condition, shape assumption, or device-specific behavior that becomes incorrect at scale. Concrete mechanisms include kernel grid dimensions that exceed hardware limits when batch size surpasses a threshold (e.g., 65,536 tokens), attention backends that crash for prompt lengths above their workspace allocation, and accelerator-specific performance cliffs. In [SGL-9142](#), the FlashInfer MLA backend triggers an illegal memory access on a 17,133-token prompt, while the same prompt processes correctly under the FA3 backend. The 13 faults concentrate in inference engines (vLLM 8, SGLang 4); Ray Serve has only 1.

**R6: Distributed System Fault** A multi-node or multi-process coordination mechanism fails when the cluster topology, parallelism degree, or inter-node transfer load exceeds what the implementation was tested against. Concrete mechanisms include communication-library assertions that hold on single-node NVLink but fail on multi-node RDMA, model-loading pipelines whose IPC channels are overwhelmed at large rank counts, KV-cache transfer deadlocks in prefill-decode disaggregation under high concurrency, and library-version incompatibilities. In [SGL-5186](#), DeepSeek-R1 deployed with DeepEP low-latency mode works on a single 8-GPU node but crashes on a 2-node (16-GPU) configuration with an NVSH-MEM assertion. Of the 9 faults in this category, 7 originate from SGLang; the remaining 2 occur in Ray Serve (1) and vLLM (1).

**R7: Configuration & Validation Defect** The system accepts a configuration that it cannot correctly execute at scale, either because a hard-coded limit is too restrictive (e.g., gRPC message size), an internal capacity estimate is incorrect (e.g., `max()` instead

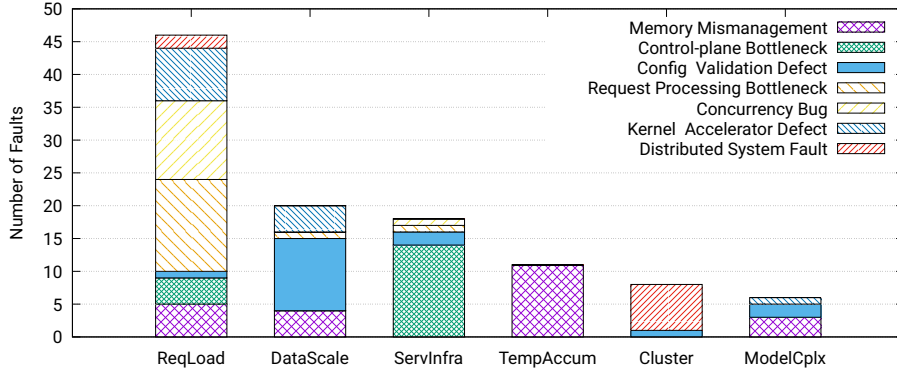


Figure 6.2: Distribution of root cause categories across scaling dimensions.

of `min()` in a memory calculation), an unsupported feature combination is not rejected at startup (e.g., TP mismatch in PD disaggregation for non-MLA models), or a required scaling capability is entirely absent (e.g., multi-application support, model multiplexing). [SGL-15674](#) illustrates this: PD disaggregation with mismatched TP sizes between prefill (TP=2) and decode (TP=1) for a non-MLA model shows a 2x TTFT increase at low QPS and crashes completely at 128 concurrent requests; the system accepts this unsupported configuration silently. The 17 Configuration & Validation Defects are spread evenly across systems (Ray Serve 8, vLLM 5, SGLang 4) and are preventable through validation, correct estimation, or explicit feature support before serving begins.

Across dimensions (Figure 6.2), several mappings are concentrated: Temporal Accumulation maps entirely to Memory Mismanagement (100%), Cluster & Parallelism mostly to Distributed System Fault (87.5%), and Serving Infrastructure mostly to Control-plane Bottleneck (77.8%). Request Load is the exception, spanning all seven causes; Data Scale is dominated by Configuration & Validation Defect (55%).

Across symptoms (Figure 6.3), Resource Exhaustion maps mostly to Memory Mismanagement (79.2%), Control-plane Degradation to Control-plane Bottleneck (94.1%), and Correctness Violation to Concurrency Bug (66.7%). Liveness Failure is led by Request Processing Bottleneck (5), Hard Failure by Kernel & Accelerator Defect (37.5%), and Throughput Degradation by Request Processing Bottleneck (50%).

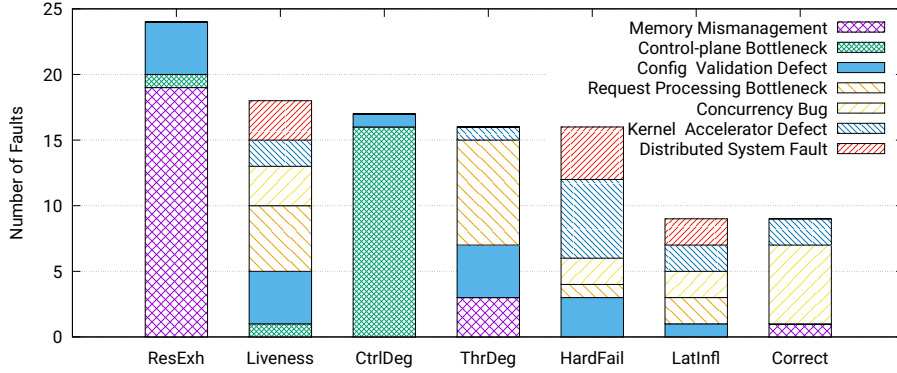


Figure 6.3: Distribution of root cause categories across symptom categories.

### RQ3 Summary:

- Memory Mismanagement is the most frequent root cause (21.1%) and is dominated by vLLM (56.5%); it is the dominant mechanism behind Resource Exhaustion (the most frequent symptom in RQ2).
- Ray Serve faults originate from a tight pair of root causes, Control-plane Bottleneck (16 faults) and Request Processing Bottleneck (11 faults), together 56.2% of all Ray Serve faults; both are downstream of centralized hot paths and insufficient backpressure.
- Configuration & Validation Defects are the highest-leverage category for systematic intervention: they are distributed evenly across systems (15.6%), admit the simplest fixes (one-line formula correction, configurable parameter, startup assertion), and are preventable by fail-fast validation.
- Distributed System Faults concentrate in SGLang (77.8%), driven by prefill-decode disaggregation, and exhibit threshold effects (e.g., DeepSeek-R1 loads on 4 nodes but fails on 16; PD disaggregation works at concurrency 32 but deadlocks at 256).

## 6.6 Fix Strategies

We examine each fault’s code-level fix, if any, using linked pull requests, commit messages, and developer discussions. A fault is *fixed* only when a code change addresses the root cause; operator workarounds (e.g., “reduce the number of LoRA adapters,” “set `CUDA_LAUNCH_BLOCKING=1`”) remain *unfixed*. Open coding yields six fix strategies, ranging from local code-path changes to structural reorganization and preventive validation. Their distribution is shown in Table 6.5.

Table 6.5: Distribution of scalability faults by fix strategy.

| Fix Strategy                   | Ray | SGL | vLLM | Total | %     |
|--------------------------------|-----|-----|------|-------|-------|
| Configuration & Validation Fix | 6   | 2   | 4    | 12    | 11%   |
| Concurrency Hardening          | 7   | 3   | 1    | 11    | 10.1% |
| Algorithmic Optimization       | 6   | 0   | 3    | 9     | 8.3%  |
| Kernel & Backend Fix           | 0   | 3   | 5    | 8     | 7.3%  |
| Resource Lifecycle Fix         | 2   | 0   | 3    | 5     | 4.6%  |
| Architectural Restructuring    | 4   | 1   | 0    | 5     | 4.6%  |
| <b>Fixed (subtotal)</b>        | 25  | 9   | 16   | 50    | 45.9% |
| Unfixed                        | 23  | 19  | 17   | 59    | 54.1% |
| <b>Total</b>                   | 48  | 28  | 33   | 109   | 100%  |

**F1: Configuration & Validation Fix** The fix makes a previously hard-coded limit configurable, corrects an internal capacity formula, adds startup validation that rejects unsupported configurations, or introduces bounds and backpressure to prevent overload. In [Ray-43109](#), a hard-coded gRPC message-size limit silently dropped large payloads; the fix exposes the limit as a configurable parameter. Configuration & Validation Fix is the most common strategy (24% of fixed faults) and spans all three systems.

**F2: Concurrency Hardening** The fix eliminates a race condition, makes a critical section atomic, adds synchronization primitives, or isolates mutable state to prevent concurrent corruption. In [SGL-17850](#), chunked prefill batches could assign the same `req.to_token_pool` slot to two concurrent requests, allowing scheduler writes to overwrite KV cache indices being read by the GPU; the fix keeps chunked requests bound to their slots until those slots are released. Concurrency Hardening accounts for 11 of 50 fixed faults, distributed across all three systems.

**F3: Algorithmic Optimization** The fix reduces the asymptotic or constant-factor cost of a hot-path operation: caching intermediate results, batching fine-grained operations, replacing linear scans with indexed lookups, or removing redundant work from critical loops. In [Ray-45792](#), the Serve controller scanned all deployments to return status for a small subset; the fix replaces the scan with indexed lookup by deployment ID. 6 of 9 Algorithmic Optimization fixes occur in Ray Serve, consistent with its control-plane bottleneck profile.

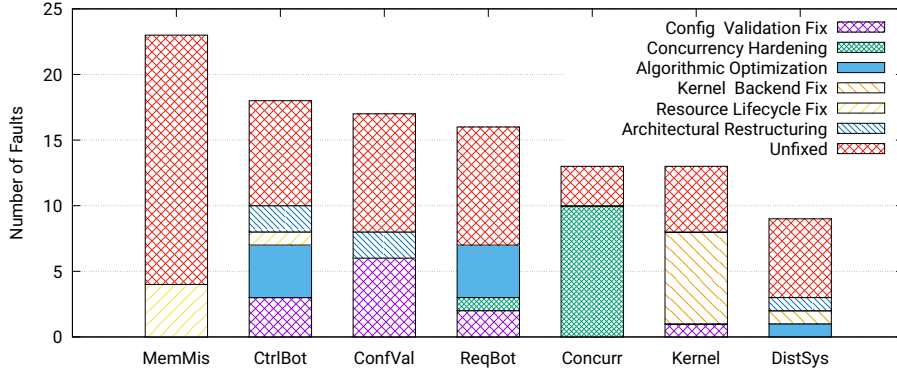


Figure 6.4: Distribution of fix strategies across root cause categories.

**F4: Kernel & Backend Fix** The fix corrects a GPU kernel boundary condition, adjusts shape or grid assumptions, adds a backend fallback for unsupported configurations, or optimizes kernel parallelization. In [SGL-7545](#), the fused MoE kernel computed its grid dimension from batch size and exceeded the hardware limit of 65,536; the fix partitions the launch grid for larger effective batches.

**F5: Resource Lifecycle Fix** The fix repairs a memory leak, adds cache eviction or size bounds, corrects garbage collection behavior, or ensures resources are released when no longer needed. In [vLLM-32789](#), encoder output caches grew without bound because cache-free attempts missed entries whose reference counts never reached zero; the fix releases entries after a full scheduling cycle.

**F6: Architectural Restructuring** The fix reorganizes component responsibilities, introduces a new architectural element (e.g., sharding, parallelism), or adds a previously missing capability. Take [Ray-44784](#) as an example: many `DeploymentHandles` in one process overloaded the Serve controller, so the fix splits controller responsibilities across multiple actors.

Root causes map to fixes in two regimes (Figure 6.4). Concurrency Bug maps to Concurrency Hardening in 76.9% cases, and Kernel & Accelerator Defect maps to Kernel & Backend Fix in 53.8%. Control-plane and Request Processing Bottlenecks split across optimization, validation, restructuring, and lifecycle fixes depending on whether the immediate cause is

an inefficient hot path, hard-coded limit, or design bottleneck. Unfixed rates are highest for Memory Mismanagement (82.6%) and Distributed System Fault (66.7%), while Configuration & Validation Defect combines a comparatively high fix rate (47.1%) with simple, local patches.

#### **RQ4 Summary:**

- Most scalability faults remain unfixed (54.1%), with comparable rates across systems (Ray Serve 47.9%, SGLang 67.9%, vLLM 51.5%), indicating a fault-class difficulty rather than a system-specific maintainer effect.
- Fix rates invert frequency: Memory Mismanagement is the most frequent root cause yet has the highest unfixed rate (82.6%), while Configuration & Validation Defects have the simplest patches and a comparatively high fix rate (47.1%).
- Root causes map systematically to fixes: Concurrency Bug → Concurrency Hardening (76.9%), Kernel & Accelerator Defect → Kernel & Backend Fix (53.8%), and Memory Mismanagement → Resource Lifecycle Fix or unfixed (82.6% unfixed).
- Architectural Restructuring is rare (10%) but addresses the deepest scalability ceilings (proxy sharding, multi-application support, model multiplexing); developers mostly apply incremental fixes, leaving structural ceilings to accumulate.

## **6.7 Discussion**

The four research questions, taken together, paint a coherent picture of code-level scalability faults in LLM serving infrastructures. In this section, we draw out the cross-cutting implications, identify the structural reasons why these faults escape standard testing, and provide recommendations for serving-system designers, tool builders, and operators.

### **6.7.1 Implications for Serving System Designers**

The two-plane pattern implies that serving system designers should treat control-plane and data-plane scalability as separate design targets. The control plane scales with management-graph cardinality (e.g., deployments, replicas, handles, metrics streams), while the data plane scales with token volume, memory lifecycle, GPU-kernel boundaries, and parallelism degree. Three concrete patterns emerge from the root-cause and fix analyses.

**Validate configurations at startup** Configuration & Validation Defects (17 faults, 15.6%) are spread evenly across all three systems and have local fixes. The fixes are structurally simple: a one-line formula correction (e.g., `max()` replaced with `min()` when computing available KV slots), a configurable parameter exposed in place of a hard-coded constant (e.g., gRPC message size in Ray Serve), or a startup assertion that rejects unsupported feature combinations (e.g., prefill-decode disaggregation with mismatched tensor-parallel sizes for non-MLA models). Adopting a fail-fast validation discipline at engine startup, in which every supported configuration is checked against system invariants before serving begins, would prevent a substantial fraction of production faults at near-zero engineering cost.

**Treat memory lifecycle as a first-class concern** Memory Mismanagement is the most frequent root cause overall (23 faults), and it has the highest unfixed rate (82.6%). The pathologies cluster into three patterns: unbounded growth (e.g., encoder caches that grow indefinitely, Triton JIT caches that accumulate across model reloads), deferred reclamation (e.g., memory that fails to return to baseline between requests), and configuration-driven over-allocation (e.g., LoRA capacity scaling with the configured maximum rather than active demand). Each pattern admits a different mitigation: explicit eviction tied to request completion, lifecycle hooks tied to model events, and lazy allocation rather than max-driven pre-allocation. A unified *memory lifecycle audit* during inference engine development, in which every persistent allocation is traced to a clear release path, would close many of the unfixed cases.

**Shard centralized control-plane components** Of the 18 Control-plane Bottleneck faults, 16 originate from Ray Serve. The recurring pattern is a single global actor (the controller, the proxy, or a metrics aggregator) whose work scales linearly with the size of the management graph and saturates CPU at hundreds to thousands of managed entities. The fixes that have landed in Ray Serve follow a consistent direction: sharding the proxy into multiple actors, replacing  $O(N)$  scans with indexed lookups, and amortizing per-handle met-

rics push. For new serving frameworks, designing the control plane with a sharded baseline rather than retrofitting it after CPU saturation would prevent this fault class entirely.

### 6.7.2 Implications for Tool Builders

**Scale-aware testing and triage** The cross-cutting analyses in RQ2–RQ4 reveal several near-deterministic mappings between scaling dimensions, symptoms, root causes, and fixes: Temporal Accumulation maps to Memory Mismanagement (100%), Resource Exhaustion symptoms map to Memory Mismanagement root causes (79.2%), Concurrency Bug root causes map to Concurrency Hardening fixes (76.9%), Control-plane Degradation symptoms map to Control-plane Bottleneck root causes (94.1%). These correlations are useful only if tests cross the scale thresholds at which faults manifest. Tool builders should therefore combine cardinality sweeps for the control plane, soak tests for memory lifecycle, multi-node smoke tests for distributed coordination, and boundary-value tests for kernel grid limits and message-size constraints. Once a symptom appears, the same correlations can drive symptom-driven triage tools that recommend a small set of likely root causes during incident response.

**Output validation oracles for correctness violations** Correctness Violations (9 faults, 8.3%) are the hardest category to detect because the system appears healthy by all standard metrics: latency is within bounds, throughput is acceptable, no error is raised. The dominant root cause is concurrency-induced state corruption: KV cache buffers overwritten during chunked prefill, LRU caches mutated during pop-and-reinsert, request-ID collisions misrouting responses. Detecting these faults requires *semantic* oracles: golden-output regression at fixed seeds, determinism checks across concurrency levels, and content-validity assertions on streaming responses. Lightweight tools that automate these checks during load testing would close a gap that is currently filled only by post-deployment user complaints.



**Anti-pattern checklists** The Configuration & Validation Defect category is amenable to automated detection. Common patterns include hard-coded protocol limits, capacity formulas that confuse `max` and `min`, missing startup assertions for unsupported feature combinations, and absent overload-control primitives. A static checklist tool that scans serving-system source for these patterns could surface latent scalability faults before deployment.

### 6.7.3 Implications for Operators

For operators of LLM serving deployments, our taxonomy offers a diagnostic vocabulary. A symptom observed in production maps with high probability to a small set of root causes. A scaling regime imposed by a workload maps to a small set of high-risk fault categories. We summarize these mappings in three operational rules: (1) sustained memory growth on long-lived deployments is almost always a Memory Mismanagement fault, regardless of which scaling dimension induced it; (2) hard crashes on multi-node deployments that succeed at single node are almost always a Distributed System Fault, often involving inter-node communication libraries; (3) throughput plateaus that persist after replica scale-out are almost always a Request Processing Bottleneck in the routing or proxy layer, and adding more replicas will not help. Equipped with these rules, an operator can route incidents to the right team or maintainer with substantially less guesswork.

## 6.8 Threats to Validity

We organize threats by whether they affect our internal conclusions or their external generalization [302].

**Internal threats** Two threats are internal to the study. First, the operational definition of *scalability fault* requires excluding bugs whose manifestations are not scale-dependent (e.g., type errors, environment-specific misconfiguration), and borderline cases (e.g., leaks that

turn catastrophic only after hours of traffic) involved judgment calls; we document these calls in the public artifact. Second, labeling along four axes is inherently subjective. We mitigated this risk through independent double-coding by two authors with discussion-based adjudication (and a third author for unresolved cases), and full release of the labeled data for third-party replication.

**External threats** Three threats limit generalization. First, GitHub-only mining biases the corpus toward *reported*, *reproducible*, and *actively maintained* subsystems; faults silently worked around in production are invisible. We partially mitigate the survivorship bias common to bug studies by including both fixed and unfixed faults (54.1% unfixed). Second, a corpus of three systems and 109 faults is on the smaller end for bug studies, but the systems were chosen to span the architectural spectrum (a serving framework, a paged-cache inference engine, and a structured-generation engine with multi-node disaggregation), and the two-plane finding rests on architectural roles rather than implementation specifics; the absolute size is comparable to peer empirical bug studies in distributed systems [146, 210, 232]. Third, our cutoff is January 31, 2026, and these systems evolve quickly; specific counts should be read as a snapshot, while the taxonomy is grounded in architectural roles (control plane vs. data plane, single-node vs. multi-node) likely to persist as individual codebases change.

## 6.9 Related Work

**Empirical bug studies of DL and LLM systems** A growing body of empirical work characterizes bugs in deep-learning software. Early studies analyzed program bugs in TensorFlow [432] and across DL libraries [180], finding that data and logic bugs dominate. Larger-scale taxonomies followed, including a fault catalog built from mining and practitioner interviews across eight DL frameworks [179] and a 1,000-bug study spanning four major DL frameworks [107]. Performance-specific issues in DL systems have also been characterized and supported with detection tools [100]. Closest to our domain is a recent study of bugs

in LLM inference engines, which analyzed 929 bugs across Llama.cpp, vLLM, DeepSpeed, MLC-LLM, and TensorRT-LLM with a six-symptom and 28-root-cause taxonomy [227]. Our study differs in three ways. First, we narrow the scope to scalability-specific faults, the subset of bugs whose manifestation depends on scale, whereas that work studies the full bug population. Second, we cover a serving framework (Ray Serve) and SGLang, neither of which appear in their corpus, capturing distinct architectural surfaces such as actor-based replica management, RadixAttention prefix sharing, and prefill–decode disaggregation. Third, we explicitly label scaling dimensions (concurrency, context length, replicas, parallelism, and so on) as a first-class axis, enabling a cross-cutting analysis (Sections 6.4 to 6.6) that is absent from prior work on LLM bugs.

**Empirical bug studies of distributed systems** Classical work on bugs in distributed systems has shaped our methodology. Concurrency bugs were first characterized at scale through a study of 105 real-world cases across MySQL, Apache, Mozilla, and OpenOffice [232], establishing the symptom plus root-cause template that subsequent studies extended. Configuration errors were found to be a major source of system unavailability [407], and performance bugs in widely used C/C++ projects were attributed predominantly to developer misunderstandings of API behavior [188]. At cloud scale, CbsDB cataloged 3,655 vital issues across six cloud systems [159], and a contemporaneous study famously argued that simple testing on small clusters could prevent most critical failures in distributed data-intensive systems [412]. Subsequent taxonomies focused on specific fault classes, including non-deterministic concurrency bugs in datacenter systems [210] and crash-recovery bugs across four large-scale systems [146].

Most directly related to our framing, a line of work formalized scalability faults as a distinct fault class. Prior work argues that 100-node testing is insufficient to surface scalability bugs [211], and the ScaleCheck approach later operationalized this insight by combining program analysis with single-machine colocation techniques to expose scalability bugs at

thousand-node scales [333]. Building on this lineage, a 444-fault analysis across ten JVM-era systems proposed dimensional code fragments as a unifying abstraction [458]. We apply the scalability-fault framing to LLM serving infrastructures and contrast the resulting taxonomy with the classical distributed-systems taxonomy: GPU kernel boundaries, KV-cache lifecycle, multi-modal inputs, and prefill–decode disaggregation have no direct analogue in classical JVM-era systems, and the control-plane code paths in Ray Serve differ structurally from the data-shuffle and replication paths analyzed in prior studies.

**LLM serving systems** The systems we study are part of a rapidly developing ecosystem. Iteration-level scheduling for transformer-based generative models was introduced by Orca [408], and vLLM pioneered PagedAttention to manage KV cache memory [202]. SGLang introduced RadixAttention for prefix sharing alongside a structured-generation frontend [442], and Ray Serve provides general distributed serving abstractions on top of the Ray runtime [254]. More recent systems explore architectural specialization: DistServe [448] and Splitwise [281] disaggregate prefill and decode onto separate GPU pools; Sarathi-Serve mitigates the throughput–latency tradeoff with chunked prefills [63]; AlpaServe explores statistical multiplexing across model variants [217]; and Mooncake describes a KV-cache-centric production serving platform [292]. These papers describe the design space and motivate the scaling dimensions our study analyzes; our contribution is empirical characterization of how their implementations fail under scale.

**ML systems reliability and characterization** Beyond bug studies, several recent papers characterize the operational behavior of large-scale ML systems. Microsoft’s Philly platform supplied 4,960 DL job failures for an empirical study of training-job reliability [430], and a six-month characterization of LLM development workloads on a 4,704-GPU production cluster captured the rhythm of pre-training failures and recoveries [177]. DeepPerf, an instance of detection tooling derived from a performance-bug study, targets DL training code [100]. Most adjacent on the infrastructure side, an analysis of 706 low-GPU-utilization

issues across 400 DL jobs at Microsoft is the closest training-domain analog of our work [147]. These workload-characterization studies complement our fault-characterization study: they describe how systems behave in production, while we describe how they break. Our focus on the inference-serving stack and on scale-dependent fault manifestation distinguishes our work from these reliability and workload studies.

## Chapter Summary

LLM-serving infrastructure inherits classical scalability fault patterns and introduces new ones. The faults we classified bifurcate into a control-plane regime in the serving framework and a data-plane regime in the inference engines, with disjoint scaling dimensions, symptoms, root causes, and fixes. The classification surfaces a scaling dimension absent from prior taxonomies, the number and size of concurrently loaded models, which drives symptoms such as GPU fragmentation and throughput collapse under multi-model workloads. A majority of these faults remain unfixed; a small set of defect categories admit the simplest fixes and emerge as the highest-leverage class for systematic detection and repair. The natural next step is to extend SCALELENS from [Chapter 5](#) to detect the patterns characterized here, and explore the design space of systematic repairs for the high-leverage classes.

This chapter is based on the paper “On the Code-Level Scalability Issues of LLM Serving Infrastructures,” co-authored with Goodness Ayinmode and Cindy Rubio-González. The work was supported by NSF grant CCF-2119348.

# Chapter 7

## Conclusions and Future Work

### 7.1 Summary of Contributions

This dissertation has presented studies and tools that address three structural concerns of modern software engineering: the reproducibility of software engineering artifacts, the reliability in the LLM era of software engineering, and the scalability of modern software infrastructure. Together, these concerns determine whether the artifacts, tools, components, and infrastructures on which software development depends can remain trustworthy as systems evolve, fail, and scale.

[Chapter 2](#) presented a comprehensive survey of 151 software defect datasets, characterizing their scope, construction, availability, usability, and actual uses by the research community. The survey identified gaps in specialized application domains, defect types, and fast-growing programming languages as opportunities for future work.

[Chapter 3](#) addressed two failure modes of software defect datasets: silent decay and uneven growth. A systematic study of reproducibility across five widely used Java defect datasets found that reproducibility criteria differ across datasets and that all five are prone to breakages. A 13-month longitudinal case study of BUGSWARM identified the dominant root causes of breakages, and dependency caching and artifact isolation preserved reproducibility

at 95%, on par with the most reliable handcrafted dataset. The chapter also presented `ACTIONSREMAKER`, the first tool designed to reproduce GitHub Actions workflow runs, which has since been incorporated as a key component of the `BUGSWARM` pipeline, mining thousands of reproducible artifacts to date.

[Chapter 4](#) addressed two ways in which LLMs now intersect with software engineering: as opaque components embedded inside software systems, and as repair tools applied to software failures. The chapter introduced semantic markers, which pair input instrumentation with delta debugging to isolate offending prompt fragments without access to model internals. This methodology transforms prompt debugging from an ad hoc process into a systematic procedure, and has been implemented as development tooling at Amazon Web Services. The chapter also presented `CI-BENCH`, a unified framework for evaluating LLM-based program repair on real CI/CD failures; the strongest tool-model pairing repaired only 35% of these failures, quantifying how far LLM-based repair remains from broad real-world CI/CD coverage.

[Chapter 5](#) presented the first systematic empirical study of scalability faults in distributed systems, manually analyzing 444 issues across 10 systems and showing that most scalability faults follow a dimensional code fragment plus anti-pattern signature. `SCALELENS` operationalized this signature by combining lightweight dynamic analysis with static pattern matching, improving DCF detection by  $4.2\times$  over a memory-monitoring baseline and surfacing 334 latent issues in the latest Cassandra, HDFS, and Ignite codebases.

[Chapter 6](#) extended the characterization of scalability faults to modern AI infrastructure through an empirical study of 109 scalability faults across Ray Serve, vLLM, and SGLang. The study revealed that LLM-serving infrastructure inherits classical scalability fault patterns while introducing new scaling dimensions specific to the GPU serving stack, with distinct control-plane and data-plane regimes. A small set of defect categories emerged as the highest-leverage class for systematic detection and repair.

## 7.2 Broader Impact

The studies and tools presented in this dissertation address fundamental challenges in modern software engineering that affect researchers and practitioners across many domains:

**Empirical foundations for software engineering research.** As the field increasingly depends on shared datasets and benchmarks, the survey and reproducibility studies in this dissertation establish empirical foundations that strengthen the replicability and continuity of research across the software engineering community.

**Production deployment of research tools.** The tools presented here have moved from research artifacts into production use: ACTIONSREMAKER is incorporated into the BUGSWARM pipeline, semantic markers run as development tooling inside Amazon Web Services, and SCALELENS surfaces latent issues in widely deployed distributed systems. This bridges the often-noted gap between research output and practical adoption.

**Cross-domain applicability of fault models.** Patterns introduced in one part of the dissertation transfer across pillars. Dimensional code fragments and four-axis taxonomies originally developed for classical distributed systems extend to LLM-serving infrastructure, demonstrating that scalability fault models generalize beyond a single class of systems.

**Rigorous evaluation in the LLM era.** As LLM-based tools rapidly enter software engineering practice, frameworks like CI-BENCH enable rigorous evaluation of LLM-based tools on real-world software failures.

## 7.3 Future Directions

Several promising directions for future research emerge from this work:

**Underrepresented defect categories.** The most pressing follow-on for Part I is to fill the underrepresented defect categories identified by the survey. ACTIONSREMAKER supplies the infrastructure for collecting reproducible fail-pass pairs at scale; matching that supply with research demand requires targeted mining policies for the underrepresented categories



identified by the survey, including specialized application domains, defect types, and fast-growing programming languages.

**Multi-turn LLM debugging.** Semantic markers can be extended beyond single prompts to multi-turn agent traces, where the failure-inducing input may be a sequence of messages, retrieved documents, tool calls, and intermediate observations rather than a single prompt. This extension becomes essential as agentic LLM systems are increasingly deployed in production software.

**LLM-based program repair on real-world failures.** CI-BENCH exposes a clear gap between current LLM-based repair capabilities and the demands of real CI/CD failures. Future repair systems should be evaluated not only on whether they can produce plausible patches, but also on whether they can plan and execute multi-step, domain-aware changes under realistic build constraints.

**Proactive detection in LLM-serving infrastructure.** Extending SCALELENS to LLM-serving infrastructure would require new dynamic instrumentation for GPU memory, batching behavior, model loading, and request scheduling, as well as an expanded static anti-pattern catalog for ML-specific code fragments. A benchmark of confirmed scalability faults in systems such as Ray Serve, vLLM, and SGLang would enable detection techniques to be evaluated against production-relevant failures rather than synthetic stress cases.

## 7.4 Concluding Remarks

Modern software engineering increasingly depends on artifacts, components, and infrastructures that are larger, more complex, and harder to inspect than those of the past. This dissertation has contributed studies and tools that help the field address three of these structural concerns: keeping research artifacts reproducible, debugging and evaluating LLM-related software, and detecting faults that emerge only at scale. As software systems continue to evolve, the need for empirical foundations and practical tools that preserve their trustwor-

thiness will only grow. The contributions presented here aim to provide a foundation for addressing these challenges in current and future software systems.

# Bibliography

- [1] Artifact review and badging. <https://www.acm.org/publications/policies/artifact-review-and-badging-current>.
- [2] CASSANDRA-15013. <http://issues.apache.org/jira/browse/CASSANDRA-15013>.
- [3] CASSANDRA-15400. <http://issues.apache.org/jira/browse/CASSANDRA-15400>.
- [4] CASSANDRA-5456. <http://issues.apache.org/jira/browse/CASSANDRA-5456>.
- [5] CASSANDRA-5506. <http://issues.apache.org/jira/browse/CASSANDRA-5506>.
- [6] HADOOP-15696. <http://issues.apache.org/jira/browse/HADOOP-15696>.
- [7] HADOOP-16385. <http://issues.apache.org/jira/browse/HADOOP-16385>.
- [8] HDFS-13039. <http://issues.apache.org/jira/browse/HDFS-13039>.
- [9] HDFS-13768. <http://issues.apache.org/jira/browse/HDFS-13768>.
- [10] IGNITE-8681. <http://issues.apache.org/jira/browse/IGNITE-8681>.
- [11] KAFKA-5642. <http://issues.apache.org/jira/browse/KAFKA-5642>.
- [12] STORM-3256. <http://issues.apache.org/jira/browse/STORM-3256>.
- [13] YARN-6188. <http://issues.apache.org/jira/browse/YARN-6188>.
- [14] YARN-7147. <http://issues.apache.org/jira/browse/YARN-7147>.

- [15] AIDER. <https://aider.chat/2024/06/02/main-swe-bench.html>.
- [16] Summary of the Amazon Kinesis Event in the Northern Virginia (US-EAST-1) Region. <https://aws.amazon.com/message/11201/>.
- [17] Summary of the AWS Service Event in the Northern Virginia (US-EAST-1) Region. <https://aws.amazon.com/message/12721/>.
- [18] Summary of the Amazon DynamoDB Service Disruption and Related Impacts in the US-East Region. <https://aws.amazon.com/message/5467D2/>.
- [19] antlr. <https://www.antlr.org/>.
- [20] Apache Cassandra. <https://cassandra.apache.org/>.
- [21] Apache Hadoop. <https://hadoop.apache.org/>.
- [22] Apache HBase. <https://hbase.apache.org/>.
- [23] Apache HDFS. [https://hadoop.apache.org/docs/r1.2.1/hdfs\\_design.html](https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html).
- [24] Apache Ignite. <https://ignite.apache.org/>.
- [25] Apache Kafka. <https://kafka.apache.org/>.
- [26] Apache Hadoop MapReduce. <https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html>.
- [27] Apache Spark. <https://spark.apache.org/>.
- [28] Apache Storm. <https://storm.apache.org/>.
- [29] Apache Hadoop YARN. <https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html>.

- [30] Apache Projects Directory. <https://projects.apache.org/projects.html?category>.
- [31] Cassandra at Apple: 1000s of Clusters, 300k Nodes, 100 PB. <https://news.ycombinator.com/item?id=33124631>.
- [32] bears-bugs/bears-benchmark: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. <https://github.com/bears-bugs/bears-benchmark>.
- [33] liuhuigmail/GrowingBugRepository: A bug repository that keeps growing. <https://github.com/liuhuigmail/GrowingBugRepository>.
- [34] jiangyanjie/BugBuilder: Extracting Concise Bug-Fixing Patches from Human-Written Patches in Version Control Systems. <https://github.com/jiangyanjie/BugBuilder>.
- [35] bugs-dot-jar/bugs-dot-jar: Bugs.jar: A Large-scale, Diverse Dataset of Bugs for Java Program Repair. <https://github.com/bugs-dot-jar/bugs-dot-jar>.
- [36] BugSwarm/bugswarm. <https://github.com/BugSwarm/bugswarm>.
- [37] Performance Antipattern: Chatty I/O. <https://docs.microsoft.com/en-us/azure/architecture/antipatterns/chatty-io/>.
- [38] DB performance issue. <https://circleci.statuspage.io/incidents/hr0mm9xmm3x6>.
- [39] CodeQL. <https://codeql.github.com/>.
- [40] rjust/defects4j: A Database of Real Faults and an Experimental Infrastructure to Enable Controlled Experiments in Software Engineering Research. <https://github.com/rjust/defects4j>.

- [41] Performance Antipattern: Extraneous Fetching antipattern. <https://docs.microsoft.com/en-us/azure/architecture/antipatterns/extraneous-fetching/>.
- [42] GitHub Search for Number of Repositories. <https://github.com/search?type=repositories>.
- [43] GitHub Actions. <https://docs.github.com/en/actions>.
- [44] GitHub - leap-laboratories/PIZZA: An attribution library for LLMs. <https://github.com/leap-laboratories/PIZZA>.
- [45] Github 2.0: Github language statistics. <https://madnight.github.io/github>.
- [46] Cloud Filestore ListInstances API failed with error code 429 globally. <https://status.cloud.google.com/incidents/X8SNkK2BPyCrc1sveeu>.
- [47] BigQuery is experiencing issues with streaming API in US region. <https://status.cloud.google.com/incidents/mREMLwZFe3FuLLn3zfTw>.
- [48] Javassist Github Repository. <https://github.com/jboss-javassist/javassist>.
- [49] Java Collections. <https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html>.
- [50] Java Instrumentation Agents. <https://docs.oracle.com/javase/8/docs/technotes/guides/instrumentation/index.html>.
- [51] Jenkins. <https://www.jenkins.io/>.
- [52] How LinkedIn customizes Apache Kafka for 7 trillion messages per day. <https://www.linkedin.com/blog/engineering/open-source/apache-kafka-trillion-messages>.

- [53] Maven – Introduction to Repositories. <https://maven.apache.org/guides/introduction/introduction-to-repositories.html>.
- [54] nektos/act. <https://github.com/nektos/act>.
- [55] Artifact Review and Badging - Current. <https://www.acm.org/publications/policies/artifact-review-and-badging-current>.
- [56] SWE-Bench Lite. <https://www.swebench.com/lite.html>.
- [57] Travis Build. <https://github.com/travis-ci/travis-build>.
- [58] Travis CI. <https://www.travis-ci.com/>.
- [59] Travis Caching. <https://docs.travis-ci.com/user/caching/>.
- [60] Kerberizing Hadoop Clusters at Twitter. [https://blog.x.com/engineering/en\\_us/topics/infrastructure/2023/kerberizing-hadoop-clusters-at-twitter](https://blog.x.com/engineering/en_us/topics/infrastructure/2023/kerberizing-hadoop-clusters-at-twitter).
- [61] The YARN Timeline Service v.2. <https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/TimelineServiceV2.html>.
- [62] S. Afrose, Y. Xiao, S. Rahaman, B. P. Miller, and D. Yao. Evaluation of static vulnerability detection tools with java cryptographic API benchmarks. *IEEE TSE*, 49(2): 485–497, 2023.
- [63] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, A. Tumanov, and R. Ramjee. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In *OSDI*. USENIX Association, 2024.
- [64] T. Ahmed, P. T. Devanbu, C. Treude, and M. Pradel. Can llms replace manual annotation of software engineering artifacts? In *MSR*, pages 526–538. ACM, 2025.
- [65] M. Ahn, J. Han, Y. Kwon, and J. Jeong. Identifying on-/off-cpu bottlenecks together with blocked samples. In *OSDI*, pages 893–910. USENIX Association, 2024.

- [66] E. N. Akimova, A. Y. Bersenev, A. A. Deikov, K. S. Kobylkin, A. V. Konygin, I. P. Mezentsev, and V. E. Misilov. Pytracebugs: A large python code dataset for supervised machine learning in software defect prediction. In *APSEC*, pages 141–151. IEEE, 2021.
- [67] M. M. U. Alam, T. Liu, G. Zeng, and A. Muzahid. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *EuroSys*, pages 298–313. ACM, 2017.
- [68] R. Aleithan, H. Xue, M. M. Mohajer, E. Nnorom, G. Uddin, and S. Wang. Swe-bench+: Enhanced coding benchmark for llms. *CoRR*, abs/2410.06992, 2024.
- [69] H. K. Almula and G. Gay. Learning how to search: generating effective test cases through adaptive fitness function selection. *ESE*, 27(2):38, 2022.
- [70] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini. Mubench: a benchmark for api-misuse detectors. In *MSR*, pages 464–467. ACM, 2016.
- [71] G. An and S. Yoo. FDG: a precise measurement of fault diagnosability gain of test cases. In *ISSTA*, pages 14–26. ACM, 2022.
- [72] G. An, M. Kwon, K. Choi, J. Yi, and S. Yoo. BUGSC++: A highly usable real world defect benchmark for C/C++. In *ASE*, pages 2034–2037. IEEE, 2023.
- [73] R. Andrade, N. Laranjeiro, and M. Vieira. Bughub: A large scale issue report dataset. In *EDCC*, pages 9–16. IEEE, 2024.
- [74] E. Andreassen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C. Staicu. A survey of dynamic analysis and test generation for javascript. *ACM CSUR*, 50(5):66:1–66:36, 2017.
- [75] G. Antal, N. Vándor, I. Kolláth, B. Mosolygó, P. Hegedüs, and R. Ferenc. Pybughive: A comprehensive database of manually validated, reproducible python bugs. *IEEE Access*, 12:123739–123756, 2024.



- [76] L. Applis and A. Panichella. Hasbugs - handpicked haskell bugs. In *MSR*, pages 223–227. IEEE, 2023.
- [77] B. Aronov, S. Har-Peled, C. Knauer, Y. Wang, and C. Wenk. Fréchet distance for curves, revisited. In *ESA*, volume 4168 of *Lecture Notes in Computer Science*, pages 52–63. Springer, 2006.
- [78] A. Avritzer, A. Janes, C. Trubiani, H. Rodrigues, Y. Cai, D. S. Menasché, and Á. J. A. de Oliveira. Architecture and performance anti-patterns correlation in microservice architectures. In *ICSA*, pages 60–71. IEEE, 2025.
- [79] S. K. Avula, V. Vobbilisetti, and S. Mondal. Minecraft: Automated mining of software bug fixes with precise code context. In *ASE*, pages 1969–1979. IEEE, 2023.
- [80] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. W. Pugh. Using static analysis to find bugs. *IEEE Softw.*, 25(5):22–29, 2008.
- [81] M. A. K. Azad, N. Iqbal, F. Hassan, and P. Roy. An empirical study of high performance computing (HPC) performance bugs. In *MSR*, pages 194–206. IEEE, 2023.
- [82] M. Bagherzadeh, N. Fireman, A. Shawesh, and R. Khatchadourian. Actor concurrency bugs: a comprehensive study on symptoms, root causes, API usages, and differences. *Proc. ACM Program. Lang.*, 4(OOPSLA):214:1–214:32, 2020.
- [83] M. Baker. Reproducibility crisis. *nature*, 533(26):353–66, 2016.
- [84] T. Ball. The concept of dynamic analysis. In *ESEC/SIGSOFT FSE*, pages 216–234. ACM, 1999.
- [85] M. Beller, G. Gousios, and A. Zaidman. Oops, my tests broke the build: an explorative analysis of travis CI with github. In *MSR*, pages 356–367. IEEE Computer Society, 2017.

- [86] M. Beller, G. Gousios, and A. Zaidman. Travistorrent: synthesizing travis CI and github for full-stack research on continuous integration. In *MSR*, pages 447–450. IEEE Computer Society, 2017.
- [87] S. Benton, A. Ghanbari, and L. Zhang. Defexts: a curated dataset of reproducible real-world bugs for modern JVM languages. In *ICSE (Companion Volume)*, pages 47–50. IEEE / ACM, 2019.
- [88] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst. Debugging distributed systems. *Commun. ACM*, 59(8):32–37, 2016.
- [89] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *FSE*, pages 308–318. ACM, 2008.
- [90] D. Beyer, L. Grunske, M. Kettl, M. L. Rosenfeld, and M. Raselimo. P3: A dataset of partial program patches. In *MSR*, pages 123–127. ACM, 2024.
- [91] M. H. M. Bhuiyan, A. S. Parthasarathy, N. Vasilakis, M. Pradel, and C. Staicu. Secbench.js: An executable security benchmark suite for server-side javascript. In *ICSE*, pages 1059–1070. IEEE, 2023.
- [92] D. W. Binkley and K. B. Gallagher. Program slicing. *Adv. Comput.*, 43:1–50, 1996.
- [93] S. Biswas, R. Rathi, A. Dutta, P. Mitra, and R. Mall. A regression test case prioritization technique targeting ‘hard to detect’ faults. *Int. J. Syst. Assur. Eng. Manag.*, 13(3):1066–1081, 2022.
- [94] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, 2003.
- [95] C. Brust, T. Sonnekalb, and B. Gruner. ROMEO: A binary vulnerability detection dataset for exploring juliet through the lens of assembly language. *Comput. Secur.*, 128:103165, 2023.

- [96] Y. Bu, V. R. Borkar, G. Xu, and M. J. Carey. A bloat-aware design for big data applications. In *ISMM*, pages 119–130. ACM, 2013.
- [97] Q. Bui, R. Scandariato, and N. E. D. Ferreyra. Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *MSR*, pages 464–468. ACM, 2022.
- [98] M. Bures, P. Herout, and B. S. Ahmed. Open-source defect injection benchmark testbed for the evaluation of testing. In *ICST*, pages 442–447. IEEE, 2020.
- [99] J. Cai, B. Li, T. Zhang, J. Zhang, and X. Sun. Fine-grained smart contract vulnerability detection by heterogeneous code feature learning and automated dataset construction. *JSS*, 209:111919, 2024.
- [100] J. Cao, B. Chen, C. Sun, L. Hu, S. Wu, and X. Peng. Understanding performance problems in deep learning systems. In *ESEC/SIGSOFT FSE*, pages 357–369. ACM, 2022.
- [101] D. G. Cavezza, R. Pietrantuono, J. Alonso, S. Russo, and K. S. Trivedi. Reproducibility of environment-dependent software failures: An experience report. In *ISSRE*, pages 267–276. IEEE Computer Society, 2014.
- [102] M. L. Chaim, K. Baral, J. Offutt, M. C. Neto, and R. P. A. de Araujo. On subsumption relationships in data flow testing. *Softw. Test. Verif. Reliab.*, 33(6), 2023.
- [103] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray. CODIT: code editing with tree-based neural models. *IEEE TSE*, 48(4):1385–1399, 2022.
- [104] B. Chen, Z. M. Jiang, P. Matos, and M. Lacaria. An industrial experience report on performance-aware refactoring on a database-centric web application. In *ASE*, pages 653–664. IEEE, 2019.

- [105] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang. A survey of compiler testing. *ACM CSUR*, 53(1):4:1–4:36, 2021.
- [106] J. Chen, X. Xia, D. Lo, J. C. Grundy, X. Luo, and T. Chen. Defining smart contract defects on ethereum. *IEEE TSE*, 48(2):327–345, 2022.
- [107] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li. Toward understanding deep learning framework bugs. *ACM Trans. Softw. Eng. Methodol.*, 32(6):135:1–135:31, 2023.
- [108] T. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. N. Nasser, and P. Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *ICSE*, pages 1001–1012. ACM, 2014.
- [109] X. Chen, M. Lin, N. Schärli, and D. Zhou. Teaching large language models to self-debug. In *ICLR*. OpenReview.net, 2024.
- [110] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. A. Wagner. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *RAID*, pages 654–668. ACM, 2023.
- [111] Z. Chen, H. Yao, Y. Lou, Y. Cao, Y. Liu, H. Wang, and X. Liu. An empirical study on deployment faults of deep learning based mobile applications. In *ICSE*, pages 674–685. IEEE, 2021.
- [112] Z. Chen, J. Chen, W. Wang, J. Zhou, M. Wang, X. Chen, S. Zhou, and J. Wang. Exploring better black-box test case prioritization via log analysis. *ACM TOSEM*, 32(3):72:1–72:32, 2023.
- [113] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *SOSP*, pages 73–88. ACM, 2001.
- [114] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz. Begbunch: benchmarking for c bug detection tools. In *DEFECTS*,

- pages 16–20, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605586540.
- [115] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall. An empirical analysis of the docker container ecosystem on github. In *MSR*, pages 323–333. IEEE Computer Society, 2017.
  - [116] L. Clapp, O. Bastani, S. Anand, and A. Aiken. Minimizing GUI event traces. In *SIGSOFT FSE*, pages 422–434. ACM, 2016.
  - [117] J. Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.
  - [118] V. Csuvik and L. Vidács. Fixjs: A dataset of bug-fixing javascript commits. In *MSR*, pages 712–716. ACM, 2022.
  - [119] Z. Cui, W. Dou, Y. Gao, D. Wang, J. Song, Y. Zheng, T. Wang, R. Yang, K. Xu, Y. Hu, J. Wei, and T. Huang. Understanding transaction bugs in database systems. In *ICSE*, pages 163:1–163:13. ACM, 2024.
  - [120] F. R. P. da Ponte, E. B. Rodrigues, and C. L. C. Mattos. Cvejoin: An information security vulnerability and threat intelligence dataset. In *AINA (1)*, volume 661 of *Lecture Notes in Networks and Systems*, pages 380–392. Springer, 2023.
  - [121] L. M. P. da Silva, J. Samhi, and F. Khomh. Llms and stack overflow discussions: Reliability, impact, and challenges. *JSS*, 230:112541, 2025.
  - [122] M. D’Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *ESE*, 17(4-5):531–577, 2012.
  - [123] T. Das, A. Ali, and T. Mikkonen. Taxonomy of security-related issues in android apps: An empirical study. In *ASE Workshops*, pages 15–21. ACM, 2024.

- [124] N. Dayan and S. Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *SIGMOD Conference*, pages 505–520. ACM, 2018.
- [125] T. L. de Santana, P. A. da Mota Silveira Neto, E. S. D. Almeida, and I. Ahmed. Bug analysis in jupyter notebook projects: An empirical study. *ACM TOSEM*, 33(4): 101:1–101:34, 2024.
- [126] D. DeFreez, A. Bhowmick, I. Laguna, and C. Rubio-González. Detecting and reproducing error-code propagation bugs in MPI implementations. In *PPoPP*, pages 187–201. ACM, 2020.
- [127] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. A. Wagner, B. Ray, and Y. Chen. Vulnerability detection with code language models: How far are we? In *ICSE*, pages 1729–1741. IEEE, 2025.
- [128] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. K. Robertson, F. Ulrich, and R. Whelan. LAVA: large-scale automated vulnerability addition. In *IEEE S&P*, pages 110–121. IEEE Computer Society, 2016.
- [129] G. Drosos, T. Sotiropoulos, G. Alexopoulos, D. Mitropoulos, and Z. Su. When your infrastructure is a buggy program: Understanding faults in infrastructure as code ecosystems. *Proc. ACM Program. Lang.*, 8(OOPSLA2):2490–2520, 2024.
- [130] X. Du, Z. Zheng, L. Ma, and J. Zhao. An empirical study on common bugs in deep learning compilers. In *ISSRE*, pages 184–195. IEEE, 2021.
- [131] L. Duboc, D. S. Rosenblum, and T. Wicks. A framework for modelling and analysis of software systems scalability. In *ICSE*, pages 949–952. ACM, 2006.
- [132] L. Duboc, D. S. Rosenblum, and T. Wicks. A framework for characterization and

- analysis of software system scalability. In *ESEC/SIGSOFT FSE*, pages 375–384. ACM, 2007.
- [133] M. Duijn, A. Kucera, and A. Bacchelli. Quality questions need quality code: Classifying code fragments on stack overflow. In *MSR*, pages 410–413. IEEE Computer Society, 2015.
- [134] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu. Empirical review of java program repair tools: a large-scale experiment on 2, 141 bugs and 23, 551 repair attempts. In *FSE*, pages 302–313. ACM, 2019.
- [135] A. Eghbali and M. Pradel. No strings attached: An empirical study of string-related software bugs. In *ASE*, pages 956–967. IEEE, 2020.
- [136] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE TSE*, 28(2):159–182, 2002.
- [137] J. Fan, Y. Li, S. Wang, and T. N. Nguyen. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *MSR*, pages 508–512. ACM, 2020.
- [138] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan. Automated repair of programs from large language models. In *ICSE*, pages 1469–1481. IEEE, 2023.
- [139] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE TSE*, 26(8):797–814, 2000.
- [140] R. Ferenc, Z. Tóth, G. Ladányi, I. Siket, and T. Gyimóthy. A public unified bug dataset for java. In *PROMISE*, pages 12–21. ACM, 2018.
- [141] R. Ferenc, P. Gyimesi, G. Gyimesi, Z. Tóth, and T. Gyimóthy. An automatically created novel bug dataset and its validation in bug prediction. *JSS*, 169:110691, 2020.
- [142] A. D. Franco, H. Guo, and C. Rubio-González. A comprehensive study of real-world numerical bug characteristics. In *ASE*, pages 509–519. IEEE Computer Society, 2017.

- [143] F. Frattini, R. Pietrantuono, and S. Russo. *Reproducibility of Software Bugs*, pages 551–565. Springer International Publishing, Cham, 2016. ISBN 978-3-319-30599-8.
- [144] W. Fu, K. Yang, R. G. Dutta, X. Guo, and G. Qu. Llm4sechw: Leveraging domain-specific large language model for hardware debugging. In *AsianHOST*, pages 1–6. IEEE, 2023.
- [145] J. Gao, X. Yang, Y. Jiang, H. Liu, W. Ying, and X. Zhang. Jbench: a dataset of data races for concurrency testing. In *MSR*, pages 6–9. ACM, 2018.
- [146] Y. Gao, W. Dou, F. Qin, C. Gao, D. Wang, J. Wei, R. Huang, L. Zhou, and Y. Wu. An empirical study on crash recovery bugs in large-scale distributed systems. In *FSE*, pages 539–550. ACM, 2018.
- [147] Y. Gao, Y. He, X. Li, B. Zhao, H. Lin, Y. Liang, J. Zhong, H. Zhang, J. Wang, Y. Lin, and M. Yang. An empirical study on low GPU utilization of deep learning jobs. In *ICSE*. ACM, 2024.
- [148] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, and Q. A. Chen. A comprehensive study of autonomous vehicle bugs. In *ICSE*, pages 385–396. ACM, 2020.
- [149] G. Gay. Challenges in using search-based test generation to identify real faults in mockito. In *SSBSE*, volume 9962 of *Lecture Notes in Computer Science*, pages 231–237, 2016.
- [150] G. Gay. Generating effective test suites by combining coverage criteria. In *SSBSE*, volume 10452 of *Lecture Notes in Computer Science*, pages 65–82. Springer, 2017.
- [151] G. Gay. Detecting real faults in the gson library through search-based unit test generation. In *SSBSE*, volume 11036 of *Lecture Notes in Computer Science*, pages 385–391. Springer, 2018.



- [152] L. Gazzola, L. Mariani, M. Orrù, M. Pezzè, and M. Tappler. Testing software in production environments with data from the field. In *ICST*, pages 58–69. IEEE, 2022.
- [153] A. Ghanbari, S. Benton, and L. Zhang. Practical program repair via bytecode mutation. In *ISSTA*, pages 19–30. ACM, 2019.
- [154] A. Gkortzis, D. Mitropoulos, and D. Spinellis. Vulinoss: a dataset of security vulnerabilities in open-source systems. In *MSR*, pages 18–21. ACM, 2018.
- [155] S. Gong, D. Altinbüken, P. Fonseca, and P. Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *SOSP*, pages 66–83. ACM, 2021.
- [156] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12. IEEE Computer Society, 1986.
- [157] Z. Gu, J. Wu, J. Liu, M. Zhou, and M. Gu. An empirical study on api-misuse bugs in open-source C programs. In *COMPSAC (1)*, pages 11–20. IEEE, 2019.
- [158] H. Guan, Y. Xiao, J. Li, Y. Liu, and G. Bai. A comprehensive study of real-world bugs in machine learning model optimization. In *ICSE*, pages 147–158. IEEE, 2023.
- [159] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In *SoCC*, pages 7:1–7:14. ACM, 2014.
- [160] D. Gupta, K. V. Vishwanath, and A. Vahdat. Diecast: Testing distributed systems with an accurate scale model. In *NSDI*, pages 407–422. USENIX Association, 2008.
- [161] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc, and

- A. Mesbah. BUGSJS: a benchmark and taxonomy of javascript bugs. *Softw. Test. Verif. Reliab.*, 31(4), 2021.
- [162] A. Habib and M. Pradel. How many of all bugs do we find? a study of static bug detectors. In *ASE*, pages 317–328. ACM, 2018.
- [163] X. Han, D. Carroll, and T. Yu. Reproducing performance bug reports in server applications: The researchers’ experiences. *J. Syst. Softw.*, 156:268–282, 2019.
- [164] F. Hassan. Tackling build failures in continuous integration. In *ASE*, pages 1242–1245. IEEE, 2019.
- [165] F. Hassan and X. Wang. Change-aware build prediction model for stall avoidance in continuous integration. In *ESEM*, pages 157–162. IEEE Computer Society, 2017.
- [166] F. Hassan and X. Wang. Hirebuild: an automatic approach to history-driven repair of build scripts. In *ICSE*, pages 1078–1089. ACM, 2018.
- [167] F. Hassan, S. Mostafa, E. S. L. Lam, and X. Wang. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *ESEM*, pages 38–47. IEEE Computer Society, 2017.
- [168] J. Henkel, D. Silva, L. Teixeira, M. d’Amorim, and T. W. Reps. Shipwright: A human-in-the-loop system for dockerfile repair. In *ICSE*, pages 1148–1160. IEEE, 2021.
- [169] S. Herbold et al. A fine-grained data set and analysis of tangling in bug fixing commits. *Empirical Softw. Engg.*, 27(6), 2022. ISSN 1382-3256.
- [170] K. Herzig, S. Just, and A. Zeller. It’s not a bug, it’s a feature: how misclassification impacts bug prediction. In *ICSE*, pages 392–401. IEEE Computer Society, 2013.
- [171] T. Hirsch and B. Hofer. A systematic literature review on benchmarks for evaluating debugging approaches. *JSS*, 192:111423, 2022.

- [172] S. C. Y. Ho, V. Majdinasab, M. Islam, D. E. Costa, E. Shihab, F. Khomh, S. Nadi, and M. Raza. An empirical study on bugs inside pytorch: A replication study. In *ICSME*, pages 220–231. IEEE, 2023.
- [173] T. Holek, M. Bures, and T. Cerný. Review of open software bug datasets. In *World-CIST*, volume 801, pages 3–12. Springer, 2023.
- [174] J. Holmes, I. Ahmed, C. Brindescu, R. Gopinath, H. Zhang, and A. Groce. Using relative lines of code to guide automated test generation for python. *ACM Trans. Softw. Eng. Methodol.*, 29(4):28:1–28:38, 2020.
- [175] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang. Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, 33(8):220:1–220:79, 2024.
- [176] J. Hu, L. Zhang, C. Liu, S. Yang, S. Huang, and Y. Liu. Empirical analysis of vulnerabilities life cycle in golang ecosystem. In *ICSE*, pages 212:1–212:13. ACM, 2024.
- [177] Q. Hu, Z. Ye, Z. Wang, G. Wang, M. Zhang, Q. Chen, P. Sun, D. Lin, X. Wang, Y. Luo, Y. Wen, and T. Zhang. Characterization of large language model development in the datacenter. In *NSDI*. USENIX Association, 2024.
- [178] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang. Evolving paradigms in automated program repair: Taxonomy, challenges, and opportunities. *ACM Comput. Surv.*, 57(2):36:1–36:43, 2025.
- [179] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella. Taxonomy of real faults in deep learning systems. In *ICSE*, pages 1110–1121. ACM, 2020.
- [180] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan. A comprehensive study on deep learning bug characteristics. In *ESEC/SIGSOFT FSE*, pages 510–520. ACM, 2019.

- [181] E. Jabbar, H. Hemmati, and R. Feldt. Investigating execution trace embedding for test case prioritization. In *QRS*, pages 279–290. IEEE, 2023.
- [182] N. Jalbert, C. Pereira, G. Pokam, and K. Sen. Radbench: A concurrency bug benchmark suite. In *HotPar*. USENIX Association, 2011.
- [183] K. Jezek and R. Lipka. Antipatterns causing memory bloat: A case study. In *SANER*, pages 306–315. IEEE Computer Society, 2017.
- [184] M. Jiang, J. Jiang, T. Wu, Z. Ma, X. Luo, and Y. Zhou. Understanding vulnerability inducing commits of the linux kernel. *ACM TOSEM*, 33(7):170:1–170:28, 2024.
- [185] Y. Jiang, H. Liu, N. Niu, L. Zhang, and Y. Hu. Extracting concise bug-fixing patches from human-written patches in version control systems. In *ICSE*, pages 686–698. IEEE, 2021.
- [186] Y. Jiang, H. Liu, X. Luo, Z. Zhu, X. Chi, N. Niu, Y. Zhang, Y. Hu, P. Bian, and L. Zhang. Bugbuilder: An automated approach to building bug repository. *IEEE TSE*, 49(4):1443–1463, 2023.
- [187] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *ICLR*. OpenReview.net, 2024.
- [188] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77–88. ACM, 2012.
- [189] Y. Jin, H. Wang, X. Tang, Z. Guo, Y. Zhao, T. Hoeffler, T. Liu, X. Liu, and J. Zhai. Leveraging graph analysis to pinpoint root causes of scalability issues for parallel applications. *IEEE Trans. Parallel Distributed Syst.*, 36(2):308–325, 2025.
- [190] R. Just, D. Jalali, and M. D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *ISSTA*, pages 437–440. ACM, 2014.

- [191] A. V. Kamienski, L. Palechor, C. Bezemer, and A. Hindle. Pysstubs: Characterizing single-statement bugs in popular open-source python projects. In *MSR*, pages 520–524. IEEE, 2021.
- [192] R. Karampatsis and C. Sutton. How often do single-statement bugs occur?: The manysstubs4j dataset. In *MSR*, pages 573–577. ACM, 2020.
- [193] M. Kellogg, N. Shadab, M. Sridharan, and M. D. Ernst. Lightweight and modular resource leak verification. In *ESEC/SIGSOFT FSE*, pages 181–192. ACM, 2021.
- [194] N. Kerzazi, F. Khomh, and B. Adams. Why do automated builds break? an empirical study. In *ICSME*, pages 41–50. IEEE Computer Society, 2014.
- [195] H. Keshavarz and M. Nagappan. Apachejit: A large dataset for just-in-time defect prediction. In *MSR*, pages 191–195. ACM, 2022.
- [196] J. Kim and S. Hong. Poster: Bugoss: A regression bug benchmark for empirical study of regression fuzzing techniques. In *ICST*, pages 470–473. IEEE, 2023.
- [197] J. Kim, G. An, R. Feldt, and S. Yoo. Ahead of time mutation based fault localisation using statistical inference. In *ISSRE*, pages 253–263. IEEE, 2021.
- [198] J. Kim, M. Kim, and E. Lee. Eench: An energy bug benchmark of ethereum client software. In *MSR*, pages 634–638. ACM, 2022.
- [199] M. Kim, Y. Kim, and E. Lee. Denchmark: A bug benchmark of deep learning-related software. In *MSR*, pages 540–544. IEEE, 2021.
- [200] S. Kim, T. Zimmermann, K. Pan, and E. J. W. Jr. Automatic identification of bug-introducing changes. In *ASE*, pages 81–90. IEEE Computer Society, 2006.
- [201] K. Ku, T. E. Hart, M. Chechik, and D. Lie. A buffer overflow benchmark for software model checkers. In *ASE*, pages 389–392. ACM, 2007.

- [202] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention. In *SOSP*, pages 611–626. ACM, 2023.
- [203] I. Laguna, D. H. Ahn, B. R. de Supinski, T. Gamblin, G. L. Lee, M. Schulz, S. Bagchi, M. Kulkarni, B. Zhou, Z. Chen, and F. Qin. Debugging high-performance computing applications at massive scales. *Commun. ACM*, 58(9):72–81, 2015.
- [204] A. Lamkanfi, J. Pérez, and S. Demeyer. The eclipse and mozilla defect tracking dataset: a genuine dataset for mining bug information. In *MSR*, pages 203–206. IEEE Computer Society, 2013.
- [205] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws. *ACM CSUR*, 26(3):211–254, 1994.
- [206] C. Le Goues, N. J. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of C programs. *IEEE TSE*, 41(12):1236–1256, 2015.
- [207] C. Le Goues, M. Pradel, and A. Roychoudhury. Automated program repair. *Commun. ACM*, 62(12):56–65, 2019.
- [208] C. Le Goues, M. Pradel, A. Roychoudhury, and S. Chandra. Automatic program repair. *IEEE Softw.*, 38(4):22–27, 2021.
- [209] J. Y. Lee, S. Kang, J. Yoon, and S. Yoo. The github recent bugs dataset for evaluating llm-based debugging applications. In *ICST*, pages 442–444. IEEE, 2024.
- [210] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ASPLOS*, pages 517–530. ACM, 2016.

- [211] T. Leesatapornwongsa, C. A. Stuardo, R. O. Suminto, H. Ke, J. F. Lukman, and H. S. Gunawi. Scalability bugs: When 100-node testing is not enough. In *HotOS*, pages 24–29. ACM, 2017.
- [212] K. Levin, N. van Kempen, E. D. Berger, and S. N. Freund. Chatdbg: Augmenting debugging with large language models. *Proc. ACM Softw. Eng.*, 2(FSE):1892–1913, 2025.
- [213] B. Li, P. Su, M. Chabbi, S. Jiao, and X. Liu. Djxperf: Identifying memory inefficiencies via object-centric profiling for java. In *CGO*, pages 81–94. ACM, 2023.
- [214] J. Li, Y. Chen, H. Liu, S. Lu, Y. Zhang, H. S. Gunawi, X. Gu, X. Lu, and D. Li. Pcatch: automatically detecting performance cascading bugs in cloud systems. In *EuroSys*, pages 7:1–7:14. ACM, 2018.
- [215] N. Li, J. Guo, B. Huang, Y. Li, Y. Zhang, C. Li, and W. Huang. TCSEA: efficient localization of busy-wait synchronization bugs for latency-critical applications. *IEEE Trans. Parallel Distributed Syst.*, 35(2):297–309, 2024.
- [216] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *ASID*, pages 25–33. ACM, 2006.
- [217] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, and I. Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *OSDI*, pages 663–679. USENIX Association, 2023.
- [218] J. Liang, M. Yuan, Z. Ding, S. Ma, X. Han, and C. Zhang. Racebench: A triggerable and observable concurrency bug benchmark. In *AsiaCCS*, pages 415–428. ACM, 2023.
- [219] Y. Liang, Y. Lin, X. Song, J. Sun, Z. Feng, and J. S. Dong. gdefects4dl: A dataset of

- general real-world deep learning program defects. In *ICSE-Companion*, pages 90–94. ACM/IEEE, 2022.
- [220] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama. Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *SPLASH*, pages 55–56. ACM, 2017.
- [221] G. Lin, W. Xiao, J. Zhang, and Y. Xiang. Deep learning-based vulnerable function detection: A benchmark. In *ICICS*, volume 11999, pages 219–232. Springer, 2019.
- [222] G. Lin, S. Wen, Q. Han, J. Zhang, and Y. Xiang. Software vulnerability detection using deep neural networks: A survey. *Proc. IEEE*, 108(10):1825–1848, 2020.
- [223] Z. Lin, D. Marinov, H. Zhong, Y. Chen, and J. Zhao. Jacontebe: A benchmark suite of real-world java concurrency bugs (T). In *ASE*, pages 178–189. IEEE Computer Society, 2015.
- [224] R. Liscano, A. Ahmed, J. Robertson, A. Azim, V. Sundaresan, and Y. Chang. A lock contention classifier based on java lock contention anti-patterns. In *ICMLA*, pages 1106–1113. IEEE, 2023.
- [225] K. Liu, J. Zhang, L. Li, A. Koyuncu, D. Kim, C. Ge, Z. Liu, J. Klein, and T. F. Bissyandé. Reliable fix patterns inferred from static checkers for automated program repair. *ACM TOSEM*, 32(4):96:1–96:38, 2023.
- [226] K. Liu, Y. Han, Y. Liu, J. M. Zhang, Z. Chen, F. Sarro, G. Huang, and Y. Ma. Trickybugs: A dataset of corner-case bugs in plausible programs. In *MSR*, pages 113–117. ACM, 2024.
- [227] M. Liu, S. Zhong, W. Bi, Y. Zhang, Z. Chen, Z. Chen, X. Liu, and Y. Ma. A first look at bugs in LLM inference engines. *ACM Trans. Softw. Eng. Methodol.*, 2025. doi: 10.1145/3788873. Just Accepted.



- [228] W. Liu, S. Mondal, and T. P. Chen. An empirical study on the characteristics of database access bugs in java applications. *ACM TOSEM*, 33(7):181:1–181:25, 2024.
- [229] Y. Liu, J. Wang, L. Wei, C. Xu, S. Cheung, T. Wu, J. Yan, and J. Zhang. Droidleaks: a comprehensive database of resource leaks in android apps. *ESE*, 24(6):3435–3483, 2019.
- [230] Z. Liu, Y. Zheng, X. Du, Z. Hu, W. Ding, Y. Miao, and Z. Zheng. Taxonomy of aging-related bugs in deep learning libraries. In *ISSRE*, pages 423–434. IEEE, 2022.
- [231] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Bugs*, 2005.
- [232] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339. ACM, 2008.
- [233] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang. How does regression test prioritization perform in real-world software evolution? In *ICSE*, pages 535–546. ACM, 2016.
- [234] S. M. Lundberg and S. Lee. A unified approach to interpreting model predictions. In *NIPS*, pages 4765–4774, 2017.
- [235] H. Luo and L. Specia. From understanding to utilization: A survey on explainability for large language models. *CoRR*, abs/2401.12874, 2024.
- [236] Q. Luo, K. Moran, D. Poshyvanyk, and M. D. Penta. Assessing test case prioritization on real faults and mutants. In *ICSME*, pages 240–251. IEEE Computer Society, 2018.
- [237] N. Machado, F. Maia, F. Neves, F. Coelho, and J. Pereira. Minha: Large-scale distributed systems testing made practical. In *OPODIS*, volume 153 of *LIPICs*, pages 11:1–11:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

- [238] C. Macho, S. McIntosh, and M. Pinzger. Automatically repairing dependency-related build breakage. In *SANER*, pages 106–117. IEEE Computer Society, 2018.
- [239] F. Madeiral, S. Urli, M. de Almeida Maia, and M. Monperrus. BEARS: an extensible java bug benchmark for automatic program repair studies. In *SANER*, pages 468–478. IEEE, 2019.
- [240] L. Madeyski and M. Kawalerowicz. Continuous defect prediction: the idea and a related dataset. In *MSR*, pages 515–518. IEEE Computer Society, 2017.
- [241] M. Maes-Bermejo, M. Gallego, F. Gortázar, G. Robles, and J. M. González-Barahona. Revisiting the building of past snapshots - a replication and reproduction study. *Empir. Softw. Eng.*, 27(3):65, 2022.
- [242] P. Mahbub, O. Shuvo, and M. M. Rahman. Defectors: A large, diverse python dataset for defect prediction. In *MSR*, pages 393–397. IEEE, 2023.
- [243] A. Makhshari and A. Mesbah. Iot bugs and development challenges. In *ICSE*, pages 460–472. IEEE, 2021.
- [244] C. Mandrioli and M. Maggio. Testing self-adaptive software with probabilistic guarantees on performance metrics. In *ESEC/SIGSOFT FSE*, pages 1002–1014. ACM, 2020.
- [245] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *ESE*, 22(4):1936–1964, 2017.
- [246] S. McGuire, E. Schultz, B. Ayoola, and P. Ralph. Sustainability is stratified: Toward a better theory of sustainable software engineering. In *ICSE*, pages 1996–2008. IEEE, 2023.

- [247] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian. Deepdelta: learning to repair compilation errors. In *ESEC/SIGSOFT FSE*, pages 925–936. ACM, 2019.
- [248] J. Mickens. The night watch. In *USENIX Login Logout (USENIX 13)*, 2013.
- [249] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino. FAST approaches to scalable similarity-based test case prioritization. In *ICSE*, pages 222–232. ACM, 2018.
- [250] G. Mishherghi and Z. Su. HDD: hierarchical delta debugging. In *ICSE*, pages 142–151. ACM, 2006.
- [251] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to java runtime bloat. *IEEE Softw.*, 27(1):56–63, 2010.
- [252] J. Mitra and V. Ranganath. Ghera: A repository of android app vulnerability benchmarks. In *PROMISE*, pages 43–52. ACM, 2017.
- [253] M. Monperrus. Automatic software repair: A bibliography. *ACM CSUR*, 51(1):17:1–17:24, 2018.
- [254] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging AI applications. In *OSDI*, pages 561–577. USENIX Association, 2018.
- [255] M. M. Morovati, A. Nikanjam, F. Khomh, and Z. M. J. Jiang. Bugs in machine learning-based systems: a faultload benchmark. *ESE*, 28(3):62, 2023.
- [256] S. Mukherjee, A. Almanza, and C. Rubio-González. Fixing dependency errors for Python build reproducibility. In *ISSTA*, pages 439–451. ACM, 2021.
- [257] N. Mündler, M. N. Müller, J. He, and M. T. Vechev. Code agents are state of the art software testers. *CoRR*, abs/2406.12952, 2024.

- [258] J. Murphy, E. T. Brady, M. S. I. Shamim, and A. Rahman. A curated dataset of security defects in scientific software projects. In *HotSoS*, pages 13:1–13:2. ACM, 2020.
- [259] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280. USENIX Association, 2008.
- [260] N. K. Nagwani and S. Verma. A comparative study of bug classification algorithms. *IJSEKE*, 24(1):111–138, 2014.
- [261] C. Nebut, F. Fleurey, Y. L. Traon, and J. Jézéquel. Automatic test generation: A use case driven approach. *IEEE Trans. Software Eng.*, 32(3):140–155, 2006.
- [262] A. Neitsch, K. Wong, and M. W. Godfrey. Build system issues in multilanguage software. In *ICSM*, pages 140–149. IEEE Computer Society, 2012.
- [263] E. C. Neto, D. A. da Costa, and U. Kulesza. The impact of refactoring changes on the SZZ algorithm: An empirical study. In *SANER*, pages 380–390. IEEE Computer Society, 2018.
- [264] K. Ngo, T. Nguyen, S. Nguyen, and H. D. Vo. Variability fault localization: a benchmark. In *SPLC (A)*, pages 120–125. ACM, 2021.
- [265] K. Nguyen, K. Wang, Y. Bu, L. Fang, and G. Xu. Understanding and combating memory bloat in managed data-intensive systems. *ACM Trans. Softw. Eng. Methodol.*, 26(4):12:1–12:41, 2018.
- [266] C. Ni, L. Shen, X. Yang, Y. Zhu, and S. Wang. Megavul: A C/C++ vulnerability dataset with comprehensive code representations. In *MSR*, pages 738–742. ACM, 2024.

- [267] G. Nikitopoulos, K. Dritsa, P. Louridas, and D. Mitropoulos. Crossvul: a cross-language vulnerability dataset with commit data. In *FSE*, pages 1565–1569. ACM, 2021.
- [268] A. Nilizadeh, G. T. Leavens, C. S. Pasareanu, and Y. Noller. Jmlkelinci+: Detecting semantic bugs and covering branches with valid inputs using coverage-guided fuzzing and runtime assertion checking. *Formal Aspects Comput.*, 36(1):2:1–2:24, 2024.
- [269] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: detecting performance problems via similar memory-access patterns. In *ICSE*, pages 562–571. IEEE Computer Society, 2013.
- [270] A. Nistor, P. Chang, C. Radoi, and S. Lu. CARMEL: detecting and fixing performance problems that have non-intrusive fixes. In *ICSE (1)*, pages 902–912. IEEE Computer Society, 2015.
- [271] T. B. Noor and H. Hemmati. A similarity-based approach for test case prioritization using historical failure data. In *ISSRE*, pages 58–68. IEEE Computer Society, 2015.
- [272] M. Ohira, Y. Kashiwa, Y. Yamatani, H. Yoshiyuki, Y. Maeda, N. Limsettho, K. Fujino, H. Hata, A. Ihara, and K. Matsumoto. A dataset of high impact bugs: Manually-classified issue reports. In *MSR*, pages 518–521. IEEE Computer Society, 2015.
- [273] L. Oliveira, D. Wilkinson, D. Mossé, and B. Childers. Supporting long-term reproducible software execution. P-RECS’18. ACM, 2018. ISBN 9781450358613.
- [274] P. Oliver, J. Dietrich, C. Anslow, and M. Homer. Crashjs: A nodejs benchmark for automated crash reproduction. In *MSR*, pages 75–87. ACM, 2024.
- [275] A. Orso, S. Joshi, M. Burger, and A. Zeller. Isolating relevant component interactions with jinsi. In *Proceedings of the 2006 International Workshop on Dynamic Systems*

- Analysis*, WODA '06, page 3–10, New York, NY, USA, 2006. Association for Computing Machinery.
- [276] Y. Ouyang, J. Yang, and L. Zhang. Benchmarking automated program repair: An extensive study on both real-world and artificial bugs. In *ISSTA*, pages 440–452. ACM, 2024.
  - [277] J. Pachouly, S. Ahirrao, and K. Kotecha. Sdptool : A tool for creating datasets and software defect predictions. *SoftwareX*, 18:101036, 2022.
  - [278] M. Paltenghi and M. Pradel. Bugs in quantum computing platforms: an empirical study. *Proc. ACM Prog. Lang.*, 6(OOPSLA1):1–27, 2022.
  - [279] M. Paltenghi and M. Pradel. A survey on testing and analysis of quantum software. *CoRR*, abs/2410.00650, 2024.
  - [280] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: fault localization in concurrent programs. In *ICSE (1)*, pages 245–254. ACM, 2010.
  - [281] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini. Split-wise: Efficient generative LLM inference using phase splitting. In *ISCA*. IEEE, 2024.
  - [282] R. Paul, A. K. Turzo, and A. Bosu. A dataset of vulnerable code changes of the chromium OS project. In *ICSE (Companion Volume)*, pages 244–245. IEEE, 2021.
  - [283] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and improving fault localization. In *ICSE*, pages 609–620. IEEE / ACM, 2017.
  - [284] J. D. Pereira, J. H. Antunes, and M. Vieira. A software vulnerability dataset of large open source C/C++ projects. In *PRDC*, pages 152–163. IEEE, 2022.
  - [285] A. Perez, R. Abreu, and A. van Deursen. A theoretical and empirical analysis of program spectra diagnosability. *IEEE TSE*, 47(2):412–431, 2021.

- [286] D. Petriu and G. Somadder. A pattern language for improving the capacity of layered client/server systems with multi-threaded servers. *Proceedings of EuroPLoP'97*, 1997.
- [287] A. Phukan, S. Somasundaram, A. Saxena, K. Goswami, and B. V. Srinivasan. Peering into the mind of language models: An approach for attribution in contextual question answering. In *ACL (Findings)*, pages 11481–11495. Association for Computational Linguistics, 2024.
- [288] R. Pincioli, A. Aleti, and C. Trubiani. Performance modeling and analysis of design patterns for microservice systems. In *ICSA*, pages 35–46. IEEE, 2023.
- [289] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *MSR*, pages 383–387. IEEE / ACM, 2019.
- [290] L. Ponzanelli, A. Mocci, A. Bacchelli, and M. Lanza. Understanding and classifying the quality of technical forum questions. In *QSIIC*, pages 343–352. IEEE, 2014.
- [291] K. D. Pramod, W. T. N. D. Silva, W. U. K. Thabrew, R. Shariffdeen, and S. Wickramanayake. Bugsphp: A dataset for automated program repair in PHP. In *MSR*, pages 128–132. ACM, 2024.
- [292] R. Qin, Z. Li, W. He, J. Cui, F. Ren, M. Zhang, Y. Wu, W. Zheng, and X. Xu. Mooncake: Trading more storage for less computation: A KVCache-centric architecture for serving LLM chatbot. In *FAST*. USENIX Association, 2025.
- [293] A. Radu and S. Nadi. A dataset of non-functional bugs. In *MSR*, pages 399–403. IEEE / ACM, 2019.
- [294] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *PLDI*, pages 335–346. ACM, 2012.

- [295] H. Ren, Y. Li, and L. Chen. An empirical study on critical blocking bugs. In *ICPC*, pages 72–82. ACM, 2020.
- [296] F. Reyes, Y. Gamage, G. Skoglund, B. Baudry, and M. Monperrus. BUMP: A benchmark of reproducible breaking dependency updates. In *SANER*, pages 159–170. IEEE, 2024.
- [297] O. Riganelli, M. Mobilio, D. Micucci, and L. Mariani. A benchmark of data loss bugs for android apps. In *MSR*, pages 582–586. IEEE / ACM, 2019.
- [298] T. Roehm, S. Nosovic, and B. Bruegge. Automated extraction of failure reproduction steps from user interaction traces. In *SANER*, pages 121–130. IEEE Computer Society, 2015.
- [299] A. Romano, X. Liu, Y. Kwon, and W. Wang. An empirical study of bugs in webassembly compilers. In *ASE*, pages 42–54. IEEE, 2021.
- [300] I. Rotaru and A. Vescan. Test case prioritization based on neural network classification with artifacts traceability. In *ASEW*, pages 78–87. IEEE, 2023.
- [301] B. Ruan, J. Liu, W. Zhao, and Z. Liang. Vulzoo: A comprehensive vulnerability intelligence dataset. In *ASE*, pages 2334–2337. ACM, 2024.
- [302] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empir. Softw. Eng.*, 14(2):131–164, 2009.
- [303] N. Saavedra, A. Silva, and M. Monperrus. Gitbug-actions: Building reproducible bug-fix benchmarks with github actions. In *ICSE Companion*, pages 1–5. ACM, 2024.
- [304] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad. Bugs.jar: a large-scale, diverse dataset of real-world java bugs. In *MSR*, pages 10–13. ACM, 2018.



- [305] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, and S. Segura. TANDEM: A taxonomy and a dataset of real-world performance bugs. *IEEE Access*, 8:107214–107228, 2020.
- [306] A. Sanders, G. S. Walia, and A. A. Allen. Analysis of software vulnerabilities introduced in programming submissions across curriculum at two higher education institutions. In *FIE*, pages 1–9. IEEE, 2024.
- [307] Q. I. Sarhan, B. Vancsics, and Á. Beszédes. Method calls frequency-based tie-breaking strategy for software fault localization. In *SCAM*, pages 103–113. IEEE, 2021.
- [308] Q. I. Sarhan, T. Gergely, and Á. Beszédes. New ranking formulas to improve spectrum based fault localization via systematic search. In *ICST Workshops*, pages 306–309. IEEE, 2022.
- [309] M. Schlichtig, S. Sassalla, K. Narasimhan, and E. Bodden. Introducing FUM: A framework for API usage constraint and misuse classification. In *Software Engineering*, volume P-332, pages 105–106. Gesellschaft für Informatik e.V., 2023.
- [310] M. Selakovic and M. Pradel. Performance issues and optimizations in javascript: An empirical study. In *Software Engineering*, volume P-267, page 63. GI, 2017.
- [311] J. Senanayake, H. K. Kalutarage, M. O. Al-Kadri, L. Piras, and A. Petrovski. Labelled vulnerability dataset on android source code (lvdandro) to develop ai-based code vulnerability detection models. In *SECRYPT*, pages 659–666. SCITEPRESS, 2023.
- [312] H. Seo, C. Sadowski, S. G. Elbaum, E. Aftandilian, and R. W. Bowdidge. Programmers’ build errors: a case study (at google). In *ICSE*, pages 724–734. ACM, 2014.
- [313] K. Serebryany. OSS-Fuzz - google’s continuous fuzzing service for open source software. Vancouver, BC, Aug. 2017. USENIX Association.

- [314] A. Shahoor, A. Y. Khamit, J. Yi, and D. Kim. Leakpair: Proactive repairing of memory leaks in single page web applications. In *ASE*, pages 1175–1187. IEEE, 2023.
- [315] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (T). In *ASE*, pages 201–211. IEEE Computer Society, 2015.
- [316] S. Shao, Z. Qiu, X. Yu, W. Yang, G. Jin, T. Xie, and X. Wu. Database-access performance antipatterns in database-backed web applications. In *ICSME*, pages 58–69. IEEE, 2020.
- [317] Q. Shen, H. Ma, J. Chen, Y. Tian, S. Cheung, and X. Chen. A comprehensive study of deep learning compiler bugs. In *FSE*, pages 968–980. ACM, 2021.
- [318] A. Shi, T. Yung, A. Gyori, and D. Marinov. Comparing and combining test-suite reduction and regression test selection. In *ESEC/SIGSOFT FSE*, pages 237–247. ACM, 2015.
- [319] R. Shi, Y. Gan, and Y. Wang. Evaluating scalability bottlenecks by workload extrapolation. In *MASCOTS*, pages 333–347. IEEE Computer Society, 2018.
- [320] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: language agents with verbal reinforcement learning. In *NeurIPS*, 2023.
- [321] A. Silva, N. Saavedra, and M. Monperrus. Gitbug-java: A reproducible benchmark of recent java bugs. In *MSR*, pages 118–122. ACM, 2024.
- [322] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.
- [323] C. U. Smith and L. G. Williams. Software performance antipatterns. In *Workshop on Software and Performance*, pages 127–136. ACM, 2000.

- [324] C. U. Smith and L. G. Williams. New software performance antipatterns: More ways to shoot yourself in the foot. In *Int. CMG Conference*, pages 667–674. Computer Measurement Group, 2002.
- [325] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *SANER*, pages 130–140. IEEE Computer Society, 2018.
- [326] J. Sohn and S. Yoo. FLUCCS: using code and change metrics to improve fault localization. In *ISSTA*, pages 273–283. ACM, 2017.
- [327] L. Song and S. Lu. Performance diagnosis for inefficient loops. In *ICSE*, pages 370–380. IEEE / ACM, 2017.
- [328] X. Song, Y. Lin, S. H. Ng, Y. Wu, X. Peng, J. S. Dong, and H. Mei. Regminer: towards constructing a large regression dataset from code evolution history. In *ISSTA*, pages 314–326. ACM, 2022.
- [329] X. Song, Y. Wu, J. Cao, B. Chen, Y. Lin, Z. Lu, D. Wang, and X. Peng. Bugminer: Automating precise bug dataset construction by code evolution history mining. In *ASE*, pages 1919–1929. IEEE, 2023.
- [330] Ó. Soto-Sánchez, M. Maes-Bermejo, M. Gallego, and F. Gortázar. A dataset of regressions in web applications detected by end-to-end tests. *Softw. Qual. J.*, 30(2):425–454, 2022.
- [331] C. Soto-Valero, T. Durieux, and B. Baudry. A longitudinal analysis of bloated java dependencies. In *ESEC/SIGSOFT FSE*, pages 1021–1031. ACM, 2021.
- [332] A. L. Strauss and J. M. Corbin. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. Sage Publications, 1990.

- [333] C. A. Stuardo, T. Leesatapornwongsa, R. O. Suminto, H. Ke, J. F. Lukman, W. Chuang, S. Lu, and H. S. Gunawi. Scalecheck: A single-machine approach for discovering scalability bugs in large distributed systems. In *FAST*, pages 359–373. USENIX Association, 2019.
- [334] M. Sulír and J. Porubán. A quantitative study of java software buildability. In *PLATEAU@SPLASH*, pages 17–25. ACM, 2016.
- [335] M. Sullivan and R. Chillarege. Software defects and their impact on system availability: A study of field failures in operating systems. In *FTCS*, pages 2–9. IEEE Computer Society, 1991.
- [336] T. R. Sumers, S. Yao, K. Narasimhan, and T. L. Griffiths. Cognitive architectures for language agents. *Trans. Mach. Learn. Res.*, 2024, 2024.
- [337] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su. Perses: syntax-guided program reduction. In *ICSE*, pages 361–371. ACM, 2018.
- [338] C. Sun, H. Jin, S. Wu, A. Fu, Z. Wang, and W. K. Chan. Identifying metamorphic relations: A data mutation directed approach. *SPE*, 54(3):394–418, 2024.
- [339] M. Sundararajan, A. Taly, and Q. Yan. Axiomatic attribution for deep networks. In *ICML*, volume 70 of *Proceedings of Machine Learning Research*, pages 3319–3328. PMLR, 2017.
- [340] F. Tambon, A. Nikanjam, L. An, F. Khomh, and G. Antoniol. Silent bugs in deep learning frameworks: an empirical study of keras and tensorflow. *ESE*, 29(1):10, 2024.
- [341] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *ICSE Companion*, pages 180–182. IEEE Computer Society, 2017.

- [342] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury. Repairing crashes in android apps. In *ICSE*, pages 187–198. ACM, 2018.
- [343] D. Tarlow, S. Moitra, A. Rice, Z. Chen, P. Manzagol, C. Sutton, and E. Aftandilian. Learning to fix build errors with graph2diff neural networks. In *ICSE (Workshops)*, pages 19–20. ACM, 2020.
- [344] M. E. A. Tebib, O. Aktouf, P. André, and M. Graa. Privbench: A benchmark capturing privilege escalation vulnerabilities in android. In *PST*, pages 1–12. IEEE, 2024.
- [345] D. H. T. That, G. Fils, Z. Yuan, and T. Malik. Sciunits: Reusable research objects. In *eScience*, pages 374–383. IEEE Computer Society, 2017.
- [346] H. Tian, K. Liu, Y. Li, A. K. Kaboré, A. Koyuncu, A. Habib, L. Li, J. Wen, J. Klein, and T. F. Bissyandé. The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches. *ACM TOSEM*, 32(4): 92:1–92:34, 2023.
- [347] H. Tian, W. Lu, T. O. Li, X. Tang, S. Cheung, J. Klein, and T. F. Bissyandé. Is chatgpt the ultimate programming assistant - how far is it? *CoRR*, abs/2304.11938, 2023.
- [348] R. Tian, Y. Ye, Y. Qin, X. Cong, Y. Lin, Y. Pan, Y. Wu, H. Hui, W. Liu, Z. Liu, and M. Sun. Debugbench: Evaluating debugging capability of large language models. In *ACL (Findings)*, pages 4173–4198. Association for Computational Linguistics, 2024.
- [349] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. In *ICSE*, pages 386–396. IEEE Computer Society, 2012.
- [350] C. S. Timperley, G. van der Hoorn, A. Santos, H. Deshpande, and A. Wasowski. ROBUST: 221 bugs in the robot operating system. *ESE*, 29(3):57, 2024.

- [351] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González. Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes. In *ICSE*, pages 339–349. IEEE / ACM, 2019.
- [352] C. Trubiani, R. Pincioli, A. Biaggi, and F. A. Fontana. Automated detection of software performance antipatterns in java-based applications. *IEEE Trans. Software Eng.*, 49(4):2873–2891, 2023.
- [353] M. Tufano, F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk. There and back again: Can you compile that snapshot? *J. Softw. Evol. Process.*, 29(4), 2017.
- [354] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM TOSEM*, 28(4):19:1–19:29, 2019.
- [355] M. Turpin, J. Michael, E. Perez, and S. R. Bowman. Language models don’t always say what they think: Unfaithful explanations in chain-of-thought prompting. In *NeurIPS*, 2023.
- [356] C. Vassallo, S. Proksch, T. Zemp, and H. C. Gall. Un-break my build: assisting developers with build repair hints. In *ICPC*, pages 41–51. ACM, 2018.
- [357] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017.
- [358] R. G. Vieira, A. da Silva, L. S. Rocha, and J. P. P. Gomes. From reports to bug-fix commits: A 10 years dataset of bug-fixing activity from 55 apache’s open source projects. In *PROMISE*, pages 80–89. ACM, 2019.
- [359] C. Wang, J. Liu, X. Peng, Y. Liu, and Y. Lou. Boosting static resource leak detection

- via llm-based resource-oriented intention inference. In *ICSE*, pages 2905–2917. IEEE, 2025.
- [360] D. Wang, S. Li, G. Xiao, Y. Liu, and Y. Sui. An exploratory study of autopilot software bugs in unmanned aerial vehicles. In *FSE*, pages 20–31. ACM, 2021.
- [361] G. Wang, R. Shen, J. Chen, Y. Xiong, and L. Zhang. Probabilistic delta debugging. In *ESEC/SIGSOFT FSE*, pages 881–892. ACM, 2021.
- [362] G. Wang, Z. Wang, J. Chen, X. Chen, and M. Yan. An empirical study on numerical bugs in deep learning programs. In *ASE*, pages 173:1–173:5. ACM, 2022.
- [363] H. Wang, Y. Lin, Z. Yang, J. Sun, Y. Liu, J. S. Dong, Q. Zheng, and T. Liu. Explaining regressions via alignment slicing and mending. *IEEE TSE*, 47(11):2421–2437, 2021.
- [364] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei. A comprehensive study on real world concurrency bugs in node.js. In *ASE*, pages 520–531. IEEE Computer Society, 2017.
- [365] P. Wang, C. Brown, J. A. Jennings, and K. T. Stolee. An empirical study on regular expression bugs. In *MSR*, pages 103–113. ACM, 2020.
- [366] P. Wang, C. Brown, J. A. Jennings, and K. T. Stolee. Demystifying regular expression bugs. *ESE*, 27(1):21, 2022.
- [367] S. Wang, T. Chen, and A. E. Hassan. How do users revise answers on technical q&a websites? A case study on stack overflow. *IEEE TSE*, 46(9):1024–1038, 2020.
- [368] X. Wang, R. Hu, C. Gao, X. Wen, Y. Chen, and Q. Liao. Reposvul: A repository-level high-quality vulnerability dataset. In *ICSE Companion*, pages 472–483. ACM, 2024.
- [369] Y. Wang, M. Kapritsos, L. Schmidt, L. Alvisi, and M. Dahlin. Exalt: Empowering researchers to evaluate large-scale storage systems. In *NSDI*, pages 129–141. USENIX Association, 2014.

- [370] Y. Wang, X. Chen, Y. Huang, H.-N. Zhu, J. Bian, and Z. Zheng. An empirical study on real bug fixes from solidity smart contract projects. *JSS*, 204:111787, 2023.
- [371] Y. Wang, Z. Zhou, Z. Ren, D. Liu, and H. Jiang. A comprehensive study of webassembly runtime bugs. In *SANER*, pages 355–366. IEEE, 2023.
- [372] Y. Wang, W. Zou, R. Geng, and J. Jia. Tracllm: A generic framework for attributing long context llms. In *USENIX Security Symposium*, pages 3845–3864. USENIX Association, 2025.
- [373] Z. Wang, D. Bu, N. Wang, S. Yu, S. Gou, and A. Sun. An empirical study on bugs in javascript engines. *Inf. Softw. Technol.*, 155:107105, 2023.
- [374] M. Waseem, T. Das, A. Ahmad, P. Liang, and T. Mikkonen. Issues and their causes in webassembly applications: An empirical study. In *EASE*, pages 170–180. ACM, 2024.
- [375] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.
- [376] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung. Context-aware patch generation for better automated program repair. In *ICSE*, pages 1–11. ACM, 2018.
- [377] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao, S. Han, and S. Cheung. Historical spectrum based fault localization. *IEEE TSE*, 47(11):2348–2368, 2021.
- [378] T. Wendland, J. Sun, J. Mahmud, S. M. H. Mansur, S. Huang, K. Moran, J. Rubin, and M. Fazzini. Andror2: A dataset of manually-reproduced bug reports for android apps. In *MSR*, pages 600–604. IEEE, 2021.
- [379] R. Widiasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh.



- Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *FSE*, pages 1556–1560. ACM, 2020.
- [380] C. C. Williams and J. Spacco. SZZ revisited: verifying when changes induce fixes. In *DEFECTS*, pages 32–36. ACM, 2008.
- [381] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Trans. Software Eng.*, 42(8):707–740, 2016.
- [382] S. Wu, K. Qiu, and Z. Zheng. An Empirical Study on Environmental Factors for Reproducing Concurrent Software Failures. *Proc. - Annu. Reliab. Maintainab. Symp.*, 2021. ISSN 0149144X.
- [383] X. Wu, W. Zheng, X. Chen, F. Wang, and D. Mu. Cve-assisted large-scale security bug report dataset construction method. *JSS*, 160, 2020.
- [384] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah. How effective are neural networks for fixing security vulnerabilities. In *ISSTA*, pages 1282–1294. ACM, 2023.
- [385] Y. Wu, Z. Li, J. M. Zhang, and Y. Liu. Condefects: A complementary dataset to address the data leakage concern for llm-based fault localization and program repair. In *FSE Companion*, pages 642–646. ACM, 2024.
- [386] C. S. Xia and L. Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for \$0.42 each using chatgpt. In *ISSTA*, pages 819–831. ACM, 2024.
- [387] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang. Demystifying llm-based software engineering agents. *Proc. ACM Softw. Eng.*, 2(FSE):801–824, 2025.
- [388] Q. Xin and S. P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *ISSTA*, pages 226–236. ACM, 2017.

- [389] Q. Xin and S. P. Reiss. Leveraging syntax-related code for automated program repair. In *ASE*, pages 660–670. IEEE Computer Society, 2017.
- [390] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *ICSE*, pages 416–426. IEEE / ACM, 2017.
- [391] Y. Xiong, M. Xu, T. Su, J. Sun, J. Wang, H. Wen, G. Pu, J. He, and Z. Su. An empirical study of functional bugs in android apps. In *ISSTA*, pages 1319–1331. ACM, 2023.
- [392] G. Xu and A. Rountev. Precise memory leak detection for java software using container profiling. *ACM Trans. Softw. Eng. Methodol.*, 22(3):17:1–17:28, 2013.
- [393] G. Xu, M. D. Bond, F. Qin, and A. Rountev. Leakchaser: helping programmers narrow down causes of memory leaks. In *PLDI*, pages 270–282. ACM, 2011.
- [394] J. Xu, L. Yan, F. Wang, and J. Ai. A github-based data collection method for software defect prediction. In *DSA*, pages 100–108. IEEE, 2019.
- [395] J. Xu, K. Lu, Z. Du, Z. Ding, L. Li, Q. Wu, M. Payer, and B. Mao. Silent bugs matter: A study of compiler-introduced security bugs. In *USENIX Security*, pages 3655–3672. USENIX Association, 2023.
- [396] Q. Xu, Y. Gao, and J. Wei. An empirical study on kubernetes operator bugs. In *ISSTA*, pages 1746–1758. ACM, 2024.
- [397] X. Xu, C. Zhu, and Y. Li. Compsuite: A dataset of java library upgrade incompatibility issues. In *ASE*, pages 2098–2101. IEEE, 2023.
- [398] A. S. Yadav and J. N. Wilson. BOSS: A dataset to train ml-based systems to repair programs with out-of-bounds write flaws. In *APR*, pages 26–33. IEEE, 2024.
- [399] G. Yan, S. Chen, Y. Bai, and X. Li. Can deep learning models learn the vulnerable patterns for vulnerability detection? In *COMPSAC*, pages 904–913. IEEE, 2022.

- [400] Y. Yan, S. Jiang, S. Zhang, and Y. Huang. CSFL: fault localization on real software bugs based on the combination of context and spectrum. In *SETTA*, volume 13071 of *Lecture Notes in Computer Science*, pages 219–238. Springer, 2021.
- [401] J. Yang, P. Subramaniam, S. Lu, C. Yan, and A. Cheung. How *not* to structure your database-backed web applications: a study of performance bugs in the wild. In *ICSE*, pages 800–810. ACM, 2018.
- [402] J. Yang, C. Yan, C. Wan, S. Lu, and A. Cheung. View-centric performance optimization for database-backed web applications. In *ICSE*, pages 994–1004. IEEE / ACM, 2019.
- [403] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In *NeurIPS*, 2024.
- [404] Y. Yang, X. Xia, D. Lo, and J. C. Grundy. A survey on deep learning for software engineering. *ACM CSUR*, 54(10s):206:1–206:73, 2022.
- [405] H. Ye, Z. Chen, and C. Le Goues. Precisebugcollector: Extensible, executable and precise bug-fix collection: Solution for challenge 8: Automating precise data collection for code snippets with bugs, fixes, locations, and types. In *ASE*, pages 1899–1910. IEEE, 2023.
- [406] W. Ye, J. Xia, S. Feng, X. Zhong, S. Yuan, and Z. Guan. Fixgpt: A novel three-tier deep learning model for automated program repair. In *DSC*, pages 499–505. IEEE, 2023.
- [407] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, pages 159–172. ACM, 2011.

- [408] G. Yu, J. S. Jeong, G. Kim, S. Kim, and B. Chun. Orca: A distributed serving system for transformer-based generative models. In *OSDI*, pages 521–538. USENIX Association, 2022.
- [409] J. Yu, X. Xie, C. Zhang, S. Chen, Y. Li, and W. Shen. Bugs in pods: Understanding bugs in container runtime systems. In *ISSTA*, pages 1364–1376. ACM, 2024.
- [410] S. Yu and Z. Wang. An empirical study on bugs in rust programming language. In *QRS*, pages 296–305. IEEE, 2024.
- [411] T. Yu and M. Pradel. Syncprof: detecting, localizing, and optimizing synchronization bottlenecks. In *ISSTA*, pages 389–400. ACM, 2016.
- [412] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *OSDI*, pages 249–265. USENIX Association, 2014.
- [413] T. Yuan, G. Li, J. Lu, C. Liu, L. Li, and J. Xue. Gobench: A benchmark suite of real-world go concurrency bugs. In *CGO*, pages 187–199. IEEE, 2021.
- [414] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *MSR*, pages 199–208. IEEE Computer Society, 2012.
- [415] D. Zan, Z. Huang, W. Liu, H. Chen, L. Zhang, S. Xin, L. Chen, Q. Liu, X. Zhong, A. Li, S. Liu, Y. Xiao, L. Chen, Y. Zhang, J. Su, T. Liu, R. Long, K. Shen, and L. Xiang. Multi-swe-bench: A multilingual benchmark for issue resolving. *CoRR*, abs/2504.02605, 2025.
- [416] A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC / SIGSOFT FSE*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 1999.

- [417] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.
- [418] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input: A retrospective on delta debugging. *IEEE Trans. Software Eng.*, 51(3):820–824, 2025.
- [419] M. Zeng, Y. Wu, Z. Ye, Y. Xiong, X. Zhang, and L. Zhang. Fault localization via efficient probabilistic modeling of program semantics. In *ICSE*, pages 958–969. ACM, 2022.
- [420] Y. Zeng, M. Chao, and R. Stoleru. Emuedge: A hybrid emulator for reproducible and realistic edge computing experiments. In *ICFC*, pages 153–164. IEEE, 2019.
- [421] C. Zhang, B. Chen, L. Chen, X. Peng, and W. Zhao. A large-scale empirical study of compiler errors in continuous integration. In *FSE*, pages 176–187. ACM, 2019.
- [422] C. Zhang, J. Li, D. Li, and X. Lu. Understanding and statically detecting synchronization performance bugs in distributed cloud systems. *IEEE Access*, 7:99123–99135, 2019.
- [423] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei. A survey on bug-report analysis. *Sci. China Inf. Sci.*, 58(2):1–24, 2015.
- [424] J. Zhang, F. Wang, S. Qiu, Y. Wang, J. Ou, J. Huang, B. Li, P. Fang, and D. Feng. Scavenger: Better space-time trade-offs for key-value separated lsm-trees. In *ICDE*, pages 4072–4085. IEEE, 2024.
- [425] M. Zhang, X. Li, L. Zhang, and S. Khurshid. Boosting spectrum-based fault localization using pagerank. In *ISSTA*, pages 261–272. ACM, 2017.
- [426] M. Zhang, Z. Xu, Y. Tian, X. Cheng, and C. Sun. Toward a better understanding of probabilistic delta debugging. In *ICSE*, pages 2024–2035. IEEE, 2025.

- [427] P. Zhang, F. Xiao, and X. Luo. A framework and dataset for bugs in ethereum smart contracts. In *ICSME*, pages 139–150. IEEE, 2020.
- [428] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen. A survey of learning-based automated program repair. *ACM TOSEM*, 33(2):55:1–55:69, 2024.
- [429] Q. Zhang, C. Fang, W. Sun, Y. Liu, T. He, X. Hao, and Z. Chen. APPT: boosting automated patch correctness prediction via fine-tuning pre-trained models. *IEEE TSE*, 50(3):474–494, 2024.
- [430] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang. An empirical study on program failures of deep learning jobs. In *ICSE*, pages 1159–1170. ACM, 2020.
- [431] X. Zhang, R. Yan, J. Yan, B. Cui, J. Yan, and J. Zhang. Excepy: A python benchmark for bugs with python built-in types. In *SANER*, pages 856–866. IEEE, 2022.
- [432] Y. Zhang, Y. Chen, S. Cheung, Y. Xiong, and L. Zhang. An empirical study on tensorflow program bugs. In *ISSTA*, pages 129–140. ACM, 2018.
- [433] Y. Zhang, S. Cao, H. Wang, Z. Chen, X. Luo, D. Mu, Y. Ma, G. Huang, and X. Liu. Characterizing and detecting webassembly runtime bugs. *ACM TOSEM*, 33(2):37:1–37:29, 2024.
- [434] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury. Autocoderover: Autonomous program improvement. In *ISSTA*, pages 1592–1604. ACM, 2024.
- [435] Z. Zhang, Z. Patterson, M. Hicks, and S. Wei. FIXREVERTER: A realistic bug injection methodology for benchmarking fuzz testing. In *USENIX Security*, pages 3699–3715. USENIX Association, 2022.
- [436] Z. Zhang, J. Xue, D. Yang, and X. Mao. Contextaug: model-domain failing test augmentation with contextual information. *FCS*, 18(2):182202, 2024.

- [437] G. Zhao, S. Georgiou, Y. Zou, S. Hassan, D. Truong, and T. Corbin. Enhancing performance bug prediction using performance code metrics. In *MSR*, pages 50–62. ACM, 2024.
- [438] H. Zhao, H. Chen, F. Yang, N. Liu, H. Deng, H. Cai, S. Wang, D. Yin, and M. Du. Explainability for large language models: A survey. *ACM Trans. Intell. Syst. Technol.*, 15(2):20:1–20:38, 2024.
- [439] P. Zhao, Z. Miao, S. Lan, and J. Zhao. Bugs4q: A benchmark of existing bugs to enable controlled testing and debugging studies for quantum programs. *JSS*, 205: 111805, 2023.
- [440] P. Zhao, X. Wu, J. Luo, Z. Li, and J. Zhao. An empirical study of bugs in quantum machine learning frameworks. In *QSW*, pages 68–75. IEEE, 2023.
- [441] Y. Zhao, L. Xiao, A. B. Bondi, B. Chen, and Y. Liu. A large-scale empirical study of real-life performance issues in open source projects. *IEEE TSE*, 49(2):924–946, 2023.
- [442] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. W. Barrett, and Y. Sheng. Sglang: Efficient execution of structured language model programs. In *NeurIPS*, 2024.
- [443] Y. Zheng, S. Pujar, B. L. Lewis, L. Buratti, E. A. Epstein, B. Yang, J. Laredo, A. Morari, and Z. Su. D2A: A dataset built for ai-based vulnerability detection methods using differential analysis. In *ICSE (SEIP)*, pages 111–120. IEEE, 2021.
- [444] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang. An overview of blockchain technology: Architecture, consensus, and future trends. In *BigData Congress*, pages 557–564. IEEE Computer Society, 2017.
- [445] L. Zhong, Z. Wang, and J. Shang. Debug like a human: A large language model

- debugger via verifying runtime execution step by step. In *ACL (Findings)*, pages 851–870. Association for Computational Linguistics, 2024.
- [446] W. Zhong, C. Li, Y. Zhang, Z. Ge, J. Wang, J. Ge, and B. Luo. An automated and flexible multilingual bug-fix dataset construction system. In *ASE*, pages 1881–1886. IEEE, 2023.
- [447] W. Zhong, C. Li, K. Liu, J. Ge, B. Luo, T. F. Bissyandé, and V. Ng. Benchmarking and categorizing the performance of neural program repair systems for java. *ACM Trans. Softw. Eng. Methodol.*, 34(1):11:1–11:35, 2025.
- [448] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *OSDI*. USENIX Association, 2024.
- [449] B. Zhou, M. Kulkarni, and S. Bagchi. Vrisha: using scaling properties of parallel programs for bug detection and localization. In *HPDC*, pages 85–96. ACM, 2011.
- [450] F. Zhou, Y. Gan, S. Ma, and Y. Wang. wperf: Generic off-cpu analysis to identify bottleneck waiting events. In *OSDI*, pages 527–543. USENIX Association, 2018.
- [451] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *ICSE*, pages 14–24. IEEE Computer Society, 2012.
- [452] W. Zhou, J. Zhang, J. Sun, and G. Sun. Using small-scale history data to predict large-scale performance of HPC application. In *IPDPS Workshops*, pages 787–795. IEEE, 2020.
- [453] W. Zhou, H. Adel, H. Schuff, and N. T. Vu. Explaining pre-trained language models with attribution scores: An analysis in low-resource settings. In *LREC/COLING*, pages 6867–6875. ELRA and ICCL, 2024.



- [454] X. Zhou, X. Peng, T. Xie, J. Sun, W. Li, C. Ji, and D. Ding. Delta debugging microservice systems. In *ASE*, pages 802–807. ACM, 2018.
- [455] H.-N. Zhu and C. Rubio-González. On the reproducibility of software defect datasets. In *ICSE*, pages 2324–2335. IEEE, 2023.
- [456] H.-N. Zhu, K. Z. Guan, R. M. Furth, and C. Rubio-González. ActionsRemaker: Reproducing GitHub actions. In *ICSE Companion*, pages 11–15. IEEE, 2023.
- [457] H.-N. Zhu, R. M. Furth, M. Pradel, and C. Rubio-González. From bugs to benchmarks: A comprehensive survey of software defect datasets, Oct. 2025. URL <https://doi.org/10.5281/zenodo.17402612>.
- [458] H.-N. Zhu, G. Ayinmode, C. A. Stuardo, H. S. Gunawi, and C. Rubio-González. Understanding and detecting scalability faults in large-scale distributed systems. 2026.
- [459] H.-N. Zhu, R. M. Furth, M. Pradel, and C. Rubio-González. From bugs to benchmarks: A comprehensive survey of software defect datasets. *ACM Comput. Surv.*, 58(10): 249:1–249:39, 2026.
- [460] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss. What makes a good bug report? *IEEE TSE*, 36(5):618–643, 2010.
- [461] M. Zolfagharinia, B. Adams, and Y. Guéhéneuc. Do not trust build results at face value: an empirical study of 30 million CPAN builds. In *MSR*, pages 312–322. IEEE Computer Society, 2017.