Intelligent Detection and Diagnosis of Anomalies in Software Delivery Pipelines

by

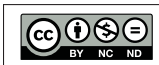
Henri AÏDASSO

MANUSCRIPT-BASED THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
Ph.D.

MONTREAL, APRIL 20, 2026

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

Henri Aïdasso, 2026



This Creative Commons license allows readers to download this work and share it with others as long as the author is credited. The content of this work cannot be modified in any way or used commercially.

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Mr. Francis Bordeleau, Thesis Supervisor
Department of Software and IT Engineering, École de technologie supérieure

Mr. Erik Andrew Poirier, Chair, Board of Examiners
Department of Construction Engineering, École de technologie supérieure

Mr. Fabio Petrillo, Member of the Jury
Department of Software and IT Engineering, École de technologie supérieure

Mr. Benoit Baudry, External Independent Examiner
Department of Computer Science and Operations Research, University of Montreal

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON MARCH 30, 2026

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

DEDICATION

À mon enfant, pour ce rayon de lumière que tu nous as apporté.

À mon épouse, mon amour et ma meilleure amie, Issifath, pour ton soutien inestimable.

À ma mère et à mon père, Alphonsine et Rémi, pour vos prières et votre exemple de résilience.

À mes frères, David et Jean-Christ, pour l'inspiration que vous ne cessez d'être.

À ma famille, d'ici et d'ailleurs, de sang et de cœur . . . pour tout.

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my thesis supervisor, Prof. Francis Bordeleau, for the immeasurable trust you placed in me from the very beginning of this doctoral work. I sincerely thank you for your guidance, support, exceptional availability, and constant encouragement, all of which greatly contributed to this work. I am especially grateful for your humanity, attentive listening, and understanding, which helped me move forward even during the most challenging moments of this journey. I am inspired by your communication skills, your integrity, and your unwavering dedication to truly bridging the gap between academic research and industry.

On the same note, I would like to sincerely thank Dr. Ali Tizghadam from TELUS, for supporting this project and for providing me with all the resources to ensure its success, for always challenging my limits, and for pushing me to give my very best.

This work would not have been possible without the support of TELUS, Mitacs, the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Arbour Foundation, whose research funding allowed me to dedicate myself fully to my thesis. Thank you sincerely.

I am also sincerely grateful to the members of the jury for agreeing to evaluate this thesis, and in particular to Prof. Fabio Petrillo, whose words of encouragement meant a great deal to me.

I would like to express my heartfelt thanks to Prof. Mickael Foursov from the University of Rennes 1, who tirelessly encouraged me to pursue the doctoral path, and to Dr. Aymeric Hervieu, whose trust and support over the past years cannot be fully expressed in words.

To all my colleagues at the ÉTS Industrial Research Chair in DevOps, especially Samah Kansab, thank you for this journey together, for the collaborations, the stimulating discussions, the laughter, the venting, the cheering, and all the moments we shared. You have made my doctoral experience so much more enjoyable. Thank you sincerely!

My gratitude also goes to my friends who, at one point or another, listened, provided me with support, or lifted my spirits, especially Dr. André Soro, Isbath Sanni, Rudi Ahissou, and Éliot

Bankolé, to name but a few. A very special word of thanks to my cousin Donald Aïdasso for being there through thick and thin and always showing up when it mattered.

I would like to thank my family, father, mother, and brothers, who have always supported me in every possible way to reach ever higher throughout all these years. You are the foundation of the person I am today, and I hope this work makes you proud.

Last but not least, my deepest gratitude goes to my wife, Issifath. I could not have done this without you. Thank you for listening during turbulent times, for celebrating the small wins, for putting up with years of talk about AI and CI, for being my first reviewer, and for every life chapter we've written together alongside this one. Thank God Almighty, we have made it!

Détection et Diagnostic Intelligents des Anomalies dans les Pipelines de Livraison Logicielle

Henri AÏDASSO

RÉSUMÉ

Les organisations modernes ont massivement adopté des pratiques d'automatisation DevOps leur permettant d'effectuer des livraisons fréquentes et rapides de versions logicielles tout en s'assurant de la qualité du code produit par les développeurs. Ces pratiques sont mises en œuvre principalement grâce aux pipelines d'intégration continue et de déploiement continu (CI/CD) dont les résultats d'exécutions servent à certifier la qualité logicielle et confirmer la réussite de déploiements en cas de « succès », et à signaler des problèmes de code aux développeurs en cas d'« échec ». Ainsi, l'utilité des pipelines de CI/CD repose essentiellement sur des résultats d'exécutions fiables et déterministes. En pratique, toutefois, des anomalies liées aux environnements complexes et distribués induisent des résultats de pipelines inattendus et trompeurs (c'est-à-dire de fausses alarmes dites « échecs intermittents », voire pire, des succès illusoires dits « échecs silencieux »). Ces anomalies sont à l'origine de grandes pertes liées aux réexécutions sporadiques et pertes de temps associées. Menée en collaboration avec TELUS, la présente recherche soutient que les anomalies de pipelines peuvent être atténuées grâce à des automatisations intelligentes basées sur les données d'exécution des pipelines, notamment les journaux et métriques d'exécutions. Le premier travail se focalise sur la détection efficiente des échecs intermittents, étape fondamentale pour l'automatisation des processus de diagnostic et de réparation associés. À cette fin, nous proposons SLID, un outil de détection fondé sur l'ajustement fin de modèles de transformation de texte pré-entraînés, qui atteint des performances élevées à partir de seulement douze paires de journaux étiquetés manuellement, surpassant l'état de l'art qui repose sur de grands ensembles de données souvent mal étiquetées. Le second travail dresse le tout premier catalogue répertoriant 46 catégories d'échecs intermittents et démontre qu'une priorisation efficace de ces catégories exige de dépasser l'analyse traditionnelle fondée uniquement sur leur fréquence. Ainsi, nous introduisons FlakeRanker, un outil qui détermine les catégories à prioriser (14 chez TELUS) grâce à un clustering intégrant leur fréquence, leur récence et leur coût pour l'organisation. Le troisième travail développe FlaXifyer, un classifieur entraîné avec seulement douze journaux étiquetés par catégorie pour prédire automatiquement laquelle des 14 catégories prioritaires correspond à un échec intermittent. Ce travail introduit également LogSift, une technique d'interprétabilité qui met en évidence les segments de journaux les plus influents, accélérant ainsi le triage et le diagnostic. Le quatrième travail se penche sur le phénomène des échecs silencieux et met en lumière les facteurs clés liés aux réexécutions de pipelines qui rapportent à tort des succès, pouvant ainsi faciliter leur détection. Il présente également une taxonomie de ces échecs en 11 thématiques observées en contexte industriel et recense les solutions de contournement ou correctives disponibles. Enfin, le cinquième travail introduit le CBDT Framework, un cadre logiciel dont l'architecture s'inspire des jumeaux numériques et qui constitue le socle permettant d'orchestrer diverses solutions d'automatisation destinées à améliorer la fiabilité et la performance des pipelines de CI/CD. Ensemble, ces travaux tracent la voie vers une véritable automatisation intelligente, capable

d'orchestrer la détection, le diagnostic et la réparation autonome des anomalies de pipelines. Cette approche contribue non seulement à renforcer la fiabilité des pipelines et la qualité des logiciels livrés, mais aussi à alléger la charge cognitive pesant sur les développeurs. Elle favorise par ailleurs une utilisation plus efficiente des ressources computationnelles (avec des retombées positives sur le plan environnemental) tout en limitant les coûts associés aux retards de livraison, ainsi qu'aux défaillances logicielles pour les organisations.

Mots-clés: intégration continue, livraison continue, détection d'anomalie, diagnostic d'échec, analyse de trace d'exécution, traitement du langage naturel, classification, jumeaux numériques

Intelligent Detection and Diagnosis of Anomalies in Software Delivery Pipelines

Henri AÏDASSO

ABSTRACT

Modern organizations have widely adopted DevOps automation practices, enabling them to deliver software versions quickly and frequently while ensuring the quality of the code produced by developers. These practices are mainly enabled by continuous integration and continuous deployment (CI/CD) pipelines, whose execution results are used to certify software quality and confirm successful deployments in case of "success", and to report code issues to developers in case of "failure". As such, the value of CI/CD pipelines lies essentially in reliable and deterministic execution results. In practice, however, anomalies related to complex and distributed environments lead to unexpected and misleading pipeline results (i.e., false alarms known as "intermittent failures" or, worse, deceptive successes referred to as "silent failures"). These anomalies cause significant waste due to sporadic reruns and associated delays. Conducted in collaboration with TELUS, this research argues that pipeline anomalies can be mitigated through intelligent automation based on pipeline execution data, including execution logs and metrics. The first study proposes an efficient detection approach for intermittent failures based on fine-tuning pre-trained text embedding models. This approach achieves high performance (70-88% F1) using only twelve pairs of manually labeled logs and outperforms state-of-the-art approaches that rely on large, often inaccurately labeled datasets. The second study presents a first-of-its-kind diagnostic catalog of 46 categories of intermittent failures and demonstrates that effective prioritization of these categories requires looking beyond their raw frequency. Hence, we introduce FlakeRanker, a tool that determines the categories to be prioritized (14 at TELUS) using a clustering algorithm that integrates their frequency, recency and organizational cost. The third study introduces FlaXifyer, a classifier trained with only twelve labeled logs per category to automatically predict which of the 14 priority categories an intermittent job failure belongs to. This study also presents LogSift, an interpretability technique that highlights the most influential log segments, accelerating triage and diagnosis. The fourth study examines the phenomenon of silent failures and outlines the key factors associated with reruns of pipelines that incorrectly report successes, thereby facilitating their detection. It also presents a taxonomy of 11 themes observed in industrial contexts and summarizes the available workarounds and corrective solutions. Finally, the fifth study introduces the CBDT Framework, a digital twin-inspired architecture for orchestrating detection, diagnosis, and repair services in production CI/CD systems. Together, these efforts pave the way for intelligent automation capable of orchestrating the autonomous detection, diagnosis, and repair of pipeline anomalies. This approach not only improves pipeline reliability and the quality of delivered software, but also reduces developers' cognitive load and minimizes costs associated with delivery delays and software defects.

Keywords: continuous integration, continuous delivery, anomaly detection, failure diagnosis, log analysis, natural language processing, classification, digital twins

TABLE OF CONTENTS

Page

INTRODUCTION 1
0.1 Context 1
0.2 Research Problem 2
0.3 Research Objectives 4
0.4 Contributions 5
0.5 Thesis Outline 6

CHAPTER 1 BUILD OPTIMIZATION: A SYSTEMATIC LITERATURE REVIEW ... 7
1.1 Preamble 7
1.2 Introduction 7
1.3 Background 10
1.3.1 Continuous Integration 10
1.3.2 Machine Learning 12
1.4 Systematic Review Methodology 13
1.4.1 Research Questions 13
1.4.2 Initial Search 14
1.4.3 Screening: Inclusion & Exclusion Criteria 15
1.4.4 Snowballing Process 17
1.4.5 SLR Update 17
1.5 Literature Review Findings 18
1.5.1 Publication Trends 18
1.5.2 Build Research Topics 22
1.5.3 Techniques for Improving Build Quality 26
1.5.3.1 Identifying the Root Causes and Factors of Build Failures 27
1.5.3.2 Automating the Prevention and Repair of Build Failures 32
1.5.4 Build Reliability: Challenges and Current Approaches 34
1.5.4.1 Flaky Tests 34
1.5.4.2 Intermittent Job Failures 35
1.5.5 Build Data Sources 37
1.5.5.1 Build Metrics 38
1.5.5.2 Public Datasets 41
1.6 Discussion and Opportunities 43
1.7 Conclusion 45

CHAPTER 2 OBJECTIVES 47
2.1 Conceptual Framework 47
2.2 Research Objectives 48

CHAPTER 3 METHODOLOGY 51
3.1 Research Paradigm 51

3.2 Industrial Collaboration 51
3.2.1 CI/CD at TELUS 52
3.2.2 Practitioner Engagement 52
3.2.3 Constraints and Implications 53
3.3 Datasets 53
3.3.1 Industrial Data 54
3.3.2 Open-Source Data 55
3.3.3 Data Sources 56
3.4 Research Design 57
3.4.1 Few-Shot Learning 58
3.4.2 Open Coding 59
3.4.3 RFM Analysis 59
3.4.4 Sufficient Input Subset Interpretability 59
3.4.5 Mixed-Effects Logistic Regression 60
3.4.6 Digital Twin Architecture 60
3.5 Summary 61

CHAPTER 4 EFFICIENT DETECTION OF INTERMITTENT JOB FAILURES USING FEW-SHOT LEARNING 63
4.1 Preamble 63
4.2 Introduction 63
4.3 Background and Motivation 66
4.4 Few-Shot Learning Approach for Detecting Intermittent Job Failures 69
4.4.1 Log Pre-processing 69
4.4.2 Sentence Transformer Few-Shot Fine-tuning 70
4.4.3 Classification 71
4.5 Study Design 71
4.5.1 Research Questions 71
4.5.2 Studied Projects and Data Preparation 72
4.5.3 Validation of the FSL-based Approach 76
4.6 Results 80
4.6.1 RQ_1. What are the error rates of the automated labeling method based on non-deterministic job reruns 80
4.6.1.1 Motivation 80
4.6.1.2 Approach 80
4.6.1.3 Results 81
4.6.2 RQ_2. How effective is our few-shot learning approach for intermittent job failure detection 82
4.6.2.1 Motivation 82
4.6.2.2 Approach 82
4.6.2.3 Results 83
4.6.3 RQ_3. Can a few-shot model for intermittent job failure detection be used on another project 85

4.6.3.1 Motivation 85
4.6.3.2 Approach 86
4.6.3.3 Results 86
4.7 Discussion 87
4.7.1 SLID: A Tool for Creating Intermittent Job Failure Detectors 87
4.7.2 Practical Implications 87
4.7.3 Limitation and Future Directions 88
4.8 Threats to Validity 89
4.9 Conclusion 90

CHAPTER 5 ON THE DIAGNOSIS OF FLAKY JOB FAILURES: UNDERSTANDING AND PRIORITIZING FAILURE CATEGORIES .. 91
5.1 Preamble 91
5.2 Introduction 91
5.3 Background and Motivation 94
5.4 Study Design 95
5.4.1 Research Questions 95
5.4.2 Data Collection 96
5.4.3 Identifying Flaky Job Failure Categories and Automating Labeling 98
5.5 Results 101
5.5.1 RQ_1. What are the main categories of flaky job failures 101
5.5.1.1 Motivation 101
5.5.1.2 Approach 101
5.5.1.3 Results 101
5.5.2 RQ_2. Which failure categories are the most costly 105
5.5.2.1 Motivation 105
5.5.2.2 Approach 105
5.5.2.3 Results 107
5.5.3 RQ_3. How do the failure categories evolve over time 108
5.5.3.1 Motivation 108
5.5.3.2 Approach 109
5.5.3.3 Results 110
5.5.4 RQ_4. What are the priority flaky failure categories 111
5.5.4.1 Motivation 111
5.5.4.2 Approach 111
5.5.4.3 Results 113
5.6 Discussion 115
5.6.1 Insights from Priority Categories 115
5.6.2 FlakeRanker: A Tool for Automated Labeling and Prioritization 117
5.6.3 Practical Implications 119
5.7 Threats to Validity 120
5.8 Conclusion 121

CHAPTER 6 PREDICTING INTERMITTENT JOB FAILURE CATEGORIES FOR DIAGNOSIS USING FEW-SHOT FINE-TUNED LANGUAGE MODELS 123
6.1 Preamble 123
6.2 Introduction 123
6.3 Background and Motivation 126
6.4 FlaXifyer: Predicting and Interpreting Intermittent Job Failure Categories 128
6.4.1 Overview 128
6.4.2 Log Pre-processing 129
6.4.3 Few-Shot Fine-Tuning 129
6.4.4 Classification Head 130
6.4.5 Interpretability: Identifying Influential Log Statements 131
6.5 Study Design 133
6.5.1 Research Questions 133
6.5.2 Dataset 133
6.5.3 Experimental Setup 135
6.5.4 Evaluation Metrics 136
6.6 Results 137
6.6.1 RQ_1. How well does FlaXifyer predict intermittent job failure categories 137
6.6.1.1 Motivation 137
6.6.1.2 Approach 138
6.6.1.3 Results 139
6.6.2 RQ_2. How does model performance evolve as new failure categories are introduced 140
6.6.2.1 Motivation 140
6.6.2.2 Approach 140
6.6.2.3 Results 141
6.6.3 RQ_3. Can we determine the most influential log statements to support interpretability 143
6.6.3.1 Motivation 143
6.6.3.2 Approach 143
6.6.3.3 Results 144
6.7 Discussion 147
6.8 Threats to Validity 148
6.9 Conclusion 149

CHAPTER 7 ON THE ILLUSION OF SUCCESS: AN EMPIRICAL STUDY OF JOB RERUNS AND SILENT FAILURES IN INDUSTRIAL CI 151
7.1 Preamble 151
7.2 Introduction 151
7.3 Background and Motivation 154
7.4 Study Design 157

7.4.1 Research Questions 157
7.4.2 Data Collection 158
7.4.3 Identifying Silent Job Failures 160
7.4.4 Independent Explanatory Variables 161
7.4.5 One-Hot Encoding of Categorical Variables 163
7.4.6 Cyclical Encoding of Periodic Variables 164
7.5 Results and Discussion 165
7.5.1 RQ_1. How often do developers rerun successful jobs? What is the impact of rerunning successful jobs 165
7.5.1.1 Motivation 165
7.5.1.2 Approach 165
7.5.1.3 Results 166
7.5.2 RQ_2. What are the most important circumstances under which developers rerun successful jobs 170
7.5.2.1 Motivation 170
7.5.2.2 Approach 171
7.5.2.3 Results 174
7.5.3 RQ_3. What types of failures occur silently within jobs marked as successful 179
7.5.3.1 Motivation 179
7.5.3.2 Approach 179
7.5.3.3 Results 181
7.6 Discussion 186
7.7 Threats to Validity 188
7.8 Conclusion 192

CHAPTER 8 TOWARDS BUILD OPTIMIZATION USING DIGITAL TWINS 193
8.1 Preamble 193
8.2 Introduction 193
8.3 CI Build Digital Twin (CBDT) 195
8.3.1 Optimization Objectives 195
8.3.2 Interactions and Collaboration 197
8.4 CBDT Framework 198
8.4.1 Implementation Setup 198
8.4.2 Framework Components 199
8.5 Leveraging Machine Learning to Build CBDTs 201
8.6 Build Optimization with the CBDT 202
8.6.1 What-if Analysis Service 202
8.6.2 Improvement Services 202
8.7 Conclusion 203

CHAPTER 9 GENERAL DISCUSSION 205
9.1 Preamble 205
9.2 From Detection to Repair: A Suite of Tools for Improving Pipeline Reliability 205

9.3 Lessons Learned from Industrial Deployment 208
9.4 Generalizability to Other Software Organizations 209
9.5 Toward Fully Autonomous Resolution of Pipeline Reliability Issues 210

CONCLUSION 213

APPENDIX I INTERVIEW PROTOCOL 215

APPENDIX II CHARACTERISTICS OF STUDIED PROJECTS 217

APPENDIX III REPLICATION OF INTERMITTENT FAILURE DETECTION ON ADDITIONAL OPEN-SOURCE PROJECTS 221

APPENDIX IV SUPPLEMENTARY MATERIAL AND REPLICATION OF SILENT FAILURE ANALYSIS ON OPEN-SOURCE PROJECTS 225

BIBLIOGRAPHY 235

LIST OF TABLES

Page

Table 1.1 Techniques investigated in the literature to improve build quality 27

Table 1.2 List of CI metrics. Each metric is described with its name, its data type, a short description, and a list of study references where it has been used . 38

Table 1.3 List of project metrics 39

Table 1.4 List of developer metrics 40

Table 1.5 List of code metrics 40

Table 1.6 List of publicly available build datasets 41

Table 3.1 Research design by objective. 58

Table 4.1 Overview of the studied projects and collected data for the detection study 73

Table 4.2 Statistics and Error rates of automated labeling compared to manual labeling across the studied projects 81

Table 4.3 Performance of FSL models trained using $N = 12$ and 15 examples per class 84

Table 4.4 Performance of FSL models for cross-project predictions 86

Table 5.1 Overview of data collected for the diagnosis study 96

Table 5.2 Categories of flaky job failures at TELUS 102

Table 5.3 Flaky job failure categories with outliers RFM values 112

Table 5.4 K-Means clustering of flaky job failure categories based on RFM scores 113

Table 5.5 Identified 14 priority flaky job failure categories 114

Table 6.1 Selected priority failure categories ranked by RFM priority: top (rank 1-2), high (rank 3-8), medium (rank 9-13) 134

Table 6.2 Model performance across shot settings for two encoder families. Values are reported as mean_{std} over seeds (%) 139

Table 6.3 Per-class F1-score across category set sizes K 142

Table 6.4 Relevance scoring for LogSift evaluation 143

Table 6.5 LogSift quantitative evaluation results 144

Table 7.1 Overview of the collected data for studying silent failures 159

Table 7.2 Factors used as independent variables. “F." stands for Family 162

Table 7.3 Statistics of job reruns at TELUS 166

Table 7.4 Mixed-effects logistic regression model performance 174

Table 7.5 Results of our mixed-effects logistic model 175

Table 7.6 Themes and solutions identified for silent job failures 182

Table 7.7 Cross-tabulation of silent failure themes (RQ_3) and job types identified from issue descriptions 185

LIST OF FIGURES

Page

Figure 1.1 Systematic Literature Review Process 16

Figure 1.2 Number of publications on build optimization by year and type of venue 19

Figure 1.3 Publication frequency over the years and by the build data source used. The annotations on the histogram bars indicate the number of papers using the data source in that particular year 20

Figure 1.4 Word Cloud generated from the content of the 97 identified papers. The frequent words indicate four primary topics: build quality (failure, result, outcome, error, code, test, dependency, issue), build duration (time, cost, prediction, skip, selection), CI systems (system, environment), and build data (data, log, metric, dataset) 22

Figure 1.5 Classification of build-related topics illustrated in a Sunburst Diagram. The numbers within the parentheses in the classification nodes represent the count of existing studies on the topic 23

Figure 1.6 Example of a pipeline highlighting the rerun sequence of the *build image* job. After an initial failure, the *build image* job is rerun three times before passing, revealing the flakiness of the job failures in the rerun sequence 35

Figure 2.1 Conceptual framework for the automated detection, diagnosis, and repair of pipeline reliability issues 48

Figure 4.1 Illustration of the two-step model training following the log pre-processing (*LP*) stage, including (1) few-shot fine-tuning a sentence transformer *ST* model used to produce embeddings of the training dataset and (2) training a classification head *C* using embeddings generated by the fine-tuned *ST* 68

Figure 4.2 Classifying a job failure as intermittent or not using a few-shot fine-tuned sentence transformer *ST* for embeddings generation and a classifier *C* 68

Figure 4.3 Overview of the experiment for evaluating our FSL-based approach for intermittent job failure detection using a number N of shots, i.e., log examples per class. D is the manually labeled dataset of logs, and *LP* depicts the log pre-processing step. L_i, V_i, T_i, and TR_i, are the i^{th} learning, validation, test, and training set, respectively. ε_i is the model

performance estimation on T_i. *HP* represents the hyperparameter space, and hpp_j is the set of hyperparameters for the j^{th} loop of hyperparameter search. *ST* represents the sentence transformer that generates log embeddings, while *C* denotes the classifier 74

Figure 4.4 Performance of FSL models versus number of manually labeled examples for intermittent job failures detection within studied projects .. 84

Figure 5.1 Overview of the approach for building the automated flaky job failure labeling tool 98

Figure 5.2 Ranking of the 20 most costly flaky job failure categories 107

Figure 5.3 Evolution over time of the 20 most frequent flaky job failure categories, arranged from top to bottom. The final scatter plot at the bottom shows the evolution of flaky job failures with missing log data 109

Figure 6.1 Predicting intermittent job failure categories from job logs using a language model and classification head 128

Figure 6.2 LogSift reduces a 359-line log to 2 lines, directly identifying the missing environment variable (`IMAGE_NAME`) 145

Figure 6.3 LogSift output for a misclassified job (true: `external_file_invalid_format`). Despite the incorrect prediction, the highlighted segment reveals the actual cause: a zip file read failure (`curl: (26)`) during upload, enabling developers to override the prediction 145

Figure 6.4 Example of LogSift producing a non-relevant segment. Despite high reduction, the identified statements contain no failure indicators or host resolution information 146

Figure 6.5 LogSift relevance assessment results 146

Figure 7.1 Illustration of a typical CI pipeline of four jobs, highlighting the rerun sequence of the third job responsible for building and publishing the container image. This job is re-executed twice due to silent failures, resulting in three total executions of this same job without any code or CI configuration file change 152

Figure 7.2 Log excerpt of a silent cache upload failure in a successful job 155

Figure 7.3 Log excerpt of a silent HTTP request failure in a successful job 156

Figure 7.4 Overview of our approach for (1) data collection and (2) preparation, including data pre-processing and labeling, to (3) answer our research questions 158

Figure 7.5 Evolution of job statuses between rerun-initiating jobs (x-axis) and the immediate rerun jobs (y-axis) in the identified job rerun sequences . 167

Figure 7.6 Distribution of time delays between rerun-initiating jobs and job reruns, by status of the initiating jobs. Each bar is annotated with the number of rerun jobs in that interval, with the corresponding proportion in parentheses 168

Figure 7.7 Month of year vs. the probability of rerunning successful jobs 177

Figure 8.1 Concepts of Digital Shadow and Digital Twin 194

Figure 8.2 Architecture of the CI Build Digital Twin (CBDT) 196

Figure 9.1 Integrated pipeline for the automated detection, diagnosis, and repair of intermittent job failures, operating within the CBDT Framework. The pipeline orchestrates SLID model for detection, FlakeRanker for prioritization, FlaXifyer model for classification, and LogSift for interpretability, supporting faster diagnosis. Repair paths vary by category complexity: automated scripts for well-understood issues, team notification with relevant log segments for issues requiring human expertise, and AI-assisted diagnosis for categories with complex root causes 206

LIST OF ALGORITHMS

Page

Algorithm 4.1 Few-Shot Learning Evaluation Using Monte Carlo Cross-Validation (MCCV) 77

Algorithm 6.1 LogSift: Influential Log Statements Identification 132

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
AT	Actual Twin
AUC-PR	Area Under the Precision-Recall Curve
BPMN	Business Process Model and Notation
CI	Continuous Integration
CD	Continuous Deployment
DT	Digital Twin
FSL	Few-Shot Learning
HPO	Hyper-Parameter Optimization
IDE	Integrated Development Environment
IR	Information Retrieval
LLM	Large Language Model
MCCV	Monte Carlo Cross-Validation
ML	Machine Learning
MTEB	Massive Text Embeddings Benchmark
MVP	Minimum Viable Product
NLP	Natural Language Processing
ONNX	Open Neural Network Exchange

OSS	Open Source Software
ROC AUC	Receiver Operating Characteristic Area Under the Curve
RFM	Recency, Frequency, and Monetary Cost
RQ	Research Question
SDN	Software-Defined Network
SIS	Sufficient Input Subsets
SLR	Systematic Literature Review
SO	Stack Overflow
SOTA	State-Of-The-Art
SSE	Sum of Squared Errors
TF-IDF	Term Frequency - Inverse Document Frequency
VCS	Version Control System

INTRODUCTION

The automation of software delivery has transformed how organizations build and release products. Central to this transformation are Continuous Integration and Continuous Deployment (CI/CD) pipelines, automated workflows that build, test, and deploy software, enabling rapid, frequent and reliable version releases (Shahin, Ali Babar & Zhu, 2017; Hilton, Nelson, Tunnell, Marinov & Dig, 2017). Although CI/CD pipelines promise to free developers from manual, error-prone delivery processes (Hilton, Tunnell, Huang, Marinov & Dig, 2016), in practice, they increasingly require human intervention to firefight issues. Software teams lose countless hours to troubleshooting pipeline failures that often have nothing to do with their code changes, i.e., intermittent failures that vanish upon rerun, and silent failures that mask defects behind false successes (Durieux, Le Goues, Hilton & Abreu, 2020; Lampel, Just, Apel & Zeller, 2021; Olewicki, Nayrolles & Adams, 2022). At software companies such as Google, Mozilla, or TELUS, our industrial partner for this research project, these reliability issues carry high financial, operational, and human costs. This thesis argues that sustainable solutions require *intelligent* automation: machine learning systems that detect, diagnose, and ultimately repair pipeline anomalies while learning from limited examples and adapting to evolving environments.

0.1 Context

CI/CD pipelines, also referred to in the literature as *build* pipelines (Aïdasso, Sayagh & Bordeleau, 2025f), have become central to modern software delivery, and their adoption has been shown to differentiate high-performing teams from low-performing ones. Notably, Forsgren, Humble & Kim (2018) demonstrate that organizations excelling in continuous integration and continuous delivery practices deploy more frequently, recover faster from failures, and achieve better business outcomes. Their widespread adoption reflects the competitive pressure to deliver features rapidly while maintaining high standards of software quality.

However, the evolution of CI/CD processes comes with increasing complexity. As organizations update and scale their pipelines, they introduce heterogeneous tools, distributed execution environments, third-party service dependencies, and complex configuration hierarchies. At software-intensive organizations, this complexity translates to substantial operational costs. Build-related costs have been estimated at millions of dollars annually at companies like Google and Mozilla, spanning compute infrastructure, energy resources, and engineering effort (Hilton *et al.*, 2016; Memon *et al.*, 2017; Seo, Sadowski, Elbaum, Aftandilian & Bowdidge, 2014).

Similar dynamics are at play at TELUS, a leading Canadian telecommunications company and the industrial partner for this thesis. At TELUS, CI/CD pipelines support the development of Software-Defined Network (SDN) components powering services in critical domains such as city-scale telecommunications, internet infrastructure, television, smart agriculture, and telehealth. Typical pipeline jobs include compilation, testing (unit, integration, and system tests), static code analysis, container packaging, security scanning, and deployment to pre-production and production environments. As TELUS evolves its software delivery practices to support more frequent releases, the scale and heterogeneity of this environment, spanning self-hosted platforms, diverse toolchains, extensive security mechanisms, and interactions with physical network devices, make pipeline reliability an increasingly pressing concern.

0.2 Research Problem

As CI/CD pipelines grow in complexity (i.e., with heterogeneous and distributed tools), they become increasingly susceptible to reliability issues. These issues burden engineering teams, inflate operational costs, erode developer trust in the process, and ultimately undermine the efficiency gains that CI/CD is intended to deliver.

Specifically, two phenomena contribute to unreliable pipeline outcomes. The first is *intermittent job failures*, also known as flaky builds (Lampel *et al.*, 2021; Olewicki *et al.*, 2022; Durieux

et al., 2020). Such failures stem from irregular, non-code-related factors such as memory limitations, network instability, or other transient issues, which often lead to non-deterministic behavior, where jobs may fail or succeed upon rerun without any changes to the source code or CI configuration. Because developers typically interpret pipeline failures as signals of code defects, intermittent failures are misleading and prompt repeated job reruns in the hope of eventual success (Olewicki *et al.*, 2022), wasting compute resources and developer time.

The second phenomenon is *silent failures*, where jobs report success despite partially or entirely failing to perform their intended tasks. Silent failures are particularly dangerous because they allow defects to propagate into production undetected. Given that developers typically investigate only when a job fails, a false success escapes scrutiny. At TELUS, silent failures also trigger repeated reruns once suspected, placing additional strain on already overburdened resources and compounding the (financial and environmental) waste associated with these reruns.

The consequences of unreliable pipelines extend beyond infrastructure costs. Developers spend considerable effort diagnosing issues whose root causes often lie outside their responsibilities, expertise, or access level. When reruns do not resolve the problem, frustration mounts, and troubleshooting drags on across several teams. Collectively, these dynamics result in delayed software delivery, reduced productivity, and, in some cases, heightened stress and burnout within teams (Kim, Humble, Debois & Willis, 2016; Beyer, Jones, Petoff & Murphy, 2016).

Against this backdrop, the motivation for this work emerged from observations at TELUS, where interviews with engineers revealed a growing prevalence of intermittent job failures attributed to the specific complexity of the TELUS CI environment. However, this problem is not isolated to TELUS. Prior studies indicate that large organizations such as Mozilla (Lampel *et al.*, 2021), Amadeus (Moriconi, Troncy, Francillon & Zouaoui, 2022), or Ubisoft (Olewicki *et al.*, 2022) face similar reliability challenges and have initiated efforts to detect intermittent job failures. Yet, these contributions remain insufficient to address the problem sustainably. In particular,

detection alone does not address troubleshooting overhead, while systematic approaches to diagnosis and repair of these issues are lacking.

Moreover, the urgency and scale of the problem are underscored by recent industry developments. In May 2025, the development analytics team at GitLab launched an initiative to address pipeline instabilities, noting that "failures caused by flaky tests, infrastructure issues, and resource exhaustion persist with no systemic resolution, hinder development velocity, increase developer frustration, mask real product issues, and result in reactive firefighting".[1]

In summary, these observations highlight the need for systematic research and practical solutions to mitigate the inefficiencies caused by unreliable CI/CD pipelines.

0.3 Research Objectives

The primary objective of this thesis is twofold:

- To advance the understanding of pipeline reliability challenges in modern CI/CD systems
- To develop practical automation techniques that mitigate their associated inefficiencies

The specific objectives emerge from the systematic literature review (SLR) in Chapter 1 and are shaped by both our conceptual framework for tackling pipeline reliability issues (Chapter 2) and the industrial context of our collaboration with TELUS. The specific objectives are:

RO_1 Develop an efficient approach for detecting intermittent job failures.

RO_2 Develop a systematic approach for understanding and prioritizing intermittent failure categories.

RO_3 Develop an automated triage approach to accelerate intermittent failure diagnosis.

[1] https://gitlab.com/groups/gitlab-org/quality/analytics/-/epics/8

RO$_4$ Investigate the contributing factors and mitigation strategies for silent failures.

RO$_5$ Develop a framework enabling the practical deployment of CI-improving solutions.

0.4 Contributions

Each research objective is addressed by a corresponding contribution (C), summarized below:

C$_1$. **Efficient detection of intermittent job failures** (Chapter 4). We demonstrate that the SOTA automated labeling method (required for training detection models with large datasets) mislabels about 33% of intermittent job failures. Then, we propose a few-shot learning approach that achieves 70-88% F1-score using only 12 manually labeled examples per class, outperforming the SOTA with greater robustness.

C$_2$. **Systematic categorization and prioritization of intermittent job failures** (Chapter 5). We establish a taxonomy of 46 intermittent failure categories that highlights the diversity of these reliability issues. Our empirical analysis reveals that the most frequent failure categories are not necessarily the most costly or the most recent, underscoring the need for prioritization beyond frequency. We therefore introduce FlakeRanker, a novel approach that leverages Recency, Frequency, and Monetary cost (RFM) scoring and clustering to identify priority categories (14 at TELUS) for targeted diagnosis and resolution efforts.

C$_3$. **Automated triage with interpretable outputs for accelerated diagnosis** (Chapter 6). Building on the priority categories from C$_2$, we develop FlaXifyer, a few-shot multi-class classifier that achieves 84% F1-score, and LogSift, an interpretability technique that identifies the most influential log segments to enable faster diagnosis, reducing log review effort by 74% while surfacing relevant failure indicators in 87% of cases.

C$_4$. **Empirical analysis of silent job failures** (Chapter 7). We conduct an empirical study of silent job failures in CI/CD systems, identifying key factors associated with false

successes, including jobs involving artifacts (testing, static analysis, code and image build) and scripting-dominant projects (e.g, Shell as the primary language). We also establish a catalogue of 11 thematic categories, such as artifact operation errors, caching issues, and non-zero exit code, along with practical mitigation strategies for each.

C_5. Framework for orchestrating CI-improving solutions in production (Chapter 8). We introduce the CBDT FRAMEWORK, a digital twin-inspired architecture that supports continuous monitoring and the orchestration of detection, diagnosis, and repair services, enabling holistic improvement of CI/CD reliability in production environments.

0.5 Thesis Outline

Chapter 1 presents our SLR on build optimization that establishes the SOTA and serves as the unified related work for this thesis. To avoid redundancy, subsequent chapters do not include separate related work sections. Instead, they provide only the background and motivation specific to each study. Chapter 2 introduces the conceptual framework for addressing pipeline reliability issues, derived from the SLR findings, and presents the defined research objectives.

Chapter 3 describes the research methodology, including the industrial collaboration with TELUS, the datasets used, and an overview of the research methods and evaluation approaches.

Chapters 4 to 6 present contributions C_1–C_3, addressing the detection and diagnosis of intermittent failures. Chapter 7 presents contribution C_4, focusing on understanding silent failure patterns and mitigation strategies. Chapter 8 presents contribution C_5, introducing a framework for orchestrating the developed solutions in practice.

Finally, Chapter 9 reflects on the practical challenges of industrial deployment, discusses ongoing work toward automated repair using AI agents, and outlines future research directions toward fully autonomous self-healing pipelines.

CHAPTER 1

BUILD OPTIMIZATION: A SYSTEMATIC LITERATURE REVIEW

Henri Aïdasso[1] , Mohammed Sayagh[1] , Francis Bordeleau[1]

[1] Department of Software and IT Engineering, École de Technologie Supérieure,
1100 Notre-Dame Ouest, Montreal, Quebec, Canada H3C 1K3

Article published in « ACM Computing Surveys » August 2025

1.1 Preamble

This chapter presents our SLR on build optimization, published in ACM CSUR (Aïdasso *et al.*, 2025f). It establishes the SOTA and serves as the unified related work for this thesis. Three clarifications are warranted regarding its scope and relationship to subsequent chapters. First, the published SLR covers build optimization broadly, encompassing techniques and data for addressing long build duration, frequent build failures, and detecting flaky builds, also referred to as intermittent job failures. Given this thesis's focus on pipeline reliability, the chapters that follow draw primarily on the findings related to build failures and flaky builds. Techniques for build duration optimization, while presented in the published SLR, are less relevant to the thesis objectives and are therefore not elaborated upon here. Second, this chapter extends the published SLR with additional related work essential to the thesis. Specifically, Section 1.5.4 covers existing studies on flaky tests, a subset of intermittent job failures. Third, Section 1.6 synthesizes the key findings and research gaps identified in the SLR, and introduces a conceptual framework for the automated detection, diagnosis, and repair of intermittent job failures. This conceptual framework guided the contributions presented in subsequent chapters.

1.2 Introduction

Continuous Integration (CI) is a widely adopted software engineering practice, enabling developers to frequently and safely integrate their code changes into a shared repository (Martin Fowler, 2006; Hilton *et al.*, 2016; Shahin *et al.*, 2017). As a result of using CI, most software

organizations (including the most successful ones such as Google, Microsoft, or Netflix) deploy multiple software versions daily into production (Memon *et al.*, 2017; Esfahani *et al.*, 2016). Central to this practice are CI pipeline executions (often referred to as the "build" process), which automatically convert the source code into standalone software artifacts ready for end-users.

CI pipelines typically involve several steps, such as code compilation, static code analysis, testing, and packaging (Zhang, Chen, Hu, Peng & Zhao, 2022; Jin & Servant, 2022). Historically, developers performed these tasks manually, with each new build generally expected to contain additional features and fewer defects. The rise of DevOps practices' adoption has promoted full automation of the build process, which is triggered upon each code submission to the central version control system (VCS) (Shahin *et al.*, 2017). Each pipeline execution produces extensive logs that developers use to diagnose errors in the event of build failures.

The extensive use of CI (with hundreds of thousands of daily builds at Google (Memon *et al.*, 2017)) has raised several challenges, mainly related to the increasingly long build times and more frequent build failures (Hilton *et al.*, 2017). Notably, CI costs have been estimated at millions of dollars at Google and Mozilla (Zhang *et al.*, 2022). These costs include the increased infrastructure and energy costs of running builds, as well as the costs associated with reduced developer productivity due to frequent failures (Seo *et al.*, 2014; Memon *et al.*, 2017; Olewicki *et al.*, 2022). To address these challenges, researchers have undertaken several studies and developed techniques for improving CI, by leveraging the build data generated during the CI build process (Jin & Servant, 2021b).

In the aim of facilitating the usage and optimization of the CI-improving techniques, it is important to gain a good overview of the existing practices for build optimization. For example, Jin & Servant (2021b) surveyed the existing state-of-the-art (SOTA) build and test selection techniques. However, their study is scoped to build time reduction techniques and is limited to only 10 papers. To the extent of our knowledge, there is no systematic literature review (SLR) that summarizes the existing practices in the literature regarding build optimization.

Hence, in this paper, we conduct a thorough SLR on build optimization. We obtained a total of 97 papers published between 2006 and 2024. The goal of our SLR is to help practitioners identify the existing approaches, techniques, and practices relevant to their problems and help researchers identify gaps and opportunities for future research on CI pipeline optimization. To do so, we address the following research questions (RQs):

RQ_1. What is the trend of the number of studies on build? The majority of papers on build optimization were published after 2017, following the release of the popular TravisTorrent dataset. In fact, most studies in the field have used the TravisTorrent dataset. Overall, the number of publications has been steadily increasing in recent years, demonstrating a growing interest among researchers in this area.

RQ_2. What are the main build-related topics covered by the literature? Build optimization studies can be classified into several hierarchical categories grouped into four main topics: 48% of the studies focused on the build duration, 38% focused on challenges related to build failures, 8% focused on the general improvement of the CI systems including reliability challenges, and 6% on the data used for build optimization research.

RQ_3. What are the existing techniques for build quality improvement? A considerable number of papers focused on build failure analysis. They found that build failures are associated with code complexity (i.e., large code changes) and mainly result from issues in the code (e.g., compilation errors, test failures) and the CI environment. The findings of these studies have been used to develop several techniques based on Machine Learning (ML) algorithms for (1) predicting when the build requires maintenance following code changes and (2) automating failure repair.

RQ_4. Which metrics are used for build studies? In addition to execution logs, several metrics are used to carry out build studies. We classify these metrics into five categories: code metrics, which are the most commonly used, CI metrics, developer metrics, pull request metrics, and project metrics. These metrics are extracted from various sources such as the VCS and the CI system, mainly using APIs and static analysis tools. Besides, the calculation of several metrics

diverges significantly from one study to another and requires further elucidation to facilitate reuse.

RQ_5. Which datasets are publicly available for build studies? A total of 8 public build datasets have been published in the literature, which can be leveraged for build failure analysis, build log analysis, build selection techniques, build prediction, and the detection of flaky failures that are unrelated to code. Four of these datasets are specific to the Travis CI system, and four to the Java programming language, highlighting the need for new datasets in a wider range of languages and modern CI tools.

The remainder of this chapter is structured as follows: Section 1.3 presents the background, Section 1.4 describes the methodology, and Section 1.5 presents and discusses the findings. Finally, Section 1.7 concludes the chapter.

1.3 Background

The goal of this section is to define and discuss relevant terminologies used in the rest of this chapter and overall in this thesis.

1.3.1 Continuous Integration

Continuous Integration (CI) is the process that automates the execution of different steps required to build and release the software through a pipeline of *jobs* triggered at the submission of code changes (Jin & Servant, 2020). These *jobs* can vary from one development context to another. In general, they include static code analysis, compilation, unit tests, and various tasks that can result in the automated deployment of a new version into production, referred to as the Continuous Deployment (CD) practice. Thus, the CI/CD pipeline (or simply CI pipeline) ensures the quality of integrated changes and the fast, reliable creation of new software versions.

For consistency with previous works, we employ the terms *build* and *CI* to refer to the CI pipeline in this SLR. Indeed, the difference between these terms in the literature remains unclear

from one study to another, and they are used interchangeably to refer to the CI pipeline (Saidani, Ouni, Chouchen & Mkaouer, 2021a; Lampel *et al.*, 2021; Gallaba, Macho, Pinzger & McIntosh, 2018; Xia & Li, 2017; Saidani, Ouni, Chouchen & Mkaouer, 2020; Esfahani *et al.*, 2016). The term *build* also aligns with the terminology used by popular CI services like Travis CI[1] and Jenkins[2] when referring to the CI pipeline. It is worth noting that GitHub Actions[3] uses the term *workflow* instead, because workflows can also be used for tasks unrelated to the CI and CD practices (Golzadeh, Decan & Mens, 2022).

In practice, the pipeline is defined in a CI configuration file called the *build script* (such as the `.travis.yml` file), which defines the steps of the pipeline, how these steps are organized, as well as tools and commands that need to be run for each step. Developers are encouraged to frequently trigger the build by committing small code changes to the VCS (Memon *et al.*, 2017; Elazhary *et al.*, 2022). When the build execution is completed (i.e., fully executed as opposed to canceled), it produces one of two possible *results* (or *outcomes*): **success** or **failure**. On the one hand, the build is successful when all its jobs are executed successfully. Such build is called a *passing build* (Jin, 2021), and a green color indicates success on each of its jobs. On the other hand, a build failure occurs if any of the build jobs fail. A red color indicates the failed job, and the subsequent jobs are not executed as a result of the interruption of the CI pipeline. Developers use failed job execution logs to diagnose the cause of failures so as to quickly implement fixes toward a successful run of the CI pipeline. In addition, researchers leverage build data (logs and metrics) to understand the factors that influence build outcomes and performance, ultimately helping to improve CI systems.

Different dedicated tools known as *CI Systems* are used for setting up and running the build jobs as specified in the *build script*. They include Jenkins, Travis CI, CircleCI, GitLab CI, and GitHub Actions (GHA). GitLab CI and GHA are among the most popular today, both in open source and in industry (Widder, Hilton, Kastner & Vasilescu, 2018; Golzadeh *et al.*, 2022).

[1] https://www.jenkins.io/doc/book/pipeline

[2] https://docs.travis-ci.com/user/customizing-the-build

[3] https://docs.github.com/en/actions/writing-workflows/quickstart

1.3.2 Machine Learning

Machine Learning (ML) is a subfield of Artificial Intelligence (AI) that leverages mathematical and statistical algorithms to enable systems to learn patterns from data and perform specific tasks (e.g., classification, regression, clustering, recommendation) without being explicitly programmed (Mitchell, 1997; Bishop, 2006). Owing to their ability to generalize beyond previously observed cases, ML techniques have been widely adopted in software engineering research, including software vulnerability detection, defect or build outcome prediction, failure analysis, and automated repair (Sharma *et al.*, 2024).

In software engineering contexts, ML models are often supported by other AI subfields, especially Natural Language Processing (NLP), which are used to transform raw and unstructured textual data, such as source code, build logs, commit messages, or issue descriptions, into structured and informative features suitable for learning (Allamanis, Barr, Devanbu & Sutton, 2018). These features may include textual representations such as TF-IDF (Ramos, 2003), or more recently, numerical embeddings (Reimers & Gurevych, 2019) that better capture textual semantics.

Traditionally, the predictive performance of an ML model largely depended on both the quantity and quality of the data used during training. High-performing models typically required sufficiently large, representative, and well-curated datasets to effectively capture underlying patterns (Domingos, 2012). However, recent advances in model training have introduced few-shot learning (FSL) techniques (Ravichandran, Bhotika & Soatto, 2019), which substantially reduce the reliance on large labeled datasets that are often costly to obtain or inherently scarce in certain domains (e.g., anomalies). Instead, FSL enables models to learn effectively from only a small number of labeled examples, referred to as *shots*. The effectiveness of such approaches has enabled the advent of Large Language Models (LLMs) (Brown *et al.*, 2020).

LLMs, often fine-tuned for specific downstream tasks, have been widely studied and used across a broad range of software engineering applications (Zhang *et al.*, 2024). Their unprecedented performance has driven both industrial adoption and academic interest, fostering the emergence of so-called *AI agents* that rely on LLMs as a backbone for reasoning and information synthesis.

However, the operational costs and practical challenges associated with training, deploying, and using LLMs have shifted attention toward more efficient alternatives, such as Small Language Models (Belcak *et al.*, 2025) and few-shot fine-tuning techniques (Tunstall *et al.*, 2022).

The choice of an appropriate ML algorithm and learning technique is closely tied to the characteristics of the available data and the nature of the target task. In practice, the development and maintenance of ML-based systems follow a lifecycle that includes data collection, cleaning, preprocessing, then model training, evaluation, deployment, and periodic retraining to address concept drift and evolving data distributions (Angluin, 1988; Turhan, 2012).

1.4 Systematic Review Methodology

To conduct this SLR, we follow the approach of (Kitchenham & Charters, 2007), which is summarized in Figure 1.1. We further discuss the main stages of our methodology in the following subsections.

1.4.1 Research Questions

Since the goal of our SLR is to identify the covered topics by the literature (RQ_1-RQ_2), the existing solutions (RQ_3-RQ_4), and which datasets are available for research on the area of CI build (RQ_5), we design the following RQs:

RQ_1. What is the trend of the number of studies on build?

RQ_2. What are the main build-related topics covered by the literature?

According to the literature, build failures and long build times are the two main challenges developers face when using CI systems (Hilton *et al.*, 2017). Although the original paper defines two RQs covering both challenges, we present only this RQ here for the sake of conciseness:

RQ_3. What are the existing techniques for build quality improvement?

ML solutions relying on build data are the most widely used techniques in the literature for improving CI (Brandt, Panichella, Zaidman & Beller, 2020; Jin & Servant, 2021b). Hence, we design the next RQs to identify metrics and datasets available for build optimization research.

RQ$_4$. Which metrics are used for build studies?

RQ$_5$. Which datasets are publicly available for build studies?

1.4.2 Initial Search

We use string-based search and snowballing techniques to identify the relevant studies in the literature. To formulate the string-based search query, we define search terms that represent, according to Kitchenham & Charters (2007), the population: `Build`, `Continuous Integration`, and the intervention including popular CI tool names: `Jenkins, Travis CI, Gitlab CI, GitHub Actions`. In addition, we add to the intervention keywords the `Build Failure` mechanism, which is designed to provide feedback on CI errors, but also represents one of the major challenges of CI (Hilton *et al.*, 2017). Finally, we complete our search terms with some abbreviations (e.g., `CI` for `Continuous Integration`) and synonyms (e.g., `Breakage` for `Failure`), to increase the search results.

Furthermore, we restricted our query results to papers published in a venue whose name contains "`Software Engineering`", as appearing in the names of international and high-quality software engineering research venues (e.g., ICSE, FSE, and TSE). This is to reduce the noise in our primary results due to a large number of papers (3000+) related to *build* or *construction* in other engineering areas. Obtaining papers published in high-quality venues is also a way for us to ensure the quality and rigor of our analysis. Our finalized **search query** is as follows.

```
(((Build AND (CI OR Continuous Integration) OR ((Jenkins
OR Travis OR GitLab CI OR GitHub Actions) OR (Failure OR
Break OR Broke))) WN AB) AND ((Software Engineering) WN ST))
```

As recommended by Kitchenham & Charters (2007), we use our search query (initially on the date of Feb. 27, 2023) on six different databases, namely IEEExplore, ACM Digital Library, Science Direct, Springer, and Inspec and Compendex provided by the digital meta-library Engineering Village[4] (El-Masri, Petrillo, Guéhéneuc, Hamou-Lhadj & Bouziane, 2020). This platform offers access to multiple engineering-focused databases (El Compendex, Inspec) that cover papers from major sources and societies such as the IEEE library, ASME Digital Collection, Springer, and ACM library (Elsevier, 2022). We include the ACM, Springer, and IEEE databases to ensure that we do not miss any relevant papers, even if the Engineering Village indexes these databases. In addition, it is worth noting that Engineering Village provides several features, including *autostemming*, which allows searching for a root word and its other variations. For example, a search for the word `Break` will retrieve results also including `Breakage`.

Besides, we opt not to limit the search results to a specific time frame, as relevant studies on build optimization may have been published at any point. This allows us to capture publication trends in the field of build optimization. Finally, our query resulted in a total of 634 papers. For each paper found, we read the titles and abstracts, then applied the inclusion and exclusion criteria presented in section 1.4.3. **We identify at the end of the first paper screening step a total of 40 seed papers.**

1.4.3 Screening: Inclusion & Exclusion Criteria

Based on the abstract and title of each paper, we decide to include or exclude it using the following criteria:

Inclusion criteria

1. The paper must be written in English
2. The paper must be a conference or journal paper
3. The paper must be a software engineering article
4. The paper must focus on the CI build process

[4] https://www.engineeringvillage.com/

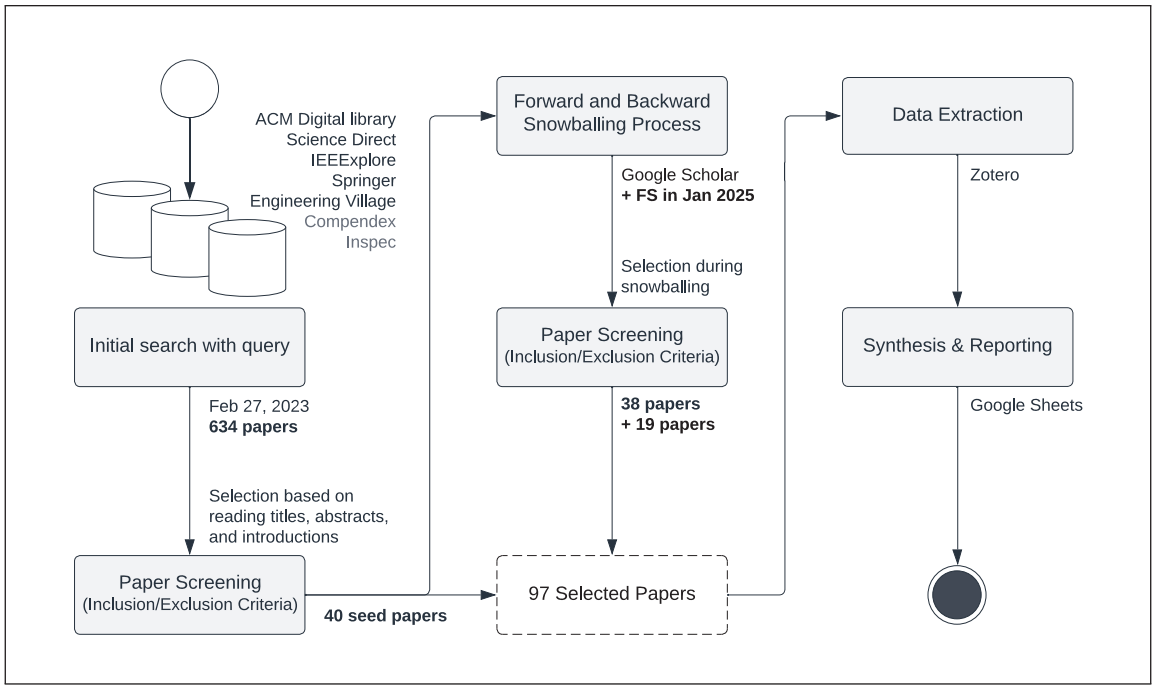


Figure 1.1 Systematic Literature Review Process

5. The study should address at least one way of improving the CI build or present findings for this purpose

Exclusion criteria

1. Duplicated papers
2. The paper is not publicly available
3. The study is not directly about the CI build, such as studies focusing on testing activities, code compilation process, or just-in-time defect prediction.
4. The study is related to deployed software (i.e., how to improve software in production)
5. The study is related to the build but does not focus on its optimization (e.g., introduction of DevOps techniques in software engineering courses (Hills, 2020) or general analyses of CI practices)

1.4.4 Snowballing Process

After identifying the seed papers, we perform the backward and forward snowballing process using Google Scholar[5] to capture papers that were not identified using our query. Backward snowballing consists of screening the list of references in each included paper, while forward snowballing consists of screening the list of papers that cited each included paper. As a result of the snowballing process, we find 38 additional papers (25 from backward snowballing and 13 from forward snowballing).

Out of the 38 papers, 31 were not identified in our query results due to the criteria on the software engineering venue's name that we have added. Indeed, these 31 papers have been published in other major venues whose names do not contain `Software Engineering`, such as the International Conference on Mining Software Repositories (MSR), the International Conference on Software Analysis, Evolution and Re-engineering (SANER), and the International Conference on Software Maintenance and Evolution (ICSME).

During the snowballing process, we encountered some special cases that led us to extend our exclusion criteria. Below are the additional exclusion criteria defined:

6. Website articles or blogs (e.g., Wikipedia)
7. Retracted papers and broken links
8. Books, Doctoral dissertations, Master's theses

We identify 78 relevant papers as of February 2023. After obtaining the final list of papers, we manually examine each of them to answer our research questions.

1.4.5 SLR Update

In January 2025, we updated our SLR to cover papers published in 2023 and 2024. To do so, we used Forward Snowballing (FS) on Google Scholar as recommended by Felizardo, Mendes, Kalinowski, Souza & Vijaykumar (2016) and Wohlin, Mendes, Felizardo & Kalinowski (2020)

[5] https://scholar.google.com/

for the efficient update of SLRs in software engineering. This forward snowballing is based on the 78 papers identified in February 2023, and the candidate papers are screened using the same inclusion and exclusion criteria. As a result, we find 19 additional relevant studies that we manually examine to include in our initial RQ results.

Finally, we obtain a final pool of 97 relevant papers as of January 2025, which we synthesize in this paper.

1.5 Literature Review Findings

As discussed earlier, the goal of our paper is to understand the state-of-the-art in build optimization. In particular, we address the following research questions:

- **RQ$_1$**. What is the trend of the number of studies on build?
- **RQ$_2$**. What are the main build-related topics covered by the literature?
- **RQ$_3$**. What are the existing techniques for build quality improvement?
- **RQ$_4$**. Which metrics are used for build studies?
- **RQ$_5$**. Which datasets are publicly available for build studies?

1.5.1 Publication Trends

The majority (88.7%) of build optimization studies have been published from 2017 onwards, and the publication rate appears to be stable in the past few years. Figure 1.2 illustrates the yearly publication trends in build optimization research, categorized by type of venue. It shows that build optimization was a very little discussed topic in the literature before 2017, with the number of publications ranging from 0 (in 2007, 2008, 2010, 2012, and 2013) to 4 (in 2014). Out of the 97 studied build papers, only 11 were published before 2017, representing 11.3% of the papers. Also, no papers on build optimization were identified before 2006, indicating that this topic has gained research interest predominantly within the last decade. Starting in 2017, the number of publications increased significantly, peaking at 15 papers published that particular year, with almost all (14 out of 15) published in conference venues. Overall, 86 papers have

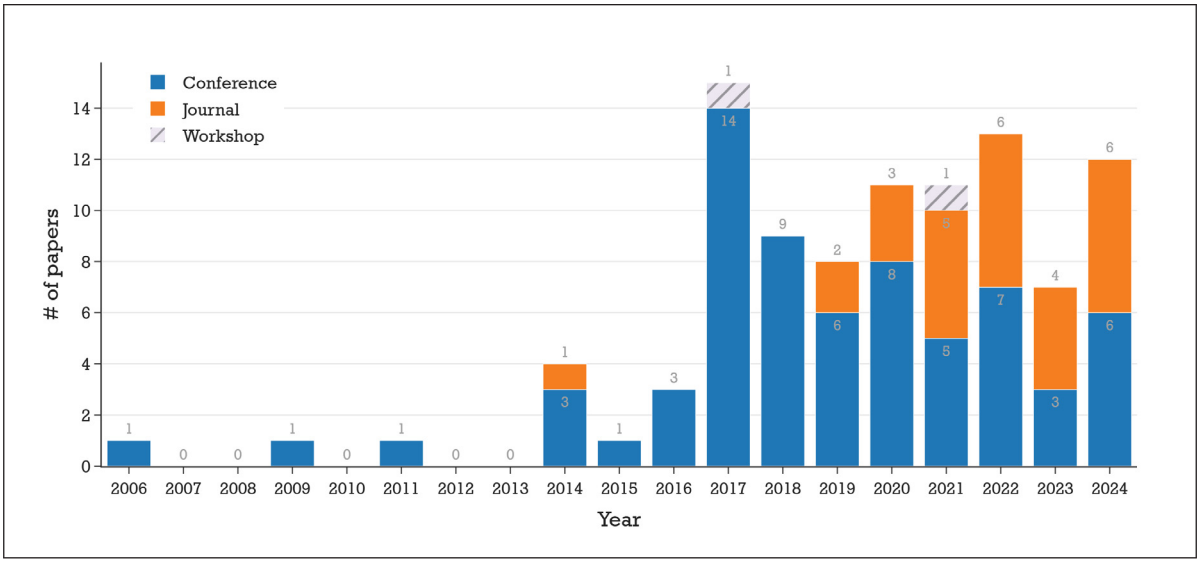


Figure 1.2 Number of publications on build optimization by year and type of venue

been published from 2017 to 2024, accounting for 88.7% of the identified papers. The growing interest in the build topics, which has significantly increased from 2017, is also reflected in consistent annual publications since then. Interestingly, the lowest number of papers published in a single year since 2017 is observed only recently in 2023, with 7 papers. Over the past five years (2020 to 2024), the number of papers published each year has been 11, 11, 13, 7, and 12, respectively, collectively representing over half (55.7%) of the 97 identified papers. This shows an overall stable publication rate of build studies over the past five years. Before 2019, only a single journal publication was identified. However, from 2019 to 2024, publications are nearly evenly distributed between conference and journal venues, reflecting the growing maturity of research in build optimization. Finally, this SLR was conducted in early 2025, and no build-related papers published in 2025 had been identified. However, we expect the number of publications to evolve in line with the trends observed in recent years.

Existing studies on build optimization are mostly motivated by the open availability of a wealth of historical build data, especially the TravisTorrent dataset, which appears to have spearheaded research in the field. Out of the 97 studied papers, 92 (≈ 95.8%) relied on historical build data for their studies. Figure 1.3 presents the distribution over time of build data

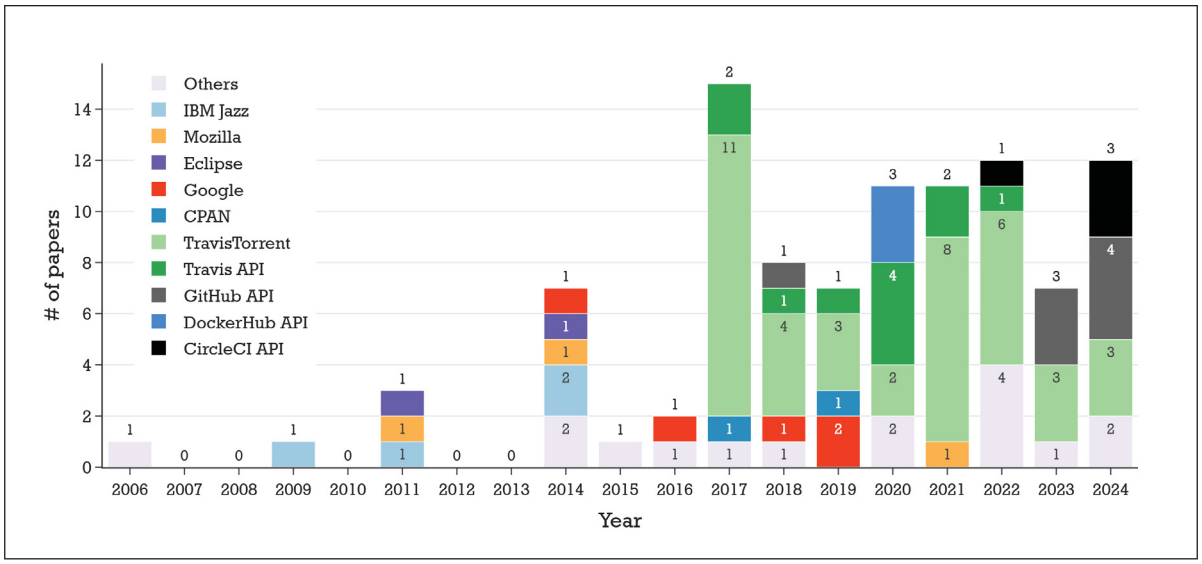


Figure 1.3 Publication frequency over the years and by the build data source used. The annotations on the histogram bars indicate the number of papers using the data source in that particular year

sources used in these 92 papers. We can observe that the use of build data over time follows approximately the same trend as the number of publications shown in Figure 1.2. Additionally, Figure 1.3 reveals that the peak of the number of papers in 2017 is related to the publication in that very year, of TravisTorrent, a large open-source dataset of 2.6 million builds introduced by Beller, Gousios & Zaidman (2017b). In fact, 11 of the 15 papers (73.3%) published in 2017 used the TravisTorrent dataset (Beller, Gousios & Zaidman, 2017a; Beller *et al.*, 2017b; Hassan & Wang, 2017; Ni & Li, 2017; Vassallo *et al.*, 2017; Bisong, Tran & Baysal, 2017; Xia & Li, 2017; Luo, Zhao, Ma & Chen, 2017; Islam & Zibran, 2017; Xia, Li & Wang, 2017; Dimitropoulos, Aung & Svetinovic, 2017). TravisTorrent has also been significantly used in the following years, from 2018 to 2024, accounting respectively for 50%, 42.8%, 18.2%, 72.7%, 46.2%, 42.8%, and 25% of the build data sources used. Besides, out of the 97 papers, a substantial number of papers have also used build data collected from Travis API (11 papers (Saidani, Ouni & Mkaouer, 2021b; Chen, Chen, Zhang & Peng, 2020; Zhang, Chen, Chen, Peng & Zhao, 2019; Pinto, Reboucas & Castor, 2017; Vassallo, Proksch, Zemp & Gall, 2018, 2020a; Rausch, Hummer, Leitner & Schulte, 2017; Barrak, Eghan, Adams & Khomh, 2021; Zhang *et al.*,

2022; Saidani *et al.*, 2020; Brandt *et al.*, 2020)), IBM Jazz (4 papers (Finlay, Pears & Connor, 2014; Wolf, Schroter, Damian & Nguyen, 2009; Mcintosh, Adams, Nagappan & Hassan, 2014; McIntosh, Adams, Nguyen, Kamei & Hassan, 2011)), and at large organizations including 5 papers at Google (Mesbah, Rice, Johnston, Glorioso & Aftandilian, 2019; Seo *et al.*, 2014; Liang, Elbaum & Rothermel, 2018; Esfahani *et al.*, 2016; Tufano, Sajnani & Herzig, 2019)) and 3 papers at Mozilla (Lampel *et al.*, 2021; Mcintosh *et al.*, 2014; McIntosh *et al.*, 2011)).

Over half (53.6%) of the build optimization studies have focused on projects using Travis CI as their CI system, although 2024 has seen a noticeable shift towards GitHub Actions and CircleCI. Figure 1.3 shows the distribution of the build data sources used in the literature for build optimization studies. It highlights the number of occurrences of each data source for each year. From 2017 to 2024, respectively 86.7%, 62.5%, 57.1%, 54.5%, 90.9%, 58.3%, 42.9%, and 25% of the build data sources (per year) are related to the Travis CI system (Travis API and TravisTorrent in shades of green). Overall, 52 papers (53.6% of the 97 identified papers) used build data collected from projects using Travis CI. This includes 40 papers that specifically employed the TravisTorrent dataset, representing 41.2% of the 97 identified papers.

The other CI systems covered in the literature are Google's CloudBuild (2 papers (Esfahani *et al.*, 2016; Tufano *et al.*, 2019)), Microsoft's CloudBuild (1 paper (Lebeuf, Voyloshnikova, Herzig & Storey, 2018)), Circle CI (5 papers (Gallaba, Lamothe & McIntosh, 2022b; Gallaba, Ewart, Junqueira & McIntosh, 2022a; Yin *et al.*, 2024; Sun, Habchi & McIntosh, 2024; Weeraddana, Alfadel & McIntosh, 2024a; Kola-Olawuyi, Weeraddana & Nagappan, 2024)), Mozilla's TaskCluster (1 paper Lampel *et al.* (2021)), Bitbucket Pipelines (2 papers (Hong *et al.*, 2024; Kawalerowicz & Madeyski, 2023)), GitLab CI (1 paper (Olewicki *et al.*, 2022)), and GitHub Actions (4 papers (Weeraddana, Alfadel & McIntosh, 2024b; Baral *et al.*, 2023; Wróbel, Szymukowicz & Weichbroth, 2023; Khatami, Willekens & Zaidman, 2024)).

Studies on GitHub Actions (with build data collected from the GitHub API) and CircleCI have mainly emerged over the past two years. In 2024, these studies account for more than half (58.3%) of the published papers, reflecting a rising interest in these CI systems. These findings

suggest that there is potential for further research on newer standard CI systems like GitHub Actions and GitLab CI, which are increasingly adopted in both industry and the open-source community (Golzadeh *et al.*, 2022).

Figure 1.4 Word Cloud generated from the content of the 97 identified papers. The frequent words indicate four primary topics: build quality (failure, result, outcome, error, code, test, dependency, issue), build duration (time, cost, prediction, skip, selection), CI systems (system, environment), and build data (data, log, metric, dataset)

1.5.2 Build Research Topics

The most frequently discussed topics are related to the build duration, then the build quality, as shown in Figure 1.5. The figure presents the classification of build optimization studies according to the four identified primary topics. Each primary topic includes multiple sub-topics organized hierarchically. Our technical review of build optimization techniques follows this proposed classification. Subsequent bold statements provide detailed discussions on each primary topic, offering critical information used to address the additional research questions outlined in Section 1.4.1.

On the topic of build duration, the literature focuses on reducing the cost of running long builds, and investigates techniques for obtaining build results more quickly to minimize waiting times. The most important topic of CI build addressed in the literature is the build

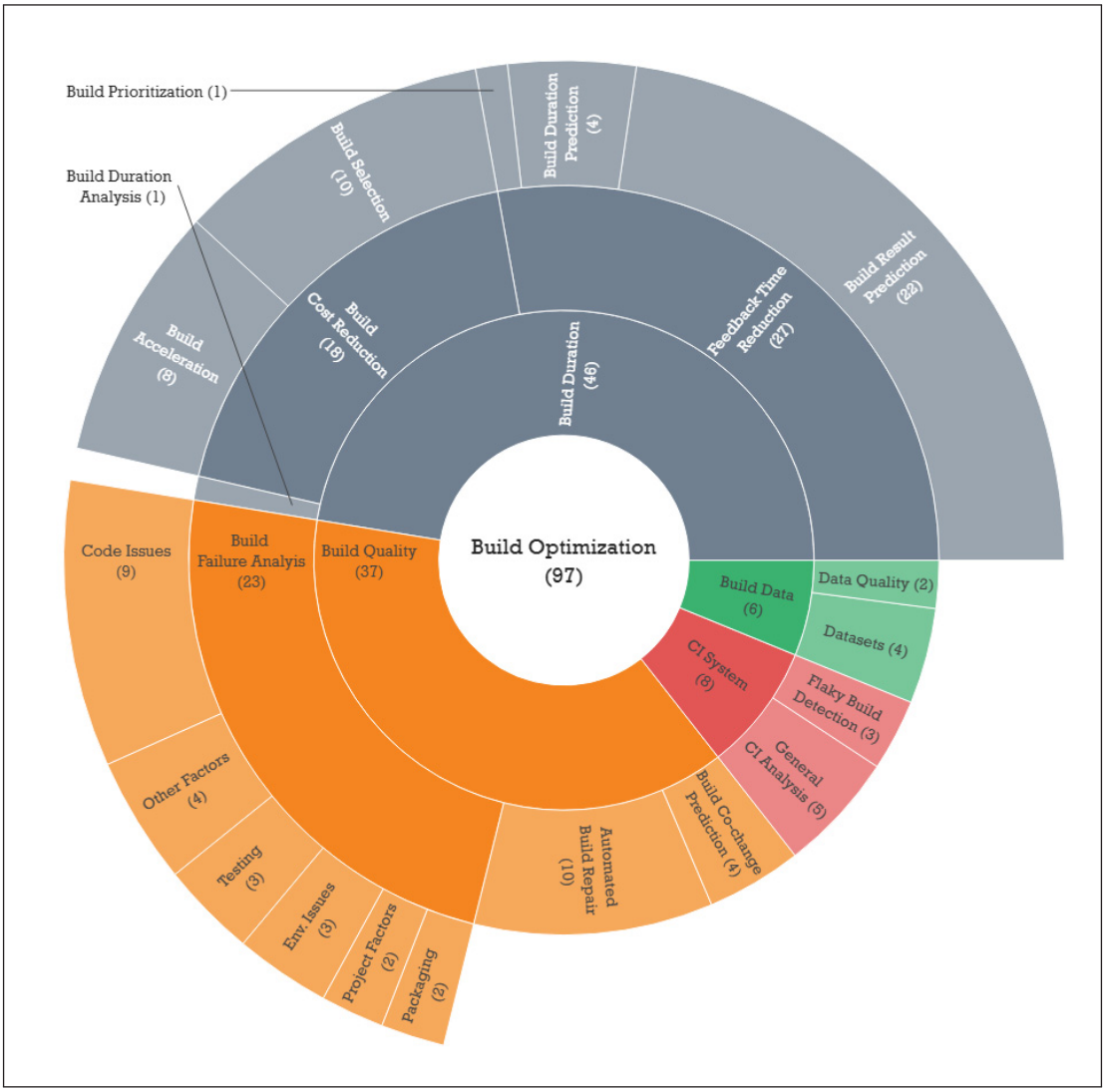


Figure 1.5 Classification of build-related topics illustrated in a Sunburst Diagram. The numbers within the parentheses in the classification nodes represent the count of existing studies on the topic

duration, accounting for 47.4% of the studies. As shown in Figure 1.5, two main goals have been identified in this research direction. The first goal (**build cost reduction**) is to reduce the high costs of frequently running long builds (estimated in millions of dollars at Google and Microsoft (Zhang *et al.*, 2022)). To do so, prior studies (Jin & Servant, 2022; Jin, 2021; Jin & Servant, 2020; Saidani *et al.*, 2021b; Abdalkareem, Mujahid, Shihab & Rilling, 2021b; Abdalkareem, Mujahid & Shihab, 2021a; Beheshtian, Bavand & Rigby, 2022; Weeraddana *et al.*,

2024b; Jin & Servant, 2023; Kamath, Fernandes, Adams & Hassan, 2024) investigated **build selection** to help practitioners identify the right builds to execute and skip the others to save costs. The right builds are the ones most likely to fail and reveal issues in the software version. Other studies (Esfahani *et al.*, 2016; Celik, Knaust, Milicevic & Gligoric, 2016; Zhang *et al.*, 2022; Gallaba *et al.*, 2022a; Randrianaina, Khelladi, Zendra & Acher, 2022; Yin *et al.*, 2024; Khatami *et al.*, 2024; Baral *et al.*, 2023) focused on **build acceleration** by reusing cached data from previous executions or repairing CI performance smells.

The second goal of existing research on build duration is to minimize the feedback waiting time (**build feedback time reduction**) for increased productivity. To do so, prior studies[6], for instance Chen *et al.* (2020); Santolucito, Zhang, Zhai, Cito & Piskac (2022); Wu *et al.* (2020a); Saidani, Ouni & Mkaouer (2022); Kawalerowicz & Madeyski (2023); Wang *et al.* (2024a); Sun *et al.* (2024), developed techniques for build result prediction before its execution, so that depending on the predicted result, practitioners can either decide to skip the execution or investigate their code for potential errors. Another line of work (Cao, Wen & McIntosh, 2017; Bisong *et al.*, 2017; Tufano *et al.*, 2019; Ghaleb, da Costa & Zou, 2019) explored **build duration prediction** to predict the total build time, enabling practitioners to better manage context switching from one task to another. Finally, Liang *et al.* (2018) investigated **build prioritization**, which consists of prioritizing the execution of builds prone to failure, so that practitioners could receive early feedback from the CI. Another study (Weeraddana *et al.*, 2024a) conducted a **build duration analysis** of timeout builds to understand the associated factors. RQ_3 discusses in more detail the techniques used for optimizing build duration.

On the topic of build quality, existing studies have sought to understand and address build failures in order to improve software quality. The second most discussed topic in the literature is the **build quality**, addressed in 37 papers accounting for 38.1% of the studies. On the one hand, a substantial number of studies (23 papers) focused on **build failure analysis** to understand the circumstances under which builds fail. In this sense, prior studies investigated the root cause of pipeline failures including issues related to the **code** (Barrak *et al.*, 2021; Seo *et al.*,

[6] A comprehensive list of build result prediction studies is provided in RQ_3

2014; Zhang *et al.*, 2019; Maes-Bermejo, Gallego, Gortázar, Robles & Gonzalez-Barahona, 2022; Rabbani, Harvey, Saquif, Gallaba & McIntosh, 2018; Luo *et al.*, 2017; Islam & Zibran, 2017; Kerzazi, Khomh & Adams, 2014; Dimitropoulos *et al.*, 2017), **testing** (Rausch *et al.*, 2017; Beller *et al.*, 2017a; Vassallo *et al.*, 2017), **packaging** (Wu, Zhang, Wang & Wang, 2020b; Henkel, Bird, Lahiri & Reps, 2020), **CI environment** (Ghaleb, da Costa, Zou & Hassan, 2021; Zolfagharinia, Adams & Guéhénuc, 2017; Zolfagharinia, Adams & Gueheneuc, 2019), factors related to the **project** (Benjamin, Mathew & Jose, 2023; Dimitropoulos *et al.*, 2017), and **other factors** such as developers and interplay with duration (Ghaleb, Hassan & Zou, 2022; Kola-Olawuyi *et al.*, 2024; Shimmi & Rahimi, 2024; Hong *et al.*, 2024). On the other hand, other studies have developed several techniques to improve the quality of the build by mitigating build failures. For this purpose, a line of work (McIntosh *et al.*, 2011; Macho, McIntosh & Pinzger, 2016; Mcintosh *et al.*, 2014; Xia, Lo, McIntosh, Shihab & Hassan, 2015) investigated **build co-change prediction** to automatically detect when the build script requires maintenance following code changes, assisting practitioners in the build maintenance. Other studies (Macho, McIntosh & Pinzger, 2018; Vassallo *et al.*, 2018; Hassan & Wang, 2018; Lou, Chen, Zhang, Hao & Zhang, 2019; Mesbah *et al.*, 2019; Hassan, 2019; Vassallo *et al.*, 2020a; Lou, Chen, Cao, Hao & Zhang, 2020; Hassan, Meng & Wang, 2023; Lee, Li & Hsu, 2024) have proposed techniques for **automated build repair** that identify root causes and fix code issues (e.g., syntax errors, dependency issues). RQ_4 discusses in more detail the techniques proposed to address build failures.

Other studies (8.2%) in the literature have focused on the topic of CI systems, to improve the reliability and overall performance of build executions. The third primary topic addressed in the literature is **CI System** and includes 8 (8.2%) of the identified studies. This topic encompasses, on one hand, general **CI analysis** of issues related to the usage of CI platforms used to run the builds. For instance, Pinto *et al.* (2017) highlighted that time pressure and inadequate testing are two main factors of build failures in organizations, where CI gives a false sense of confidence. Then, further studies (Vassallo, 2019; Gallaba *et al.*, 2022b) investigated the problems in the CI system that cause irregular build failures (i.e., flaky builds) and found a wide

variety of reasons, including user cancellation, infrastructure availability problems, configuration errors, and timeouts. Other studies (Lebeuf *et al.*, 2018; Jin & Servant, 2021b) have evaluated existing techniques for optimizing CI regarding various aspects such as execution costs, failure observation, and feedback time. Besides, some studies (Lampel *et al.*, 2021; Olewicki *et al.*, 2022; Moriconi *et al.*, 2022) explored **flaky build detection** approaches for identifying and diagnosing irregular, non-deterministic, build failures (also known as flaky builds), to improve the reliability of build executions. The techniques used are also further discussed in RQ_4.

A small proportion (6.2%) of the papers focused on providing datasets or analyzing the quality of data used in build optimization research. The fourth primary topic investigated in the literature is **build data**, which encompasses studies that introduced build datasets or discussed the quality of build data. In this sense, prior studies (Beller *et al.*, 2017b; Brandt *et al.*, 2020; Sulír, Bačíková, Madeja, Chodarev & Juhár, 2020; Jin & Servant, 2021a) presented large and publicly available build datasets that can be leveraged for different build optimization purposes. Other studies (Gallaba *et al.*, 2018; Ghaleb *et al.*, 2021) discussed the quality of build data used, and the potential bias that could result from using noisy datasets. Their findings are further discussed in RQ_5.

1.5.3 Techniques for Improving Build Quality

The literature has mainly relied on artificial intelligence techniques to understand, prevent, and repair build failures, as shown in Table 1.1. The table shows that machine learning, statistical analysis, NLP, and information retrieval (IR) techniques are the four most widely used techniques, respectively accounting for 40%, 35%, 10%, and 10% of the 40 examined studies (37 related to **build quality** and 3 to **flaky build detection** as depicted in Figure 1.5). The remaining studies used data mining (9.4%) and deep learning (3.1%) techniques. In the following, we provide more details on the techniques.

1.5.3.1 Identifying the Root Causes and Factors of Build Failures

Existing studies used various statistical analysis and machine learning techniques to investigate the factors causing build failures. Seo *et al.* (2014) used the open coding (Corbin & Strauss, 2008) technique to categorize build failures at Google based on historical failed build logs. They found that build failures are mostly due to compiler errors, including dependency issues, syntactic errors, and missing repository artifacts. These findings have been further validated by additional studies in industrial (Kerzazi *et al.*, 2014; Rabbani *et al.*, 2018) and open source (Zhang *et al.*, 2019) contexts. Maes-Bermejo *et al.* (2022) also investigated the reasons why past builds that were later rerun failed and came to the same conclusion. Further studies (Beller *et al.*, 2017a; Islam & Zibran, 2017; Rausch *et al.*, 2017) used statistical tests (e.g., Shapiro-Wilk Normality test[7], Wilcoxon Rank Sum test[8] and effect size measures[9]) to conduct empirical analysis on the build failure factors. In general, they have identified test failures, task complexity, and environment-related issues as the main causes of build failures. Environment-related issues have even been identified as dominant in several studies (Ghaleb *et al.*, 2021; Zolfagharinia *et al.*, 2017, 2019).

Table 1.1 Techniques investigated in the literature to improve build quality

Topic	Technique	Description	Ref.
Build Failure Analysis (23)	Statistical Analysis (14)	Studied the different categories of compiler errors in 26.6M builds using open coding for log analysis.	Seo *et al.* (2014)
		Studied build failure factors on 3k+ builds collected in an industrial context.	Kerzazi *et al.* (2014)
		Examined the impact of testing on build failures using statistical tests.	Beller *et al.* (2017a)
		Studied the cause of build failures in Java open-source software (OSS) using an open coding-based tool.	Rausch *et al.* (2017)

[7] https://real-statistics.com/tests-normality-and-symmetry/statistical-tests-normality-symmetry/shapiro-wilk-test

[8] https://real-statistics.com/non-parametric-tests/wilcoxon-rank-sum-test

[9] https://real-statistics.com/hypothesis-testing/effect-size

Continuation of Table 1.1

Topic	Technique	Description	Ref.
		Analyzed build results and their interplay with operating systems and build environments in 30M CPAN builds.	Zolfagharinia *et al.* (2017)
		Studied build result factors in a 3.6M builds dataset extending TravisTorrent.	Islam & Zibran (2017)
		Studied programming errors using Visual Studio Code interaction log data.	Rabbani *et al.* (2018)
		Studied compiler errors in 6.8M Java builds finding the 10 most common errors with their fix patterns.	Zhang *et al.* (2019)
		Studied the root cause of failures in the context of CPAN builds on different environment configurations.	Zolfagharinia *et al.* (2019)
		Analyzed failure rate and fix time of 857k failed Docker builds in OSS.	Wu *et al.* (2020b)
		Examined the errors preventing the build of past snapshots of Java projects.	Maes-Bermejo *et al.* (2022)
		Analyzed the impact of code change types on build failures using the SmartSHARK dataset.	Shimmi & Rahimi (2024)
		Studied CI practices associated with successful builds across GitHub OSS.	Wróbel *et al.* (2023)
		Studied the impact of code ownership of DevOps artifacts on build outcomes using mixed effects logistic regression and linear regression models.	Kola-Olawuyi *et al.* (2024)
	Machine Learning (9)	Analyzed build outcome factors in TravisTorrent using clustering models.	Dimitropoulos *et al.* (2017)
		Studied the factors impacting the build result in TravisTorrent using correlation analysis and feature importances of ML models.	Luo *et al.* (2017)
		Identified the most influential features explaining build failures at Atlassian using feature importances of a Logistic Regression model.	Hong *et al.* (2024)
		Compared the different categories of build errors in open source software versus industrial context using cluster analysis.	Vassallo *et al.* (2017)

Continuation of Table 1.1

Topic	Technique	Description	Ref.
		Compared the quality of real-world Dockerfiles collected from GitHub projects to a gold set of Dockerfiles using tree association rules.	Henkel *et al.* (2020)
		Examined the causal relationship between build failures and code quality factors, by training models using code and test smell datasets.	Barrak *et al.* (2021)
		Studied the relationship between build duration and build failures using ML.	Ghaleb *et al.* (2022)
		Examined the factors influencing build failure correction time using association rules mining.	Silva et al. (2023)
		Studied how contextual project factors influence build outcomes.	Benjamin *et al.* (2023)
Build Co-change Prediction (4)	Machine Learning (4)	Analyzed the factors driving build maintenance during development activity using statistical analysis and association rules.	McIntosh *et al.* (2011)
		Built Random Forest classifiers to predict and explain the need for build changes using source code and test file changes metrics.	Mcintosh *et al.* (2014)
		Predicted build co-change on new projects with small or no historical build data using data from other projects.	Xia *et al.* (2015)
		Improved the previous build co-change prediction models (Mcintosh *et al.*, 2014; Xia *et al.*, 2015)	Macho *et al.* (2016)
Automatic Build Repair (10)	Information Retrieval (4)	Summarized failed build logs and mined Stack Overflow (SO) for fixes.	Vassallo *et al.* (2018)
		Improved the tool with error localization and more accurate fixes from SO.	Vassallo *et al.* (2020a)
		Surveyed build issue resolution on SO to provide a catalog of build issues.	Lou *et al.* (2020)
		UniLoc. Leveraged build logs as search queries and source codes and build scripts as documents to improve fault localization.	Hassan *et al.* (2023)

Continuation of Table 1.1

Topic	Technique	Description	Ref.
	Natural Language Processing (4)	Developed three repair strategies for dependency issues and analyzed logs to identify the right strategy.	Macho *et al.* (2018)
		Generated patch candidates to failed Gradle build scripts using log similarity and historical fix patterns.	Hassan & Wang (2018)
		Repaired builds based on build log analysis and historical fix patterns.	Hassan (2019)
		Improved SOTA by using current code changes and online resources.	Lou *et al.* (2019)
	Deep Learning & LLMs (2)	Trained a neural network to produce the fix to compiler errors using Neural Machine Translation and historical fix patterns.	Mesbah *et al.* (2019)
		Fine-tuned the BERT model to automate the classification of build failure logs into failure fix categories.	Lee *et al.* (2024)
Flaky Build Detection (3)	Machine Learning (3)	Used job telemetry data with tree-based ML classifiers to detect flaky builds.	Lampel *et al.* (2021)
		Improved flaky build detection models using build logs' textual features.	Olewicki *et al.* (2022)
		Used knowledge graphs of build data to detect and diagnose flaky builds.	Moriconi *et al.* (2022)

Numerous studies have also investigated ML techniques to identify the most influential factors of build failures, aiding in the improvement of software development and CI practices, as well as build failure prediction performance. Dimitropoulos *et al.* (2017) used the K-means algorithm for clustering the features in the TravisTorrent dataset, to identify features that are most correlated with build results. As a result, they determined that features related to large code changes significantly impact the build outcome. Several other studies (Luo *et al.*, 2017; Barrak *et al.*, 2021; Wróbel *et al.*, 2023; Silva *et al.*, 2023) have come to similar conclusions after conducting exploratory data analysis or modeling build results with ML. Specifically, they found that in open source software (OSS), build failures are mostly explained by the build complexity

indicated by a high number of commits or code changes. Further, Silva *et al.* (2023) found that build complexity metrics and project-level characteristics also significantly influence build repair durations. Given the widespread use of container tools in CI, Henkel *et al.* (2020) assessed the quality of Dockerfiles against best practices using association rule mining and developed a set of rules for improving Dockerfiles.

Other non-code-related factors of build failures have been identified in OSS projects. For instance, Shimmi & Rahimi (2024) investigated the impact of various types of code changes on build failures using discriminant function analysis[10], and identified maintenance changes (e.g., bug fixes) as more influential than changes for adding new functionalities. Kola-Olawuyi *et al.* (2024) studied the impact of developers' code ownership of DevOps artifacts on build outcomes and ascertained that successful builds are associated with increased ownership over time and contributions from developers more experienced with DevOps. Contrasting prior studies, Benjamin & Mathew (2024) determined that contextual project-level metrics alone can sufficiently explain build failures. Earlier, Ghaleb *et al.* (2022) studied the relationship between build failures and duration and highlighted that some actions to improve build duration performance (e.g., caching dependencies) can later lead to build failures (e.g., outdated dependencies), and vice versa. Such a finding underscores the need for more holistic build optimization solutions.

To extend findings beyond OSS, Vassallo *et al.* (2017) used K-means clustering to compare build failure reasons in OSS and industrial projects, finding notable differences. In particular, this study revealed that OSS build failures typically stem from unit test failures, while industrial build failures are largely due to release preparation and deployment issues. More recently, Hong *et al.* (2024) analyzed the factors impacting build failures at Atlassian using an LR model and found that the most influential factors are related to the past build performance, such as the ratio of failed builds within the last five builds.

[10] https://real-statistics.com/multivariate-statistics/discriminant-analysis/linear-discriminant-analysis

1.5.3.2 Automating the Prevention and Repair of Build Failures

Several machine learning techniques have been evaluated to understand and predict when source code changes require build script (or CI configuration) changes (build co-changes) to prevent build failures. McIntosh *et al.* (2011) used association rule mining to identify factors driving build script maintenance and trained an RF classifier with code change metrics to predict build co-changes (Mcintosh *et al.*, 2014). They identified a strong relationship between code changes and build script changes, and obtained a prediction accuracy of up to 88%. Predicting build co-changes requires sufficient historical data about code changes, which are typically limited in newly initiated projects. To address this issue, Xia *et al.* (2015) leveraged the limited training data and non-project-specific features from older projects to train an adaptation of the AdaBoost[11] algorithm. Macho *et al.* (2016) improved the existing build co-changes prediction models by using detailed code changes and commits categories features to train RF classifiers. This technique outperformed existing models, both for intra-project and cross-project predictions.

Various techniques, from mining online resources to deep learning models, have been investigated to generate patch candidates and predict repair strategies for automatic build failure repair. In the IR techniques, Vassallo *et al.* (2018) developed a tool that uses a three-step approach to provide a summary of errors and repair hints to help developers understand and repair build failures more quickly. Essentially, the tool (1) extracts the summary and root cause of failures from build log files, then (2) mines the Stack Overflow (SO) platform to find potential repair solutions in the posts, and finally (3) presents to developers the most relevant posts identified using a keyword-based ranking algorithm. Vassallo *et al.* (2020a) evaluated the tool as relevant in practice to address failures due to compiler errors. Lou *et al.* (2020) have also mined SO to identify the different categories of build errors and how they are fixed in practice.

Repair suggestions that are retrieved from the internet often fail to apply to the specific context of projects. To remedy this, Hassan & Wang (2018) used a similarity measure of the build logs with past failed build logs, to identify previous fix patterns used in the projects. From these fix

[11] https://en.wikipedia.org/wiki/AdaBoost

patterns, they generated and evaluated patch candidates. The technique was able to repair 46% of the studied build failures taken from TravisTorrent. Similarly, Hassan (2019) pre-generated build fix templates created from historical code changes of how developers fixed past issues. To repair a failure, they used IR techniques to identify the root cause and location of errors from the build logs, then generated patch candidates using the fix templates. Mesbah *et al.* (2019) used a deep neural network to generate sets of fixes for compiler errors at Google. The model has been trained with patterns of compiler error resolution, extracted from historical code changes, and encoded into abstract syntax tree[12] (AST) features. As a result, the model inferred accurate repairs for 50% of failures related to dependency and typo issues.

Using historical fix patterns to generate patches is limited by the quality of the old fixes. To address this issue, Macho *et al.* (2018) predetermined three repair strategies for dependency-related failures: updating the version, removing the dependency, or adding a repository. In the event of a build failure, the build logs are analyzed to identify the most appropriate repair strategy before it is executed. The technique successfully repaired 54% of the studied builds. Further, Lou *et al.* (2019) improved the SOTA history-driven build repair technique (Hassan & Wang, 2018), by relying on the present build code information and online resources to generate the patch candidates. This way, they fixed twice as many build failures as the original technique.

Most existing build repair techniques focus on fault localization in the source code, disregarding the build script. To address this shortcoming, Hassan *et al.* (2023) developed *UniLoc*, an IR-based technique that uses build logs as search queries to identify in the code or build scripts the precise error locations. An evaluation of this tool on 600 build failures outperformed SOTA techniques for build error localization. A recent work (Lee *et al.*, 2024) using LLMs for build repair, labeled over 16k Java build error logs and fine-tuned a BERT model to classify fix strategies. The fine-tuned model performed exceptionally well (F1-score: 0.91-1) for categories of fix strategies with sufficient examples (≥ 14), but struggled with less frequent ones.

[12] https://en.wikipedia.org/wiki/abstract_syntax_tree

1.5.4 Build Reliability: Challenges and Current Approaches

1.5.4.1 Flaky Tests

A flaky test is a test case that exhibits non-deterministic behavior, typically due to factors such as asynchronous waits, race conditions, randomness, test order dependencies, and other test code–related issues (Luo, Hariri, Eloussi & Marinov, 2014). Such tests are particularly problematic because they may pass or fail unpredictably, allowing defects to escape detection and eroding trust in the software certification process. Consequently, a substantial body of research has focused on mitigating flaky tests to improve the reliability of testing processes and, ultimately, overall software quality.

Although these studies fall outside the scope of our SLR, which focuses on the CI/CD pipeline as a whole, flaky tests are a major contributor to unreliable pipeline failures. For this reason, we briefly review relevant research on flaky tests in this section.

Several studies have investigated language-specific methods for the automated detection of flaky tests. Early studies developed Maven plugins such as *iDFlakies* (Lam, Oei, Shi, Marinov & Xie, 2019b) and *NonDex* (Gyori, Lambeth, Shi, Legunsen & Marinov, 2016), which rely on multiple exhaustive reruns of suspected tests. However, rerun-based approaches are both inefficient and often inaccurate (Silva, Teixeira & d'Amorim, 2020). To reduce the costs associated with test reruns, Bell *et al.* (2018) introduced *DeFlaker*, an approach leveraging differential code coverage to identify failing tests that do not cover any code change as flaky. Subsequently, Silva *et al.* (2020) proposed *Shaker*, which detects flaky tests by applying environmental stressors. Another line of work (Pinto *et al.*, 2020; Camara, Silva, Endo & Vergilio, 2021; Alshammari, Morris, Hilton & Bell, 2021; Pontillo, Palomba & Ferrucci, 2021; Parry, Kapfhammer, Hilton & McMinn, 2022) has explored ML techniques for more efficient detection, classifying tests as flaky or non-flaky. These approaches rely on features derived from test code, source code, test smells, and other code-related metrics, making existing flaky test detection techniques largely programming language-dependent.

Further studies have focused on categorizing flaky test failures to support the development of techniques aimed at fixing the underlying issues. In particular, Luo *et al.* (2014) manually analyzed commit data and classified flaky tests into 11 categories. Building on this, Eck, Palomba, Castelluccio & Bacchelli (2019) conducted developer interviews to create a taxonomy with four additional categories, while Jiang, Li, Yang & Xuan (2017); Lam, Godefroid, Nath, Santhiar & Thummalapenta (2019a) developed log analysis tools to identify flaky test categories. Common categories identified, such as `async await`, `test order dependency`, `unordered collection`, `concurrency`, and `time`, have since been used in different approaches for diagnosing and repairing flaky tests. For example, *FlakyCat* (Akli, Haben, Habchi, Papadakis & Le Traon, 2023) fine-tuned LLMs for flaky test classification, enabling automated diagnosis of the most frequent categories. *FlakeSync* (Rahman & Shi, 2024) automated the repair of flaky tests caused by *async await* misuses, and more recently, *FlakyFix* (Fatima, Hemmati & C. Briand, 2024) leveraged LLMs to predict fix categories, supporting the automated repair of the most common types of flaky tests.

1.5.4.2 Intermittent Job Failures

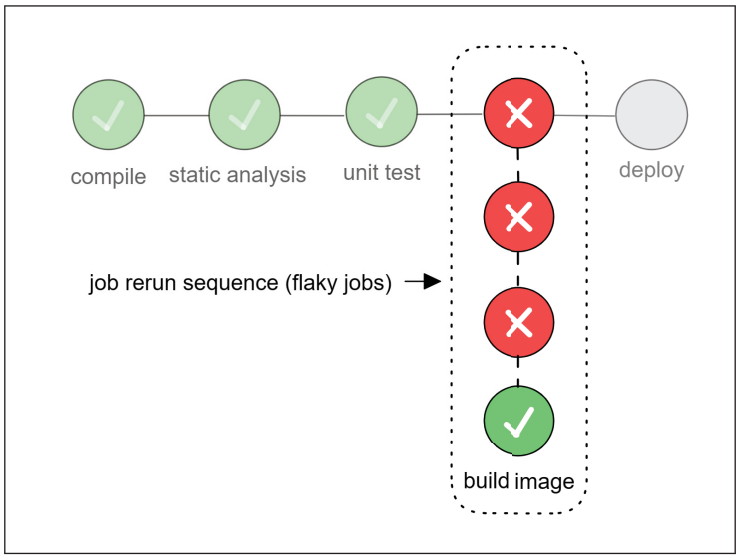


Figure 1.6 Example of a pipeline highlighting the rerun sequence of the *build image* job. After an initial failure, the *build image* job is rerun three times before passing, revealing the flakiness of the job failures in the rerun sequence

Also referred to as *flaky job failures* (Durieux *et al.*, 2020), *intermittent job failures* are non-deterministic failures of CI/CD pipeline jobs caused by irregular issues, typically related to the CI environment rather than to coding errors (Olewicki *et al.*, 2022; Durieux *et al.*, 2020). Such failures may alternately pass or fail across multiple reruns without any changes to the source code or build configuration. As a result, they undermine the reliability of the CI/CD process and are particularly challenging to detect and diagnose (Lampel *et al.*, 2021; Olewicki *et al.*, 2022; Moriconi *et al.*, 2022). Figure 1.6 illustrates the exhibition of intermittent job failures through a sequence of job reruns, starting from an initial failure. These repeated reruns impose a wasteful load on already constrained CI servers, delay feedback to developers, and ultimately slow down software development activities and delivery. In large-scale software organizations (e.g., Google), such rerun practices translate into substantial operational costs and significantly contribute to increased carbon footprints (Memon *et al.*, 2017; Olewicki *et al.*, 2022).

Unlike flaky tests, intermittent failures can arise in any pipeline job (not just testing jobs) (Lampel *et al.*, 2021). Indeed, the issues behind intermittent job failures are broader in scope and significantly more diverse than those of flaky tests. For example, transient network outages may disrupt container image publication or deployment jobs, transient issues with version control systems can cause compilation job failure, and infrastructure issues can cause failures at any stage of the pipeline. Therefore, detecting intermittent job failures requires language-independent approaches (i.e., not relying on language-specific code or test metrics). We present existing studies on such language-independent detection approaches in the following.

A limited number of studies have investigated machine learning-based approaches for detecting intermittent job failures. Lampel *et al.* (2021) proposed the first approach to language-independent detection of intermittent job failures. This approach relied on job telemetry data, including CPU load and run time as key metrics, from about 2 million jobs at Mozilla. The jobs were manually labeled over time by a dedicated team of engineers known as *build sheriffs*, and used to train an ML classifier. While this approach achieved good performances (73% precision score on average) at Mozilla, it fails to generalize to other contexts because of two main reasons: (1) the manual labeling of large number of jobs is a costly process to which most

organizations, such as Ubisoft (Olewicki *et al.*, 2022) and also TELUS, are reluctant to dedicate human resources; (2) job telemetry data like CPU load are either not accessible or can not be accurately measured in shared public cloud CI infrastructure due to multitenancy and unknown concurrent changes (Olewicki *et al.*, 2022).

At Amadeus, Moriconi *et al.* (2022) evaluated an approach leveraging Knowledge Graphs for flaky build detection, which they argued is better suited for representing heterogeneous build data. For this purpose, they created a labeled dataset of builds using Knowledge Graph Embedding[13] (KGE) methods, with which they trained an RF classifier that outperformed a baseline TF-IDF-based ML model. However, this approach remains experimental and insufficiently evaluated for generalizability.

To address these gaps, Olewicki *et al.* (2022) introduced the SOTA language-independent approach to detecting intermittent job failures. The SOTA approach leveraged the similarity in TF-IDF features (Ramos, 2003) of job logs, along with additional metrics such as the number of prior job reruns, to train a combination of two XGBoost models. Also, instead of costly manual labeling, the labeling process was automated using the non-deterministic job rerun heuristic (i.e., a job failure is labeled as intermittent if it was rerun on the same commit and changed its status to success; otherwise, it is considered regular). As a result, the SOTA approach achieved satisfying performance in the context of Ubisoft, with almost similar performance (up to 88% F1-score) as a manual classification, but faster.

1.5.5 Build Data Sources

In this section, we present metrics used to conduct build studies and leveraged in CI-improving techniques, then provide an overview of publicly available build datasets that can be reused.

[13] https://en.wikipedia.org/wiki/knowledge_graph_embedding

1.5.5.1 Build Metrics

We identified four main categories of metrics used to conduct research on the build, the most prevalent of which are code metrics. Tables 1.2, 1.3, 1.4, 1.5 present the four categories of build metrics identified, respectively: CI metrics, project metrics, developer metrics, and code metrics. As can be seen in the tables, code metrics are the most widely used category of metrics, identified in 36 (97.3%) of the 37 examined studies. Meanwhile, CI metrics, developer metrics, and project metrics were identified in 30 (81.1%), 26 (70.3%), and 21 (56.8%) studies, respectively. In the following, we present each category of build metrics.

Table 1.2 List of CI metrics. Each metric is described with its name, its data type, a short description, and a list of study references where it has been used

ID	Metric	DT[a]	Description
B_1	On default branch	C	Whether the build is triggered on the default branch or not.
B_2	Build type	C	The type of build: push, pull request, merge, or unknown.
B_3	Time of the day	C	Adjusted hour of the day [0, 23] of the commit that triggered the build (or of the build itself).
B_4	Day of the week	C	Adjusted weekday [0, 6] (0 being Monday) of the commit that triggered the build (or of the build itself).
B_5	#Jobs	N	Number of jobs executed in the build.
B_6	Build duration	N	Total duration of the last build.
B_7	Build wait time	N	Wait (or queued) time before executing the build jobs.
B_8	Build status	C	Result of the last build: passed, failed, errored, timed out, or canceled.
B_9	CI lifespan	N	Time difference between the project's last and first builds.
B_{10}	Time interval	N	Interval of time since the last triggered build (or commit).
B_{11}	Failure distance	N	The number of builds since the last build failure.
B_{12}	Build performance	N	Ratio of failed builds in the previous builds.
B_{13}	Failed builds fix time	N	Time taken to fix a build failure in a specified time interval.
B_{14}	Job type	C	Type of the build job (e.g., compilation, testing).
B_{15}	Job environment	C	Environment where the build job was run (e.g., JDK version).
B_{16}	Job logs	N	Execution logs, typically transformed into textual features.
B_{17}	CI Server OS	C	Name and distribution of the OS used to run the build job.

[a] **Data Type (DT)**: Categorical (C) – Numeric (N).

CI metrics are related to the current build, as well as the historical build activity and performance of projects. Table 1.2 summarizes the CI metrics identified in the literature. The raw data for calculating these metrics can be collected mainly from the CI system[14] and the VCS. The CI metrics can be grouped into four sub-categories. First, metrics that pertain to the characteristics of the build before its execution (B_1 - B_5). Second, metrics related to a single build execution performance and outcome (B_6 - B_8). Third, metrics that provide general insights into the historical performance of the CI process and are calculated using past build data (B_9 - B_{13}). Finally, the fourth sub-category includes job-specific metrics such as execution logs and details about the environment used to run the build jobs (B_{14} - B_{17}). These job-specific metrics are the least used in the literature, found in only five studies (Ghaleb *et al.*, 2022; Lampel *et al.*, 2021; Barrak *et al.*, 2021; Olewicki *et al.*, 2022; Brandt *et al.*, 2020).

Project metrics provide high-level information about the characteristics and maturity of the projects. These metrics include the age of the project (P_3), its size in lines of code (P_4), and other general information on the project such as the dominant programming language (P_1) and the size of the team contributing to the project (P_2). Table 1.3 summarizes the project metrics identified. These metrics are typically extracted from the VCS.

Table 1.3 List of project metrics

ID	Metric	DT	Description
P_1	Language	C	The main programming language of the project.
P_2	Team size	N	The number of contributors in the team working on the project.
P_3	Project age	N	Time difference (in days) between the last executed build and the project creation date.
P_4	Project size	N	Size of the project in Source Lines Of Codes (SLOC).
P_5	Test density	N	Median number of test cases or test lines per 1k SLOC.

Developer metrics provide information regarding the developers who authored the triggered build. Table 1.4 summarizes the developer metrics identified, that can be collected from the

[14] An example of data source for the CI metrics on the GitLab CI system is the GitLab Pipelines API

Table 1.4 List of developer metrics

ID	Metric	DT	Description
D_1	#Contributors	N	Number of distinct authors of the code changes since the last build.
D_2	Core developer	C	Whether the developer who triggered the build has committed at least one in the last 3 months.
D_3	Dev. experience	N	Number of days (or commits) the developer who authored the build has been contributing to the project.
D_4	Dev. failure rate	N	Failure rate of the builds authored by the developer in the project history.

VCS. These metrics include the number of contributors (D_1), the role (D_2) and experience of the developer who triggered the build with contributions (D_3) and with failures (D_4).

Table 1.5 List of code metrics

ID	Metric	DT	Description
C_1	Commits	N	Number of unique commits pushed since the last build. Also known as the number of revisions (Kawalerowicz & Madeyski, 2023).
C_2	Files changed	N	Number of files/lines changed by the commits in the build.
C_3	Files deleted	N	Number of files/lines deleted by the commits in the build.
C_4	Files added	N	Number of files/lines added by the commits in the build.
C_5	Tests added	N	Number of added test cases.
C_6	Tests deleted	N	Number of deleted test cases.
C_7	Imports added	N	Number of import statements added.
C_8	Classes changed	N	Number of classes changed, added, or deleted.
C_9	Source churn	N	Lines of source (src.) code changed by the commits in the build. Also referred to as the commit length by Benjamin & Mathew (2024).
C_{10}	Test churn	N	Lines of test code changed by the commits in the build.
C_{11}	Src. files changed	N	Number of source files changed by the commits in the build.
C_{12}	Doc. files changed	N	Number of documentation files changed by the commits in the build.
C_{13}	Config. files changed	N	Number of configuration (e.g., `.xml` and `.yml`) files modified by the commits in the build.
C_{14}	Other files changed	N	Number of other files changed by the commits in the build.

Code metrics encompass metrics related to code changes (including build configuration changes) associated with the triggered build. Table 1.5 summarizes the identified code metrics.

These metrics measure the number of code changes made overall in the project (C_1, C_2), the number of different types of changes, (C_3 - C_8), and the number of changes affecting specific types of files (C_9 - C_{14}). The code metrics can be collected from the VCS and using static code analysis tools such as SciTools Understand.[15] Depending on the context, C_1 and C_2 can also be considered as pull request (PR) metrics (as suggested by Hong *et al.* (2024)) given that they both reflect the changes associated with the build's PR.

1.5.5.2 Public Datasets

Table 1.6 List of publicly available build datasets

Name	# Builds	CI Service	Description	Year
TravisTorrent	2,640,825	Travis CI	A dataset of .csv and log files collected from 1,300 Java and Ruby projects using Travis CI; for build failure analysis.	2017
CI Compiler Errors	6,854,271	Travis CI	3.8TB of .json and log files created from builds in 3,799 Java projects using Travis CI. Created for build failure analysis.	2019
LogChunks	797	Travis CI	Collection of labeled log files collected from 80 projects (29 programming languages) using Travis CI, for build log analysis.	2020
Local Java Builds	14,497	N/A	A dataset of .csv and log files collected local build results of 20,000 projects written in Java. Created for build failure analysis.	2020
CIBench	82,427	Travis CI	Extended TravisTorrent using builds from 100 projects, to evaluate build selection and prioritization techniques.	2020
Mozilla CI Jobs	1,997,075	Taskcluster	CI job telemetry data stored in SQL database to support flaky builds detection.	2021
SmartSHARK	89,160	Travis CI	740 GB MongoDB data from VCS, CI service, and issue tracking across 77 Java projects for software evolution analysis.	2021
Deps Updates Builds	10,412	GitHub Actions	3.3GB of build data related to dependency updates commits of JavaScript projects, to analyze unused dependencies' wastes.	2023

A total of eight publicly accessible build datasets have been identified in the literature, to support various build research topics, with build failure analysis being the most prominent.

[15] https://scitools.com/

Table 1.6 presents the publicly available build datasets identified. It shows the name of the dataset, its size, the associated CI system, and a short description including the data format, the programming languages covered, and the original purpose of the dataset. The dataset name is clickable and is a link to its DOI or download page. Four of the eight datasets have been the exclusive topic of papers (in which they were introduced) (Beller *et al.*, 2017b; Brandt *et al.*, 2020; Sulír *et al.*, 2020; Jin & Servant, 2021a), while the remaining datasets have been made available as part of studies in different contexts (Zhang *et al.*, 2019; Lampel *et al.*, 2021; Shimmi & Rahimi, 2024; Weeraddana *et al.*, 2024b). The TravisTorrent (Beller *et al.*, 2017b) and CI Compiler Errors (Zhang *et al.*, 2019) datasets are outstanding for the included number of builds, containing metadata and log files collected across 2.6 and 6.8 million builds, respectively. In addition, the Mozilla CI Jobs (Lampel *et al.*, 2021) dataset contains telemetry data of nearly 2 million CI jobs. The remaining of the datasets (Brandt *et al.*, 2020; Sulír *et al.*, 2020; Jin & Servant, 2021a; Shimmi & Rahimi, 2024; Weeraddana *et al.*, 2024b) cover a more limited number of builds, ranging from 797 to 89k.

The available build datasets support a wide range of build-related research topics. In particular, the widely used TravisTorrent dataset has enabled studies on build failures, prediction, selection, and repair. SmartSHARK offers similar potential but has seen limited use in build studies so far. Extending TravisTorrent, CIBench focuses on evaluating build selection and prioritization. Datasets like LogChunks support log analysis, while Local Java Builds and CI Compiler Errors target Java build failures. Additionally, Mozilla CI Jobs focuses on flaky build detection, and Deps Updates Builds supports research on dependency-related waste.

Most of the existing build datasets focused on projects in Java and using Travis CI. As shown in Table 1.6, out the eight datasets, five (62.5%) (Beller *et al.*, 2017b; Zhang *et al.*, 2019; Brandt *et al.*, 2020; Jin & Servant, 2021a; Shimmi & Rahimi, 2024) are created from projects using Travis CI. Also, five (62.5%) datasets (Beller *et al.*, 2017b; Zhang *et al.*, 2019; Sulír *et al.*, 2020; Jin & Servant, 2021a; Shimmi & Rahimi, 2024) are created from projects mainly written in Java. In particular, the Local Java Builds and CI Compiler Errors datasets contain only

Java builds. As a result, the vast majority of existing build studies have been carried out in the context of the Java programming language and the Travis CI system.

1.6 Discussion and Opportunities

To mitigate recurring build failures, so as to improve software quality and team productivity, prior studies have first focused on understanding their root causes, which has enabled the development of techniques that can support the fully automated resolution of build failures. Several studies (Barrak *et al.*, 2021; Beller *et al.*, 2017a; Rausch *et al.*, 2017; Zhang *et al.*, 2019; Luo *et al.*, 2017; Islam & Zibran, 2017) have investigated the root causes of build failures using statistical models. While there seems to be a consensus on the association of *regular* code complexity (e.g., high number of commits and code changes) with frequent failures, other important factors such as developer experience and ownership have also been identified. The insights from these studies proved foundational in developing relevant solutions to address build failures. In particular, a large body of research developed techniques to help practitioners anticipate build failures by, for instance, predicting their occurrence prior to execution (e.g., Saidani *et al.* (2021a); Wolf *et al.* (2009); Sun *et al.* (2024); Chen *et al.* (2020)). If failures still occur, several techniques for diagnosing root causes and locating bugs have been proposed to assist developers (e.g., Hassan *et al.* (2023); Lee *et al.* (2024); Lou *et al.* (2020)). Finally, complementing these solutions with repair techniques, mainly based on ML (e.g., Mesbah *et al.* (2019); Hassan & Wang (2018); Lou *et al.* (2019)), can enable the complete automation of the early detection, diagnosis, and repair workflow for regular code-related failures.

Several studies have highlighted the limitations of studying builds in too narrow contexts, suggesting the need for insights derived from data related to other CI platforms and richer contexts. Many important aspects of the build (e.g., configuration, environment) are tightly coupled to the build tool and CI system used. Hence, studying builds in the context of a limited number of these tools can hinder the generalization of results and limit their applicability at a broader scale (Mcintosh *et al.*, 2014; Abdalkareem *et al.*, 2021b; Saidani *et al.*, 2021a; Lampel *et al.*, 2021). These limitations are particularly significant because Travis CI, extensively

studied in the literature, has been replaced by GitHub Actions as the most commonly used CI tool (Golzadeh *et al.*, 2022). Additional studies outlined the difficulty of adopting existing techniques due to programming language specifics (Abdalkareem *et al.*, 2021b; Olewicki *et al.*, 2022). In particular, Zolfagharinia *et al.* (2019) argued that the differing specific requirements of programming languages (such as runtime environments and packages) make it challenging to reuse CI-improving techniques across languages. Also, Mcintosh *et al.* (2014) noted a performance difference between Java and web-based projects when predicting build co-changes, due to the peculiarities of the build technologies used in each context. Therefore, future research should more carefully design solutions with practical relevance in mind, grounding them in up-to-date CI environments (e.g., contemporary systems, tools, and programming languages) and fostering closer collaboration with industry to support real-world adoption.

Although irregular issues increasingly contribute to failures, existing contributions remain limited and mainly focus on their detection. Nevertheless, approaches developed to mitigate regular failures can inspire future research aimed at addressing these unreliable pipelines. Several studies highlight environmental factors, infrastructure issues, and other unexpected non-code-related errors as the major contributors to build failures (Gallaba *et al.*, 2018; Rausch *et al.*, 2017; Lampel *et al.*, 2021). These issues were first treated as noise in datasets aimed at training build outcome models (Gallaba *et al.*, 2018). However, due to their increased prevalence causing important inefficiencies, recent studies have focused on their automated detection (Lampel *et al.*, 2021; Olewicki *et al.*, 2022; Moriconi *et al.*, 2022), with emerging interest in their systematic diagnosis (Lampel *et al.*, 2021). Therefore, we call for further research on the diagnosis and repair of unreliable builds, an area that remains largely underexplored. Finally, to foster practical adoption, we suggest more empirical studies conducted in industrial settings, where unique challenges related to deployment, scale, and build flakiness arise. In addition, greater attention should be devoted to the design and evaluation of novel architectures that enable CI-improving solutions to be effectively deployed and operated in real-world environments.

1.7 Conclusion

We conducted a systematic literature review of 97 papers to better understand the existing practices for build optimization and guide future studies on CI build improvement. More than three-quarters (88.7%) of the papers were published after 2017, following the release of the TravisTorrent dataset, which was used to carry out almost half (41.2%) of the build studies. Further, the SLR shows that 95.8% of the build studies have leveraged different build datasets, out of which we have identified 8 publicly available that can be used for future build studies.

Our analysis of the 97 papers shows that build studies have focused on two primary challenges: long build times and build failures. Indeed, these two challenges have been associated with significant infrastructure and human (reduced productivity) costs, estimated in millions of dollars in large organizations (Zhang *et al.*, 2022). Beyond the papers addressing these primary challenges, the remaining papers fall into two categories: general analyses of CI system adoption and quality, and studies examining the datasets used in build-related research.

To optimize the build and mitigate its costs, existing studies have developed several techniques and approaches, most of which are based on ML algorithms. For example, ML-based build selection techniques have been developed to minimize execution costs, while predicting build outcomes and durations has proven effective in reducing developer waiting times. As for build failures, existing studies have generally focused on root cause analysis, build maintenance prediction, and automatic build repair. In addition, recent efforts have focused on detecting flaky builds, most often caused by instabilities within the CI system, such as memory errors.

This SLR identifies several research gaps that warrant further investigation. In particular, relatively few studies have been conducted in industrial settings or within the context of modern CI platforms, such as GitHub Actions and GitLab CI, which limits the practical adoption of existing techniques. Expanding the availability of build datasets for these platforms would substantially benefit both practitioners and the research community. Moreover, while recent work has increasingly focused on detecting intermittent build failures, the diagnosis and repair of such failures remain largely underexplored and warrant attention to improve build reliability.

CHAPTER 2

OBJECTIVES

This chapter presents the research objectives guiding this thesis. It first introduces the conceptual framework for addressing pipeline reliability issues, derived from the SLR in Chapter 1, then formulates the specific research objectives (ROs) that structure the contributions presented in Chapters 4 to 8.

2.1 Conceptual Framework

The SLR revealed that while build optimization has received considerable attention, pipeline reliability remains underexplored despite rising challenges. As discussed in Chapter 1 (Section 1.6), existing work has mainly focused on detecting intermittent failures, with very limited attention to their systematic diagnosis and repair. Moreover, silent failures (i.e., jobs that incorrectly report success) have received almost no attention in spite of their high potential to propagate defects into production undetected.

Building on the SLR findings, we propose a conceptual framework for addressing pipeline reliability issues, illustrated in Figure 2.1. Inspired by existing work tackling regular failures, the framework organizes the problem space into three sequential phases:

1. **Detection.** Identifying that an anomaly has occurred. For intermittent (or *flaky*) failures, this involves distinguishing determinism- and environment-related issues from genuine bugs. For silent failures, this involves recognizing that a reported success is, in fact, a failure.
2. **Diagnosis.** Understanding the nature and root cause of the detected anomaly. This includes categorizing the failure, identifying priority categories for automated triage and resolution, and surfacing the relevant information to accelerate troubleshooting.
3. **Repair.** Resolving the issue, either automatically through scripts and API calls, or by routing actionable information to the appropriate team for manual intervention.

Figure 2.1 illustrates how these phases compose into an automated workflow. Upon a build failure event, the *Flaky Build Detection Model* first determines whether the failure is intermittent

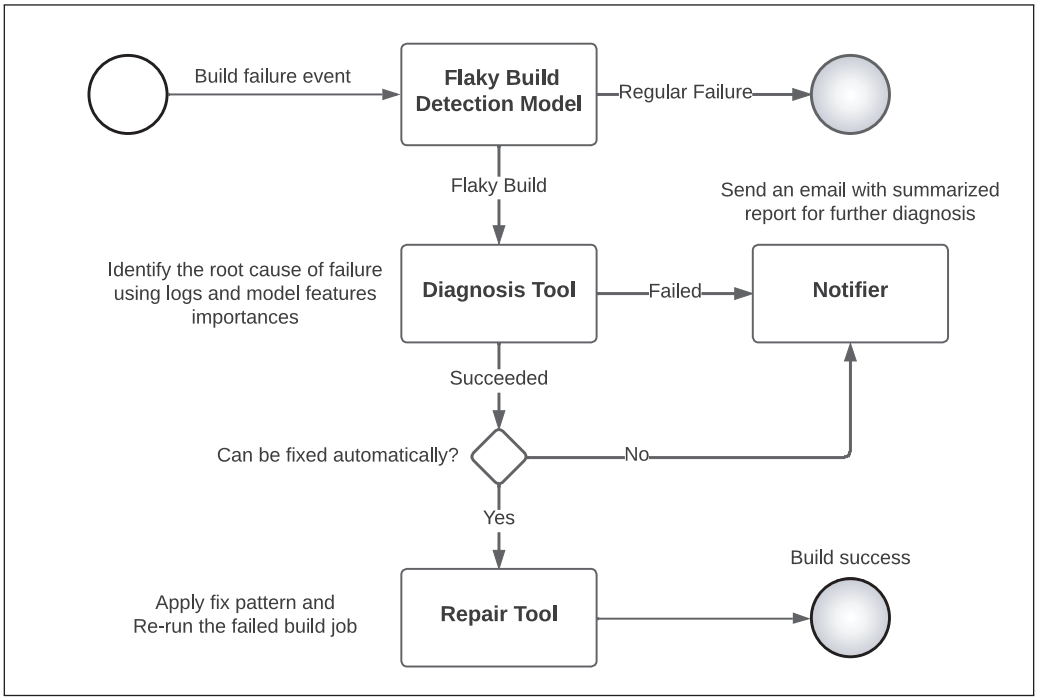


Figure 2.1 Conceptual framework for the automated detection, diagnosis, and repair of pipeline reliability issues

or regular. Regular failures, which indicate genuine code defects, follow the standard developer debugging workflow and fall outside the scope of this framework. Detected intermittent builds are passed to the *Diagnosis Tool*, which attempts to identify the root cause using logs and model feature importances. If diagnosis fails, the *Notifier* dispatches a summarized report by email for further manual investigation. On success, the identified failure type is looked up against a registry of available repair scripts to determine whether an automated fix exists. If one is found, the *Repair Tool* applies the appropriate fix pattern and re-runs the failed job, aiming for build success. Otherwise, the case is routed to the *Notifier* for human intervention. This framework guided the formulation of research objectives and the design of contributions presented in this thesis.

2.2 Research Objectives

The primary goal of this thesis is twofold:

- To advance the understanding of pipeline reliability challenges in modern CI/CD systems.
- To develop practical automation techniques that mitigate their associated inefficiencies.

To achieve these goals, this thesis combines empirical investigation with the development of practical automation solutions grounded in efficient and adaptable machine learning techniques. Guided by the conceptual framework, this work addresses five specific research objectives:

RO_1 Develop an efficient approach for detecting intermittent job failures.

Automated detection of intermittent failures is the foundational step of the automation chain. Before any diagnosis or repair can be automated, the system must first reliably identify that an anomaly has occurred. Existing approaches rely on large labeled datasets that are impractical to obtain and prone to mislabeling. This objective seeks an efficient detection approach that achieves high performance under limited labeled data (Chapter 4).

RO_2 Develop a systematic approach for understanding and prioritizing intermittent failure categories.

Intermittent failures arise from diverse root causes, including network instabilities, resource exhaustion, and non-deterministic tests. Effective mitigation requires understanding this diversity and prioritizing categories based on their organizational impact. This objective aims to develop a systematic categorization of intermittent failures and a prioritization approach to focus automation efforts on the most wasteful categories (Chapter 5).

RO_3 Develop an automated triage approach to accelerate intermittent failure diagnosis.

Once priority categories are identified, failures must be automatically classified and relevant diagnostic information surfaced to accelerate the troubleshooting process. This objective seeks an automated triage approach that accurately predicts intermittent failure categories and highlights the most relevant log segments for diagnosis (Chapter 6).

RO_4 Investigate the contributing factors and mitigation strategies for silent failures.

Silent failures represent a complementary reliability challenge: jobs that report success despite having failed. Unlike intermittent failures, silent failures have received little

to no attention in the literature. So, this objective aims to characterize silent failures, identify factors associated with their occurrence, and catalog practical mitigation strategies (Chapter 7).

RO_5 Develop a framework enabling the practical deployment of CI-improving solutions. Individual detection and diagnosis solutions require a unifying infrastructure for deployment in production environments. This objective seeks a framework that supports continuous data integration, real-time monitoring, and orchestration of automation services (Chapter 8).

CHAPTER 3

METHODOLOGY

This chapter presents the research methodology adopted to address the objectives outlined in Chapter 2. It provides an overview of the research paradigm, the industrial and open-source contexts in which the research was conducted, the datasets used, the research methods and evaluation approaches. Detailed procedures specific to each study are presented in the Study Design sections of Chapters 4 to 8.

3.1 Research Paradigm

This thesis integrates two complementary research paradigms.

Empirical Software Engineering (Runeson & Höst, 2009; Stol & Fitzgerald, 2018) guides the investigation of pipeline anomalies through systematic observation, quantitative analysis of collected data, and practitioner inquiry. This paradigm informs the characterization of intermittent and silent failures and the construction of failure taxonomies (RO_2, RO_4).

Design Science Research (Hevner, March, Park & Ram, 2004; Wieringa, 2014) guides the development and evaluation of artifacts that address identified problems. Each artifact follows a cycle of problem identification, design, implementation, and evaluation (RO_1, RO_3, RO_5).

These research paradigms interact throughout the thesis. On the one hand, empirical findings reveal requirements that inform artifact design, and on the other hand, artifact evaluation generates insights that refine understanding of pipeline reliability issues.

3.2 Industrial Collaboration

This research was conducted within the ÉTS Kaloom-TELUS Industrial Research Chair in DevOps. The author served for over three years as a full-time Mitacs Accelerate Fellow at TELUS, a major Canadian telecommunications company. This collaboration provided sustained access to (1) production CI/CD infrastructure used to deliver software versions across dozens of projects,

(2) domain experts, and (3) historical execution data spanning six years (2018–2024). Specifically, the work took place within the TELUS Intelligent Network Automation and Analytics (TINAA) ecosystem, which comprises over 100 actively maintained projects. These projects serve diverse purposes, including network controllers, web portals, analytical dashboards, REST APIs, API gateways, infrastructure-as-code automation, and production engineering.

3.2.1 CI/CD at TELUS

The TELUS CI/CD environment presents characteristics that make it particularly prone to pipeline reliability issues. TELUS mainly use a self-hosted GitLab CI instance for configuring and running CI/CD pipelines. Hence, pipelines execute on self-hosted runners deployed across distributed environments, introducing higher variability in resource availability and runner state. pipeline jobs frequently interact with software-defined network (SDN) controllers, virtual network functions, and hardware devices, creating dependencies on external services that may experience transient unavailability. Pipelines also heavily rely on container registries, artifact repositories, and third-party APIs, each a potential point of failure. Additionally, stringent security requirements introduce authentication layers, network policies (e.g., virtual private networks), and secrets management that are subject to misconfiguration, can fail intermittently, or become obsolete. These CI/CD environment characteristics, combined with an organizational structure in which responsibilities are distributed across application, network, and platform engineering teams (with developers often lacking direct access to environment variables, deployment servers, or job runners) result in prolonged diagnosis times, averaging two business days and extending up to a week for complex cases, as noted during interviews.

3.2.2 Practitioner Engagement

At the outset of the research, semi-structured audio-recorded interviews (approximately one hour each) were conducted with five TELUS engineers. The interviews followed a structured guide (Appendix I) organized around four themes: (1) demographics, including background, experience with CI/CD, current position, and team responsibilities; (2) CI/CD pipeline architecture, including

workflow stages, tooling, and infrastructure; (3) pipeline failures, including frequency, failure rates, and prevalence of non-code-related issues; and (4) hard-to-resolve wasteful failures, focusing on recurring failure types requiring long and manual resolution process, their symptoms, diagnosis procedures, and fix strategies. Participants' experience ranged from 3 to 15 years and spanned application development, network engineering, and platform operations teams. These interviews informed the initial problem formulation, revealed key pain points such as environment misconfigurations and infrastructure-related failures, and (along with the gaps identified through the SLR) guided the identification of research priorities. Throughout the research, regular exchanges continued with the same practitioners, following a structured cadence: weekly or bi-weekly during active data collection, tool development, or evaluation, and on-demand during follow-up periods. These ongoing interactions validated research findings (e.g., failure taxonomies) and assessed the practical relevance and utility of developed artifacts.

3.2.3 Constraints and Implications

Working within an industrial setting introduced some constraints. Confidentiality requirements limited publishable data granularity, as logs often contain sensitive information. Nevertheless, these constraints guided the development of data-efficient approaches (few-shot learning) and interpretable outputs (surfacing relevant log statements) suited to broader industrial adoption. Moreover, validation on open-source projects, presented below, further supports the generalizability of findings beyond TELUS.

3.3 Datasets

This section summarizes the datasets used across the thesis. Detailed data collection procedures, sampling strategies, and labeling processes are described in the *Study Design* section of each thesis chapter (Chapters 4-8).

3.3.1 Industrial Data

Pipeline data were collected from TELUS through the GitLab REST API, which enables the extraction of pipeline metadata, job logs, and execution metrics. In collaboration with TELUS engineers, we selected 14 main GitLab groups containing actively maintained projects, excluding inactive and toy projects (having fewer than 50 commits). This yielded 81 projects written in 13 programming languages and varying in size, maturity, and purpose (e.g., network controllers, data analytics services, APIs, web portals, etc.). Table II-1 in Appendix II presents the characteristics of each project.

The complete CI/CD job history of these projects, covering 6.5 years (February 2018 to July 2024), was extracted and resulted in 157,496 job metadata records. After filtering out non-completed jobs (e.g., `skipped`, `canceled`), 140,567 successful and failed jobs remained, from which 31,058 failure logs (1.3GB) were collected. Additional data collection over time increased the total to 142,387 jobs across the 81 studied projects.

From this corpus, subsets were derived for specific objectives:

- **RO_1** (Efficient Detection): 2,407 manually labeled jobs from 5 projects (due to tedious manual labeling involved), selected for their diversity and engineer availability for validation.
- **RO_2** (Categorization and Prioritization): 4,511 intermittent job failures from the 80 projects, labeled using a semi-automated regex-based tool developed from a representative sample.
- **RO_3** (Automated Triage for Diagnosis): 2,458 intermittent job failures covering 13 of the 14 priority categories identified in RO_2. A priority category was excluded due to a lack of sufficient examples for the experimental setting. This is discussed in more detail in Chapter 6.
- **RO_4** (Silent Failures Understanding): The full 142,387 completed jobs across 81 projects.
- **RO_5** (Deployment Framework): Continuous data ingestion from real-time webhook events.

3.3.2 Open-Source Data

To assess generalizability beyond TELUS, the 10 most-starred public GitLab projects at the time of selection (March 2023) were studied. Starting from the highest-starred projects, identified via the GitLab REST API, those meeting one of the following exclusion criteria were excluded:

- Do not use CI (No CI configuration file)
- Restricted Pipeline API access (403 Forbidden errors)
- Excessive data volume (job data load beyond collection capacity, e.g., the `gitlab/gitlab` project running hundreds of jobs per pipeline requiring Big Data management tools)

The final selection comprises 10 popular open-source projects (e.g., Wireshark, Inkscape, Graphviz, OpenRGB, Veloren) spanning 8 main programming languages and diverse domains, including network analysis, vector graphics edition, graph visualization, multiplayer game, and more. Table II-2 in Appendix II presents the characteristics of each OSS project.

Together, these OSS projects represent about 1.7 million jobs totalling 25GB of pipeline data made publicly available (Aïdasso, 2025a). Notably, their flaky failure ratio (FFR), i.e., the proportion of failed jobs that succeed upon rerun, ranges from 0.47% (HomelabOS) to 34.25% (OpenRGB), with an overall average of 16.57%. These proportions confirm that intermittent failures are also prevalent across diverse open-source projects, not only in industrial settings, reinforcing the relevance of this work beyond TELUS.

Subsets of these OSS projects were used for specific objectives:

- **RO_1** (Efficient Detection): Five OSS projects: Veloren, Inkscape, OpenRGB, GitLab Runner, and F-Droid. Veloren, which exhibits the highest FFR among the five, is the only one presented in the main paper due to space constraints. Results for the remaining four projects are presented and discussed in Appendix III.
- **RO_4** (Silent Failures Understanding): Eight projects with sufficient rerun data, excluding AuroraStore and HomelabOS, were used to replicate the rerun and silent failure analysis conducted at TELUS.

- **RO_5** (Deployment Framework): All ten projects were used to validate the CBDT Framework, demonstrating its applicability to diverse project settings.

Categorization (RO_2) and triage (RO_3) were not replicated on open-source projects because failure categories are inherently tied to organizational context, and rigorous analysis required substantial manual work. The specific categories observed at TELUS (e.g., environment variable misconfigurations, authentication failures, transient network issues, and server overloads) reflect TELUS's CI architecture, security requirements, and reliance on distributed services. Other organizations face different reliability challenges shaped by their own infrastructure. For instance, prior work reports that intermittent failures at Ubisoft arise primarily from heterogeneous repositories containing audio, animation, and graphics files that are difficult to test (Olewicki *et al.*, 2022), while failures at Mozilla stem largely from test suites with large file volumes that overload servers (Lampel *et al.*, 2021). A broader discussion of context-dependent failure patterns and their implications for generalizability is presented in Chapter 9.

That said, the underlying causes of many TELUS-specific categories, such as container registry errors, container issues, dependency resolution failures, and timeout issues, stem from widely adopted CI tools, including Docker, Kubernetes, SonarQube, and GitLab CI, for instance. Similar failure patterns were indeed observed in the open-source projects during manual analysis, suggesting that while category distributions vary, the detection and triage methodologies developed in this thesis remain applicable to other CI environments.

Besides, findings were also compared against prior studies (Durieux *et al.*, 2020; Olewicki *et al.*, 2022), which reported similar intermittent failure patterns.

3.3.3 Data Sources

This section describes the three primary data sources used across the research objectives. Each source captures a different facet of CI/CD failures, including unstructured execution logs, structured job metadata, and expert domain knowledge.

Job logs. Job logs capture the full textual output produced during the execution of a CI pipeline job. A single log typically interleaves several categories of information, such as infrastructure-level events (e.g., runner provisioning, git operation traces, timeout signals, environment variable prints), build tool output (e.g., compiler messages, dependency resolution, code stack traces), container tool traces (image build output, image push/pull results), test framework output (e.g., assertion failures, test failure reports, test suite summaries), and tool-specific diagnostics (e.g., linter warnings, static analysis reports). Because logs are unstructured free text, their length, format, and verbosity vary considerably across projects and toolchains. This makes them rich but noisy inputs for failure detection (RO_1), categorization (RO_2), and triage (RO_3), where the core challenge lies in extracting relevant failure signals from heterogeneous, often verbose, output.

Job metrics. Job metrics are the structured metadata that CI platforms record for every job and pipeline execution. These include job status (e.g., `success`, `failed`, `canceled`), duration, queue and start timestamps, stage, execution branch, runner type, etc. Unlike logs, metrics are tabular and consistently formatted, making them suitable for quantitative and longitudinal analysis of influential factors (RO_4).

Practitioner input. Practitioner input was gathered through semi-structured, audio-recorded interviews with five TELUS engineers. This input served three purposes: informing the initial problem formulation by identifying pain points that are difficult to observe from logs and metrics alone (e.g., diagnosis requiring cross-team collaboration), validating the failure taxonomies (RO_2, RO_4) against real-world experience, and assessing the practical utility of the developed artifacts (RO_1, RO_3, RO_5) through feedback on relevance, completeness, and actionability.

3.4 Research Design

Table 3.1 maps each research objective to its paradigm, methods, and evaluation approach. Design decisions throughout the thesis reflect practical constraints from the industrial context. Below are discussed the key decisions that guided the main contributions of this thesis. Specific methodology details can be found in the Study Design section of Chapters 4 to 8.

Table 3.1 Research design by objective.

RO	Paradigm	Methods	Evaluation
RO_1	Design science	• Statistical sampling • Few-shot fine-tuning • Binary classification	• F1-score, Precision, Recall • Monte Carlo Cross-Validation • Comparison with SOTA • Replication on OSS projects
RO_2	Mixed	• Open coding • Statistical sampling • Semi-structured interviews • Descriptive analysis • RFM analysis • K-means clustering	• Labeling tool precision and recall • Taxonomy coverage and saturation • Cluster quality (SSE)
RO_3	Design science	• Few-shot fine-tuning • Multi-class classification • SIS interpretability	• F1-score, Top-2 Accuracy • Monte Carlo Cross-Validation • Comparison with SOTA encoder • Relevance assessment
RO_4	Empirical	• Statistical sampling • Descriptive analysis • Semi-structured interviews • Mixed-effects logistic regression • Thematic analysis	• Discrimination (AUC, AUC-PR) • Calibration (Brier score) • Explained variance (R^2) • Cohen's Kappa score • Theme saturation • Replication on OSS projects
RO_5	Design science	• Architecture design • Prototype implementation	• Functional validation

3.4.1 Few-Shot Learning

Organizations cannot realistically dedicate resources to labeling thousands of failure instances, yet accurate detection and classification of reliability issues remain essential (RO_1, RO_3). CI/CD job logs serve as the primary data source for failure classification, framing this as a natural language processing (NLP) problem. However, unlike typical natural language text, raw logs are noisy and verbose, containing timestamps, file paths, stack traces, and repetitive boilerplate that obscure the underlying failure signal. Careful preprocessing is therefore required to reduce noise and help models capture discriminative patterns. Pre-trained sentence transformers, such as BERT (Devlin, Chang, Lee & Toutanova, 2019), can then encode the preprocessed logs into dense semantic representations. Few-shot fine-tuning of these models achieves strong

classification performance with minimal examples per class (Tunstall *et al.*, 2022), reducing labeling burden while improving robustness and maintainability over approaches relying on large automatically labeled datasets or manually defined regexes.

3.4.2 Open Coding

Categorizing intermittent failures (RO_2) requires systematic analysis of unstructured log data without preconceived classifications. Open coding, a qualitative method from grounded theory (Strauss & Corbin, 1998), enables inductive derivation of failure categories directly from log content through iterative refinement until consensus. To scale this process, representative sampling is used to select a statistically representative subset of intermittent failures for manual analysis. The resulting codes inform the development of a labeling tool that maximizes precision, enabling reliable broader coverage of the failure population with reduced manual effort.

3.4.3 RFM Analysis

Frequency alone does not capture the organizational impact of a failure category. In fact, the most frequent categories are not necessarily the most costly or recent. Recency, Frequency, and Monetary cost (RFM) analysis (Birant, 2011), established in customer analytics, enables context-sensitive prioritization tailored to organizational needs (RO_2). This prioritization informs downstream automation efforts in two ways. First, it focuses the multi-class classifier for triage (RO_3) on a tractable set of high-impact categories, improving classification performance. Second, it enables engineering teams to focus efforts on developing targeted remediation scripts for the most wasteful and current failure categories.

3.4.4 Sufficient Input Subset Interpretability

Practitioners need actionable information to accelerate intermittent failure resolution (RO_3), not just root cause category predictions. Sufficient Input Subsets (SIS) (Carter, Mueller, Jain & Gifford, 2019) is a model-agnostic interpretability method that treats the classifier as

a black box and leverages (recursive binary) occlusion to identify the most influential input segments. By iteratively masking portions of the logs and observing changes in predictions, SIS isolates the minimal subsets, i.e., log segments, sufficient for each predicted category. This approach reduces log review effort without requiring model retraining, access to internal weights, or architectural modifications, making it practical and efficient.

3.4.5 Mixed-Effects Logistic Regression

Silent failures are associated with factors at multiple levels: job characteristics, developer practices and experiences, and project-level features. Because jobs are nested within projects, standard logistic regression violates the independence assumption. Mixed-effects logistic regression accounts for this hierarchical structure, modeling project-level variation as random effects while estimating job and developer factors as fixed effects. This approach was used to identify which factors are significantly associated with silent failures under the proxy of successful job reruns (RO_4), yielding robust model fit, well-calibrated risk estimates, and enabling variance partitioning between fixed effects (marginal R^2) and the full model (conditional R^2).

3.4.6 Digital Twin Architecture

Digital Twins (DTs) are virtual replicas of physical systems that enable real-time monitoring, simulation, and optimization, traditionally applied to cyber-physical systems such as manufacturing plants and IoT networks (Grieves & Vickers, 2017). This thesis extends the concept of DT to software systems, treating the CI/CD process as the actual counterpart and its digital twin as a continuously synchronized replica that aggregates execution data, tracks pipeline health, and orchestrates automation services. This architecture provides the unifying infrastructure needed to deploy detection, diagnosis, and repair solutions in production (RO_5).

3.5 Summary

This chapter presented the research methodology guiding this thesis. The research integrates empirical software engineering and design science, conducted in an industrial context with replication on open-source projects. Design decisions reflect practical constraints, favoring efficient, scalable, and interpretable approaches. Detailed methodological procedures are presented in the Study Design sections of each subsequent chapter.

CHAPTER 4

EFFICIENT DETECTION OF INTERMITTENT JOB FAILURES USING FEW-SHOT LEARNING

Henri Aïdasso[1] , Francis Bordeleau[1] , Ali Tizghadam[2]

[1] Department of Software and IT Engineering, École de Technologie Supérieure,
1100 Notre-Dame Ouest, Montreal, Quebec, Canada H3C 1K3
[2] Network Softwarisation and AI, TELUS,
Toronto, Ontario, Canada

Article published in « IEEE International Conference on Software Maintenance and Evolution »
September 2025

4.1 Preamble

This chapter addresses the foundational step of the automation chain, consisting of intermittent job failure detection. Before any diagnosis or repair can be automated, the system must first effectively and reliably detect that an anomaly has occurred. Our contribution, published in IEEE ICSME 2025 (Aïdasso, Bordeleau & Tizghadam, 2025b), is complemented in this chapter by a discussion in Section 4.7. This section discusses the practical implications of our proposed approach and introduces SLID[1] (*Small Language models for Intermittent job failures Detection*), the open-source tool developed to operationalize it, accepted at ICSME 2025, Artifact Track.

4.2 Introduction

Detecting intermittent job failures presents several practical challenges, which prior studies have sought to address through different approaches leveraging ML models (Lampel *et al.*, 2021; Olewicki *et al.*, 2022). Such models perform a binary classification of job failures to distinguish *intermittent* from *regular* job failures. In particular, Lampel *et al.* (2021) leveraged at Mozilla, in-house CI telemetry data such as CPU load from over 2 million manually labeled job failures, to build tree-based ML classifiers. However, these infrastructure telemetry data are not typically

[1] https://github.com/ahenrij/efficient-detection-intermittent-job-failures

available in public or shared CI infrastructure (Olewicki *et al.*, 2022). Moreover, the manual labeling of large datasets, as performed at Mozilla, is impractical for most organizations. To tackle these challenges, Olewicki *et al.* (2022) introduced the SOTA approach for intermittent job failure detection at Ubisoft. This approach relies on the similarity in TF-IDF features (Ramos, 2003) of job logs. They also proposed an automated data labeling method based on the heuristic about non-deterministic reruns: if a job is rerun at least once on the same commit ID and changes outcome, it is labeled as intermittent; otherwise, it is considered regular (Olewicki *et al.*, 2022).

While non-deterministic job reruns indicate whether a job failure is intermittent, the associated heuristic does not reliably capture all the intermittent job failures. As outlined in the limitations of the SOTA approach (Olewicki *et al.*, 2022), if a developer omits to rerun an intermittent job failure or does not rerun it enough times for it to become successful, the automated labeling method incorrectly labels it as regular. In fact, this method was employed as an alternative to the manual labeling (previously conducted at Mozilla), based on the key assumption that at Ubisoft, rerunning suspicious job failures is an established policy, which reduces the risk of mislabeling in that context (Olewicki *et al.*, 2022). Consequently, in project contexts where that fundamental assumption about reruns does not necessarily hold (such as at our industrial partner TELUS), there is a high likelihood of mislabeling many intermittent job failures as regular ones. If substantial, such mislabeled instances can significantly undermine the performance of the SOTA intermittent failure detection models.

On the other hand, recent improvements in small language models' performance and efficient fine-tuning methods like few-shot learning (FSL) have minimized the need for large labeled datasets and heavy computing resources (Tunstall *et al.*, 2022). As such, FSL represents an unprecedented opportunity to dispense with the need for potentially erroneous automated labeling of large datasets. Instead, one could manually, thus reliably, label a few job log examples to build performant models for intermittent job failure detection. Yet, to our knowledge, no prior study has (1) evaluated how often the automated labeling method (using the non-deterministic rerun heuristic) mislabels intermittent jobs in different project contexts, or (2) investigated

few-shot classification using small language models, as a more efficient alternative to the SOTA for detecting intermittent job failures.

To fill these gaps, we collected data from 16,786 job failures across 5 main TELUS projects and 1 open-source project, and manually labeled representative samples amounting to 2,125 job failures. Using these datasets, we conduct several FSL experiments to answer the following research questions (RQs):

RQ_1. What are the error rates of the automated labeling method based on non-deterministic job reruns? Results obtained from manual analysis show that an average of 32% of job failures (and up to 50%) are incorrectly labeled as regular failures, when they are actually intermittent. Such high labeling error rates highlight the limitations of the automated labeling method and cast doubt on the effectiveness of the SOTA approach for detecting intermittent job failures.

RQ_2. How effective is our few-shot learning approach for intermittent job failure detection? Our FSL-based approach achieves F1-scores of 70-88% in all studied projects, outperforming the state-of-the-art in five of the six projects. As suspected, the SOTA approach proved ineffective in the four projects with the highest mislabeling rates, with significantly lower F1-scores ranging between 34 and 52%. Its best performance of 89% is only achieved in the project with minimal labeling error, where our FSL approach still achieves 83%, suggesting our approach is overall more robust and effective.

RQ_3. Can a few-shot model for intermittent job failure detection be used on another project? Cross-project predictions using the FSL models also outperform the SOTA approach in four of the six studied projects, suggesting they can be reused in non-critical projects to reduce the number of deployed models. However, in many of the best cases, the cross-project performance remains average (49-70%) compared to a project-specific FSL model, which is recommended.

The main contributions of this study are (1) an empirical evaluation of the existing automated labeling method's accuracy in different industrial and open-source project contexts, and (2) a

novel framework leveraging small language models and few-shot learning for efficient detection of intermittent job failures, achieving F1-scores close to or twice that of the state-of-the-art, using only 12 shots. A replication package (Aïdasso, 2025a) to foster reuse of our framework is made publicly available. This package has been awarded the "Open Research Object" and "Research Object Reviewed" badges at the ICSME 2025 Artifact Evaluation Track.

4.3 Background and Motivation

In principle, build job failures are expected to arise from regular code-related issues, such as compilation errors, legitimate test failures, or code quality problems (Hilton *et al.*, 2016; Hassan, 2019). So, the occurrence of intermittent job failures, typically due to irregular and unexpected issues, misleads developers at TELUS and leads to considerable waste, as also reported at Mozilla (Lampel *et al.*, 2021) and Ubisoft (Olewicki *et al.*, 2022). Because of the inherent non-deterministic nature of intermittent job failures, which also sometimes present very dense and unclear execution logs, developers tend to rerun suspicious job failures in the expectation of a successful outcome (Olewicki *et al.*, 2022; Aïdasso, Bordeleau & Tizghadam, 2025a).

While these multiple job reruns (reaching up to 12 reruns of the same job at TELUS) may ultimately result in success, they constitute a wasteful usage of machine resources and add extra load to already overburdened CI servers (Olewicki *et al.*, 2022; Aïdasso *et al.*, 2025a). In more complex cases, the diagnosis and resolution of intermittent job failures are primarily the responsibility of TELUS' operations and infrastructure teams. However, developers often waste a significant amount of time trying to determine whether a job failure is intermittent, only to find out the problem is beyond their responsibilities, and even their access permissions.

Therefore, there is a strong need to automate the detection of intermittent job failures, particularly in industrial contexts like that of TELUS. Such automation, via ML models for instance, would at least improve teams' productivity by acting as an initial filter, enabling developers to focus on regular job failures, while intermittent issues could be confidently redirected to the teams responsible for diagnosing and resolving them. In addition, these models are expected as in

previous studies (Olewicki *et al.*, 2022; Lampel *et al.*, 2021), to be used as an alternative to costly manual job reruns. Besides, in the broader context of addressing intermittent failures through automation, automated detection constitutes the primary step enabling an intelligent system to determine when intervention (i.e., automated diagnosis and repair) is required.

Olewicki *et al.* (2022) introduced the SOTA approach for the automated detection of intermittent job failures, which consists of a weighted combination of two ML models. The first model is trained using TF-IDF (Ramos, 2003) similarity features extracted from job logs, while the second includes additional metrics such as the number of prior job reruns. Instead of costly manual labeling previously performed at Mozilla (Lampel *et al.*, 2021), Olewicki *et al.* (2022) leveraged the heuristic of non-deterministic job reruns associated with intermittent job failures to automate the labeling process: a job failure is labeled as *intermittent* if it was rerun on the same commit and changed its status to success; otherwise, it is considered *regular*. As a result, the SOTA approach achieved satisfying performances in the context of Ubisoft.

A major limitation of the SOTA approach, as outlined in the study (Olewicki *et al.*, 2022), lies in the fact that its labeling accuracy largely depends on whether developers consistently rerun suspicious job failures, and if so, enough times to exhibit their intermittency. If the rerun practice is not fully and consistently applied, the automated labeling method incorrectly labels the job failure as regular when in fact it is intermittent. One such example from the open-source Veloren project is an intermittent job failure[2] due to an overloaded server, mislabeled as *regular* by the heuristic based on non-deterministic reruns. An identical failure[3] was rerun on another job and passed, enabling the heuristic to assign a correct *intermittent* label. This highlights the inconsistency of labeling based on the heuristic, which relies too strongly on re-execution practice rather than the root cause of failures. If the number of mislabeled intermittent jobs is high, a model based on log TF-IDF similarity is bound to learn confusing patterns (i.e., similar features in different classes) and therefore perform very poorly.

[2] https://gitlab.com/veloren/veloren/-/jobs/2720269150

[3] https://gitlab.com/veloren/veloren/-/jobs/3036439287

Even in contexts where the rerun practice is initially adopted, the automated labeling method exhibits practical limitations when it comes to model retraining in the event of concept drift. Indeed, one of the main purposes of using an intermittent job failure detection model is to minimize job reruns. So, when such a model needs retraining after a certain period of time, there is a high likelihood that recent job failures will not be accurately labeled because of the labeling method's strong reliance on the reruns. Finally, the existing SOTA approach involves computationally expensive steps for model training, which we discuss further in Section 4.5.3.

To address these limitations, we introduce in this chapter a novel efficient approach leveraging few-shot learning to create intermittent job failure detection models using only a limited number of manually labeled log examples.

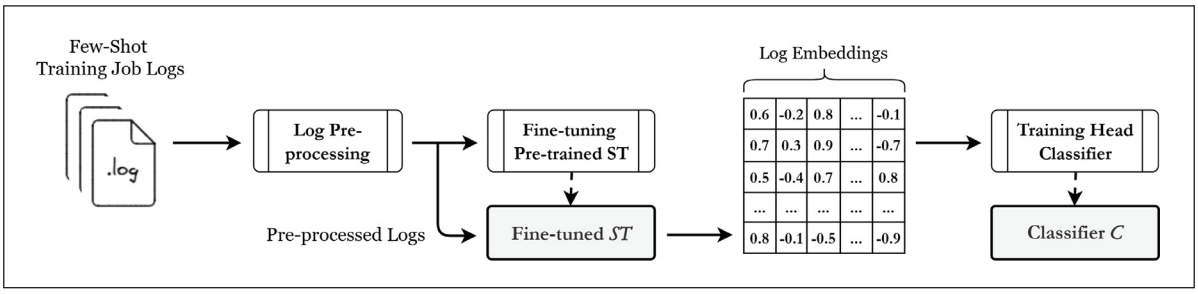


Figure 4.1 Illustration of the two-step model training following the log pre-processing (*LP*) stage, including (1) few-shot fine-tuning a sentence transformer *ST* model used to produce embeddings of the training dataset and (2) training a classification head *C* using embeddings generated by the fine-tuned *ST*

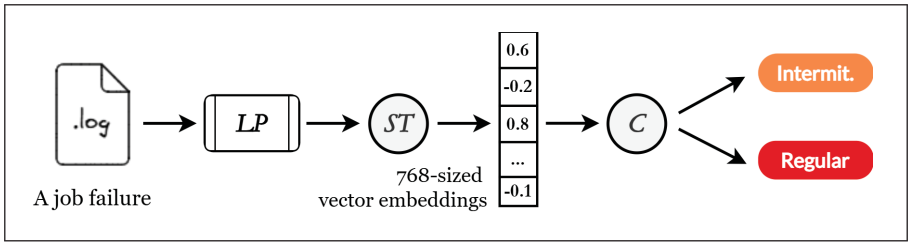


Figure 4.2 Classifying a job failure as intermittent or not using a few-shot fine-tuned sentence transformer *ST* for embeddings generation and a classifier *C*

4.4 Few-Shot Learning Approach for Detecting Intermittent Job Failures

Fig. 4.2 shows an overview of the proposed approach for intermittent job failure detection involving three main stages: (1) log pre-processing (*LP*), (2) embeddings generation using a fine-tuned sentence transformer (*ST*) (Devlin *et al.*, 2019), and (3) prediction using the classifier (*C*). Unlike the SOTA (Olewicki *et al.*, 2022), our approach is designed to be more practical, where practitioners only need to provide the job failure logs for the model to classify it as either *intermittent* or *regular*. In addition, the model training process (illustrated in Fig. 4.1) only requires a few examples of job failure logs from each class (i.e., shots).

To help the model effectively understand log patterns, we first apply a set of log pre-processing rules to reduce noise and minimize log size. For model training, we use SetFit (Tunstall *et al.*, 2022), an efficient few-shot learning framework with a two-step training process including (1) fine-tuning a pre-trained *ST* model to generate rich text embeddings, and (2) training a classification head model (*C*) using the generated embeddings. At prediction time, the predicted class c of a job failure given its log file is determined as $c = C(ST(LP(log)))$. In the following, we present a detailed description of each step of our approach.

4.4.1 Log Pre-processing

Similarly to previous studies (Olewicki *et al.*, 2022; Jiang *et al.*, 2017), we pre-process the job logs to enable effective learning of the failure patterns. To this end, we leverage the log pre-processing (*LP*) rules described below. These rules, commonly used in natural language processing (NLP), are also inspired by prior studies (Olewicki *et al.*, 2022; Brandt *et al.*, 2020). We refined them through manual inspections and iterative trials on a random sample of 100 logs. This ensured that variable elements in the logs (e.g., URLs) were properly abstracted and that noisy elements (e.g., special characters) were cleaned out, while preserving the structure and key information in the logs.

Rule 1 URLs, file paths, directory paths, durations, and versions identified by regexes are respectively replaced by the constant string <URL>, <FILEPATH>, <DIRPATH>, <DURATION>, and <VERSION>.

Rule 2 IDs (series of characters containing at least one letter and one number) are replaced by the string <ID>.

Rule 3 Non letter or numerical characters are removed.

Rule 4 All numbers, except for HTTP status codes and exit codes, are removed.

Rule 5 Trailing single letters are removed.

Rule 6 White spaces and blank lines are removed.

Rule 7 Duplicate lines are removed.

As a result, these rules reduced the size of raw logs by an average of 67%. We train the models using the pre-processed logs. For prediction, the job logs are also first pre-processed using the same *LP* rules before being passed to the *ST*.

4.4.2 Sentence Transformer Few-Shot Fine-tuning

The model training phase involves two steps. The first step consists of fine-tuning a pre-trained *ST* to produce numerical vector representations called *embeddings* that effectively (i.e., with semantic understanding) distinguish textual logs of the different classes. For this purpose, we use `BAAI/bge-small-en-v1.5`[4], the SOTA pre-trained embedding model available on HuggingFace. It maps textual input data to a 768-dimensional dense vector space used for semantic similarity tasks. We chose this language model because it achieves state-of-the-art performance on the Massive Text Embeddings Benchmark (MTEB) leaderboard, while maintaining a small size (33.4M parameters), which makes it well-suited for the scalability needs at TELUS. Indeed, small language models enable easy and cost-effective in-house deployments that preserve data confidentiality, unlike LLMs. Moreover, as our results with this small language model were satisfactory, it was unnecessary to explore LLMs in this study.

[4] https://huggingface.co/BAAI/bge-small-en-v1.5

For *ST* fine-tuning, SetFit leverages a contrastive learning approach (Koch, Zemel & Salakhutdinov, 2015) where the *ST* learns a discriminative embedding space using a dataset of triplets generated from the training set. This dataset includes (1) a set of positive triplets $\{(x_i, x_j, 1)\}$ where the pairs x_i and x_j are logs randomly selected from the same class; and (2) a set of the same size composed of negative triplets $\{(x_i, x_j, 0)\}$ where the pairs x_i and x_j are logs randomly selected from different classes, i.e., one is *intermittent* and the other *regular*. With this dataset, the pre-trained *ST* model adjusts the embeddings using a cosine similarity loss function, so that positive log pairs are positioned closer together in the embedding space. In contrast, negative pairs are pushed far apart. The resulting fine-tuned *ST* model is then used to generate the embeddings of logs in the training set for training the classification head model (see Fig. 4.1). Similarly, the fine-tuned *ST* generates embeddings for each log to be classified at prediction, as shown in Fig. 4.2.

4.4.3 Classification

In the second training step, a binary classifier *C* is trained using the training set's embeddings generated by the fine-tuned *ST* model. To this end, we use the *scikit-learn* implementation of the Logistic Regression[5] (LR) algorithm. This model classifies failed jobs as either *intermittent* (with a predicted value of 1) or *regular* (with a predicted value of 0).

The validation of our FSL-based approach, covering hyperparameter optimization (HPO) and performance evaluation, is discussed in section 4.5.3.

4.5 Study Design

4.5.1 Research Questions

The primary goal of this study is to investigate few-shot learning for intermittent failure detection, as an innovative and more effective approach that leverages a small set of well-labeled log

[5] https://scikit-learn/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

examples, rather than relying on large datasets that are too expensive to annotate manually and are possibly mislabeled using the existing automated method. Hence, we first assess the labeling error of using the non-deterministic job rerun heuristic for automated labeling (compared to manual labeling) in different project contexts (RQ_1). We then evaluate the performance of our FSL-based approach in detecting intermittent job failures, compared to the state-of-the-art approach that relies on large datasets and automated labeling (RQ_2). Finally, we investigate the performance of our FSL-based approach in cross-project predictions (RQ_3). For this purpose, we design the following research questions (RQs):

RQ_1. What are the error rates of the automated labeling method based on non-deterministic job reruns?

RQ_2. How effective is our few-shot learning approach for intermittent job failure detection?

RQ_3. Can a few-shot model for intermittent job failure detection be used on another project?

4.5.2 Studied Projects and Data Preparation

We conducted this study using build data extracted from five major projects at TELUS and one open-source software (OSS) project. We focused on these five industrial projects because the study requires tedious manual labeling and consultation with engineers working on the projects. In addition, we studied the popular open-source role-playing game (RPG) project `veloren/veloren`.[6] We specifically selected this project because the SOTA approach was mainly evaluated on game projects, where intermittent job failures might be associated with the heterogeneous nature of the codebase (i.e., containing not only source code but also audio, scene, animation, and data files (Aleem, Capretz & Ahmed, 2016)). In summary, the studied projects are presented in Table 4.1, with the five industrial projects anonymized for confidentiality reasons. These projects vary in size, maturity, and purpose and encompass 5 programming languages.

We collected job metadata and logs from the projects' build history using the GitLab REST APIs[7]. The collected build jobs data span between 1.2 and 4.9 years of build history across the

[6] https://gitlab.com/veloren/veloren

[7] https://docs.gitlab.com/api/jobs/

Table 4.1 Overview of the studied projects and collected data for the detection study

Project	main languages	# commits	# years	# jobs	# failed jobs	BFR	S*
A	TS, Py	1.9k	4.9	7k	1,597	21%	366
B	TS, Py, JS	0.7k	3.1	3k	1,575	53%	346
C	Py	1.7k	3.8	3k	1,820	10%	347
D	Py, Shell	1.6k	1.3	8k	2,531	14%	369
E	Shell	2.9k	1.2	4k	2,328	19%	331
OSS	Rust	5.3k	2.5	57k	6,935	22%	366

*sample size

studied projects. From these jobs, we have only kept completed ones (i.e., with *success* and *failed* statuses) since other non-completed jobs (e.g., *canceled*) are not relevant for the present study. We obtained a total of 83,214 jobs, including 16,786 failed jobs, split across the projects as indicated in Table 4.1.

Automated Labeling. We labeled intermittent job failures using the non-deterministic job rerun heuristic proposed by Olewicki *et al.* (2022). To this end, we proceeded for each project as described in the following. First, we grouped the jobs by *name* and *commit* to identify the job rerun groups, i.e., groups containing at least two executions (or runs) of the job on the same commit. Next, we selected rerun groups where the job outcome changed between *failure* and *success*. We assigned the label *intermittent* to the failed jobs within these rerun groups. Olewicki *et al.* (2022) refer to these intermittent job failures as *brown job failures*, emphasizing their non-deterministic nature of shifting from red failure to green success. We then calculated the Brown Failure Ratio ($BFR = \frac{\#\ brown\ job\ failures}{\#\ total\ job\ failures}$) in each project, and the obtained BFR values are reported in Table 4.1.

The studied projects present varying BFR values, ranging from 10% in project C to 53% in project B. An average *BFR* of 23% (i.e., in 5 job failures, 1 to 2 are intermittent) across the five TELUS projects, highlights the prevalence of intermittent job failures as a major challenge in these projects. Such a high average BFR also underlines the relevance of focusing on these projects for the present study. Besides, we observe that the BFR of Veloren (22%) is close to

the mean BFR (23%) of the other studied projects, which suggests its oracle is representative. We expect the ratios of intermittent job failures to be even higher, as we hypothesize that the non-deterministic rerun method mislabels some of them as regular ones.

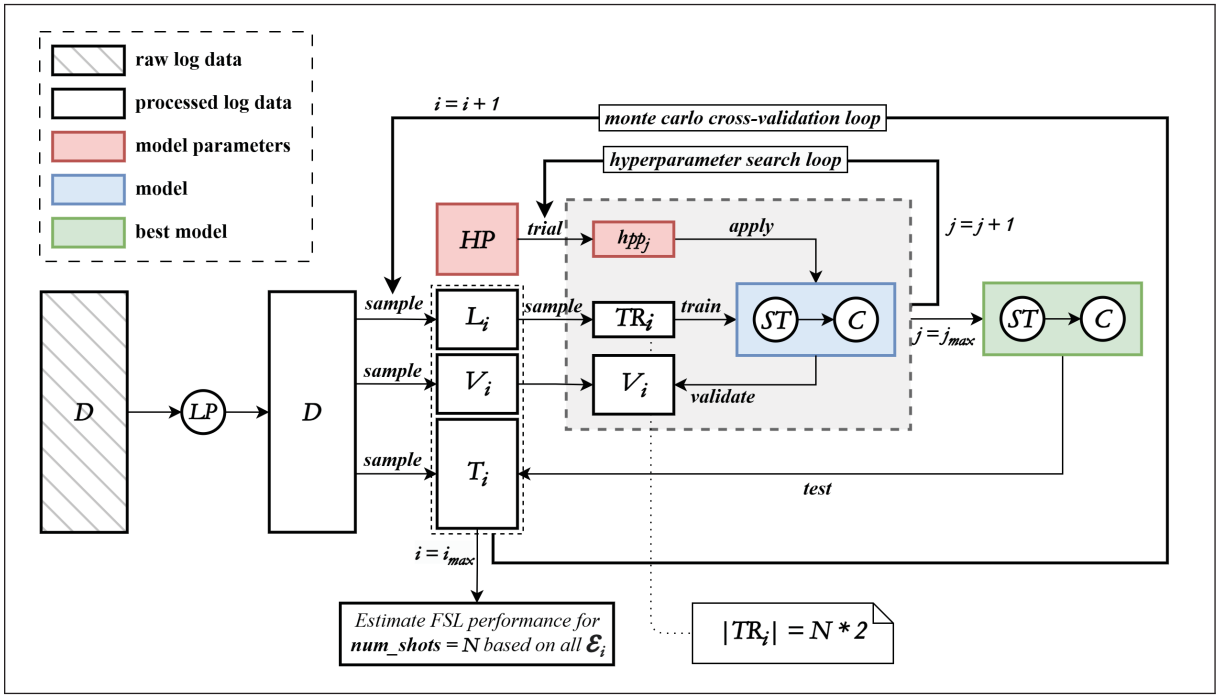


Figure 4.3 Overview of the experiment for evaluating our FSL-based approach for intermittent job failure detection using a number N of shots, i.e., log examples per class. D is the manually labeled dataset of logs, and LP depicts the log pre-processing step. L_i, V_i, T_i, and TR_i, are the i^{th} learning, validation, test, and training set, respectively. ε_i is the model performance estimation on T_i. HP represents the hyperparameter space, and hpp_j is the set of hyperparameters for the j^{th} loop of hyperparameter search. ST represents the sentence transformer that generates log embeddings, while C denotes the classifier

Dataset Sampling. To address the RQs, this study requires manual labeling of failed jobs. Given the impracticality of manually labeling thousands of jobs, we focus on statistically representative samples, following prior work (Aïdasso *et al.*, 2025a; Ghaleb *et al.*, 2021). For each project, we determined the sample size using a confidence level of 95% and a 5% margin of error. The obtained sample sizes are indicated in Table 4.1. To maintain the representativeness of the original population, we applied a stratified random sampling so that each sample dataset preserves the same BFR as its corresponding original set of failed jobs. Finally, we kept the job

logs and labels assigned from automated labeling based on the non-deterministic rerun heuristic. In total, we obtained the execution logs and automated labels of 2,125 failed jobs.

Manual Labeling. The automated labeling method offers the advantage of being inherently reliable when assigning *intermittent* labels. Its potential for mislabeling lies exclusively with job failures that are labeled as *regular* simply because they have not been rerun or have not changed status following reruns. Therefore, our manual labeling process focuses on the job failures initially labeled as *regular*. The manual labeling process of these jobs for each project is described below.

For each job failure initially labeled as *regular* in the sample dataset, we manually analyzed the execution logs to identify the job failure reason using Stack Overflow[8] (SO). When the failure reason is suspected to be an intermittent issue, we identify the log chunks that best describe the root cause. We then create regular expressions (regex) to match highly similar logs in the ground truth set of intermittent job logs identified with the automated labeling. In some cases, these regexes enabled us to identify similar logs, allowing us to conclude the intermittent nature of the failure initially labeled as *regular*.

However, we noticed that regexes missed several matches due to their rigid nature (i.e., lacking semantics understanding, and therefore failing to associate `insufficient space` with `out-of-memory` errors, for example). We also observed that many regular failure logs in our industrial projects show symptoms of intermittent issues (i.e., include similar log traces), without failing because of them, which limits the effectiveness of the regexes. Furthermore, we identified a total of 35 regexes during our investigation, which not only fail to reliably match intermittent job failures, but are highly prone to human errors during maintenance.

For these reasons, we ruled out the use of regexes and relied instead on TELUS engineers' expertise to complete the manual labeling. The failure reasons identified on SO and the associated failure logs have been discussed through weekly meetings with experienced TELUS engineers actively involved in the studied projects and working in different teams (i.e., software

[8] https://stackoverflow.com/

development, networking, and infrastructure teams). When a failure reason is finally determined by consensus to be intermittent, the associated job failures (initially labeled as *regular*) are manually relabeled as *intermittent*; otherwise, they are considered to be truly *regular*.

We saved for each manually labeled job failure, the initial automated label, the manual label, and the justification for relabeling. This includes, in some cases, the regex and IDs of matching logs from the ground truth set of intermittent job failures, and in other cases, the corresponding intermittent failure category as identified in a recent study (Aïdasso *et al.*, 2025a). For confidentiality reasons, only data from the OSS project are made publicly available in our replication package (Aïdasso, 2025a).

Summary of Datasets. To answer our RQs, we use two datasets per project: (1) the manually labeled samples totaling 2,125 failed jobs with their automated labels, and (2) the remaining jobs totaling 14,661 failed jobs for all projects and labeled only using the automated method.

4.5.3 Validation of the FSL-based Approach

Fig. 4.3 illustrates our experimental setup for evaluating the proposed FSL-based approach. The manually labeled dataset D is randomly split into learning (L), validation (V), and test (T) sets. The split is stratified so that each set contains the same ratio of intermittent job failures. Similarly to Tunstall *et al.* (2022), the learning set is used as the source from which the actual training set (TR) is selected using a random sampling. The resulting training set consists of an equal number N of shots per class, i.e., N regular failure logs and N intermittent failure logs. Finally, the validation set V is used for HPO, while the test set T is used for the model's performance estimation. Algorithm 4.1 shows the detailed steps of our evaluation method.

Hyperparameter Optimization. We identified several hyperparameters to tune during the SetFit training steps described in Section 4.4.2 and 4.4.3. These hyperparameters were selected based on recommendations from the official documentation[9]. In particular, we opted for smaller

[9] https://huggingface.co/docs/setfit/

Algorithm 4.1 Few-Shot Learning Evaluation Using Monte Carlo Cross-Validation (MCCV)

Input: Dataset D, iterations $i_{\max} = 100$, hyperparameter space HP, trials $j_{\max} = 5$, shots N
Output: Estimated average performance $\bar{\varepsilon}$

Initialize performance list: $\mathcal{E} \leftarrow []$;
for $i \leftarrow 1$ **to** $i_{\max}$ **do**
  $L_i, V_i, T_i \leftarrow$ `Split(`D`)`;
  $TR_i \leftarrow$ `Sample(`L_i, N`)` ; `// Sample` N `examples per class from` L_i
  best_model $\leftarrow \emptyset$, best_score $\leftarrow -\infty$;
  for $j \leftarrow 1$ **to** $j_{\max}$ **do**
    $hpp_j \leftarrow$ `Sample(`HP`)`;
    $M_{ij} \leftarrow$ `Train(`TR_i, hpp_j`)`;
    $\varepsilon_{val}^{(ij)} \leftarrow$ `Evaluate(`M_{ij}, V_i`)`;
    if $\varepsilon_{val}^{(ij)} >$ *best_score* **then**
      best_score $\leftarrow \varepsilon_{val}^{(ij)}$;
      best_model $\leftarrow M_{ij}$;
    end if
  end for
  $\varepsilon_i \leftarrow$ `Evaluate(`*best_model*, T_i`)`;
  Append ε_i to $\mathcal{E}$;
end for
$\bar{\varepsilon} \leftarrow \frac{1}{i_{\max}} \sum_{i=1}^{i_{\max}} \varepsilon_i$;
return $\bar{\varepsilon}$

batch sizes (due to limited memory resources) and lower learning rate ranges following recent guidelines for effective fine-tuning of small language models (Pareja *et al.*, 2024). Hence, we define the hyperparameters search space *HP* as follows:

- *Body Learning Rate*: Learning rate of the *ST* model. (Range: $1e^{-6}$ to $1e^{-3}$)
- *Num Epochs*: Number of training epochs for the *ST*. (Choices: 1, 2)
- *Batch Size*: Batch size for training the *ST* model. (Choices: 2, 4, 8)
- *Max Iter*: Maximum number of iterations for the solvers to converge during *C* training. (Range: 50 to 300, by 50)

In this study, we run $j_{max} = 5$ trials for hyperparameter search. During each trial, the model (composed of the *ST* and the classifier *C*) is trained using a different set of hyperparameters

randomly initialized from the defined *HP* space. After training, each model is evaluated on the validation set V_i. At the end of the five trials, the model with the highest performance (i.e., the *best model* out of five) is selected, and its generalization performance is estimated on the test set.

Cross-validation Settings. We use the Monte Carlo Cross-Validation (MCCV) (Simon, 2007), also known as repeated random subsampling cross-validation, to estimate the generalization performance of the models for a given number N of shots. This validation technique minimizes the performance estimation error associated with the random selection of the $N * 2$ examples forming the training set. Furthermore, MCCV provides more reliable performance estimates than traditional K-fold CV in binary classification with limited datasets (Shan, 2022).

As depicted in Fig. 4.3, MCCV consists of repeating model training and performance estimation i_{max} times, each time using different stratified random splits for the learning, validation, and test sets. During our experiments on a 16GB GPU, we found each model training to be fast, from just a few minutes for the smallest number of shots to about 40 minutes for the larger ones. Consequently, we set the number of repeats for MCCV to $i_{max} = 100$ for a given number of shots. In each iteration i, the dataset D is split as follows:

- Learning set (L_i): 25% of the dataset D
- Validation set (V_i): 25% of the dataset D
- Test set (T_i): 50% of the dataset D

We choose large split sizes for the validation and test sets since (1) we are using limited (manually labeled) sample datasets and (2) our FSL-based approach only requires a few examples for training. Given the sample sizes as indicated in Table 4.1, the size of learning sets across the projects ranges between 83 and 92, which is large enough for FSL. In fact, the actual training set TR_i is created by randomly selecting N intermittent and N regular job failures from the learning set L_i. Finally, the validation set V_i is used to optimize the training parameters of the *ST* and Logistic Regression C models, while the test set T_i is used to estimate the performance ε_i of the best model obtained at the i^{th} iteration.

MCCV is performed once for a given number N of shots and a project. Each cross-validation involves 500 model training (100 repeats and 5 trials per repeat for HPO). As only the best model is retained after HPO, the final performance $\bar{\varepsilon}$ for cross-validation is determined by averaging the 100 ε_i performance values obtained at the end of each iteration.

Performance Metrics. To evaluate the performance of our models, we used the common evaluation metrics, including precision ($\frac{TP}{TP+FP}$), recall ($\frac{TP}{TP+FN}$), and F1-score ($\frac{2}{pre^{-1}+rec^{-1}}$) similarly to prior studies (Olewicki *et al.*, 2022; Lampel *et al.*, 2021). We report precision and recall scores for each of the *intermittent* and *regular* classes. Nonetheless, the F1-score is used for HPO and for analysis and comparisons with the SOTA baseline.

Baseline. We use the SOTA approach for intermittent job failure detection proposed by Olewicki et al. (Olewicki *et al.*, 2022) as a baseline. For model training, this approach relies on the larger datasets automatically labeled. We removed the manually labeled samples from the datasets, as they are later used for model evaluation. The SOTA model creation involves the following main stages, more extensively described in the original paper. First, we perform log pre-processing and compute the additional required job metrics, including the `number of prior reruns` and the `number of commits since` the last intermittent job failure. In the second stage, we compute the TF-IDF vectors of the processed logs and perform feature selection to reduce the vectors' dimensionality (using the SelectKBest[10] algorithm from *scikit-learn*). The SOTA detector comprises two eXtreme Gradient Boosting[11] (XGBoost) models: (1) the first model is trained using the best TF-IDF features exclusively, and (2) the second model uses the job metrics and the Shap[12] values resulting from the first model training. The final prediction is made through a weighted vote of the two models. In line with the original study, we use a 10-fold cross-validation for hyperparameter optimization. The model with the best hyperparameters is evaluated on the same test sets T_i as the corresponding FSL model for comparison.

[10] https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html

[11] https://xgboost.readthedocs.io

[12] https://shap.readthedocs.io

4.6 Results

In this section, we present and discuss the results of our research questions (RQs).

4.6.1 RQ_1. What are the error rates of the automated labeling method based on non-deterministic job reruns

4.6.1.1 Motivation

The goal of this RQ is to evaluate the error rate of the existing automated labeling method based on the non-deterministic job rerun heuristic. Such a finding will help assess the accuracy of the automated labeling method in project contexts where there is no explicit policy for rerunning suspicious job failures. If the automated method proves inaccurate, the results will serve as empirical evidence that further motivates the need for our FSL-based approach for intermittent job detection, since manually labeling large datasets is costly.

4.6.1.2 Approach

To answer this RQ, we analyze the manually labeled dataset of 2,125 job failure logs composed of representative samples across the six studied projects. These logs are labeled using (1) the automated method relying on non-deterministic reruns (Olewicki *et al.*, 2022) and (2) the manual labeling approach detailed in Section 4.5.2 and inspired by the study at Mozilla (Lampel *et al.*, 2021). The assigned labels are integer values: 1 for *intermittent* and 0 for *regular*.

Let a_i and m_i be the assigned automated and manual labels, respectively, to the i^{th} job in a given project; and n the total number of jobs in this project. We calculate the automated labeling error rate E in the project as follows:

$$E = \frac{\sum_{i=1}^{n} \mid a_i - m_i \mid}{n} \tag{4.1}$$

The numerator part of Eq. 4.1 counts mislabeled jobs, i.e., jobs for which $a_i \neq m_i \implies | a_i - m_i | = 1$. In addition to the error rate per project, we include in our replication package (Aïdasso, 2025a) statistics about the identified failure reasons (see approach in Section 4.5.2) of the intermittent job failures mislabeled as regular by the automated labeling method.

4.6.1.3 Results

Table 4.2 Statistics and Error rates of automated labeling compared to manual labeling across the studied projects

Project	Automated		Manual		Error Rate (E)
	Intermittent	Regular	Intermittent	Regular	
A	77	289	175	191	26.78%
B	184	162	240	106	16.18%
C	36	311	173	174	39.48%
D	52	317	237	132	50.14%
E	64	267	206	125	42.90%
OS	81	285	141	225	16.39%

On average, one-third (32%) of job failures (and up to 50%) are incorrectly labeled as regular by the automated labeling method, when they are actually intermittent. Such high ratios of mislabeled job failures exhibit the limitations of this method and cast doubt on the generalisability of the SOTA detection approach. Table 4.2 summarizes the automated and manual labeling statistics for each of the six studied projects, along with the calculated error rates. The results show that the mislabeling rates (E) vary from one project to another. In the best case, the labeling error rate is at 16.18% in project B, meaning that roughly 1 to 2 jobs in 10 is incorrectly labeled in that project. In the worst case, the ratio of mislabeled jobs rises to more than half (50.14%) of the jobs in project D. The labeling error rates for projects A, C, and E are 26.78%, 39.48%, and 42.90%, respectively, resulting in an average error rate of 35.09% across the five industrial projects. In contrast, the labeling error is at 16.39% in Veloren, which is nonetheless quite considerable. Overall, the average labeling error is 31.97% (i.e., about one-third of the jobs are mislabeled) across the six studied projects.

4.6.2 RQ_2. How effective is our few-shot learning approach for intermittent job failure detection

4.6.2.1 Motivation

In this second RQ, we investigate whether our few-shot learning approach can effectively detect intermittent job failures with minimal manual labeling, addressing the limitations of the SOTA (Olewicki *et al.*, 2022) that relies on error-prone automated labeling, as highlighted in RQ1. In particular, we aim to (1) determine the optimal (i.e., smallest) number of shots required to achieve satisfying performance and to (2) evaluate the relevance in practice of our approach compared to the SOTA for intermittent job failure detection. The findings will guide practitioners in the choice of the optimal number of manually labeled log examples necessary to achieve good and reliable performance, enabling them to apply this FSL-based approach on a larger scale with minimal effort. In addition, the results will offer insights into the effectiveness of our FSL-based approach compared to the SOTA, particularly in projects with no explicit policy for rerunning suspicious job failures.

4.6.2.2 Approach

We evaluate our FSL-based approach for a given number N of shots using the MCCV technique described in Section 4.5.3. We investigate different numbers N of shots ranging from 1 to 15. Indeed, above 15 shots, we deviate from the concept of FSL (Cao, Law & Fidler, 2020). This maximum of 15 shots (i.e., 30 jobs) for manual labeling has also been discussed with TELUS engineers, who consider it tedious but acceptable. We then conduct the evaluation experiment described in Fig. 4.3, using the manually labeled sample datasets for each project and each number of shots. In total, 45,000 models are trained ($5_{trials} * 100_{repeats} * 15_{shots} * 6_{projects}$), and the resulting performance evaluation measures are saved for analysis. Specifically, we analyze the performance evolution of our FSL-based approach as the number of shots is increased, and discuss the choice of the number of shots. Furthermore, we compare for each project, the performance of the optimal FSL model with that of the SOTA model described in Section 4.5.3.

In line with the original paper (Olewicki *et al.*, 2022), we train and optimize hyperparameters of the SOTA model using the large and automatically labeled datasets. These training sets are deprived of the manually labeled test set splits T to avoid set contamination. Hence, the SOTA and our FSL-based approaches are evaluated using the same manually labeled test sets in each project, providing comparable and reliable estimates of their performance in practice.

4.6.2.3 Results

Fig. 4.4 shows, for each project, the evolution of the F1-score as the number of shots is increased from 1 to 15. It also depicts the standard deviation of F1-scores, using a shaded error band surrounding each line plot. Table 4.3 presents the performance obtained with 12 and 15 shots, as well as that of the baseline SOTA model trained on full datasets with automated labels. For each project, we highlight in **bold** the best model performance and underline the second best one.

Our models achieve F1-scores above 70% in all the studied projects (70-88%), with just 12 shots, demonstrating consistent effectiveness with limited training data. Beyond 12 shots, the performance improvement is no longer statistically significant (p-values > 0.05). As can be observed in Fig. 4.4, the model performance tends to improve with a greater number of shots in all projects. However, it exhibits higher variability below 10 shots (i.e., larger error bands from 1 to 9 shots). Specifically, average F1-scores decrease in project C between 7 and 8 shots, in B between 5 and 6 shots, and in the OS project between 6 and 8 shots. Starting from 10 shots, the average performance becomes quite stable and satisfying, with F1-scores at 10 shots ranging between 69% and 88% across all the studied projects. With 12 shots, all FSL models achieve average F1-scores exceeding 70%, specifically 81%, 83%, 70%, 88%, 80%, and 82% in projects A, B, C, D, E, and OS, respectively. Performance further improves and peaks at 15 shots across all projects in our experiments. As reported in Table 4.3, the F1-scores at 15 shots remain very close to those at 12 shots, with no to minimal increase of 1% in projects A, B, and C.

We investigated whether the small improvement obtained with 15 shots is statistically significant compared with 10 to 14 shots, as we observed a convergence of F1-scores from 10 shots onwards

Table 4.3 Performance of FSL models trained using $N = 12$ and 15 examples per class

Project	*N* = 12					*N* = 15					SOTA (*N* = *Full*)				
	F1	Intermit.		Regular		F1	Intermit.		Regular		F1	Intermit.		Regular	
		Pre	Rec	Pre	Rec		Pre	Rec	Pre	Rec		Pre	Rec	Pre	Rec
A	81_{5}	77_{8}	87_{8}	87_{6}	75_{12}	**82_{4}**	78_{8}	88_{7}	88_{6}	76_{12}	52	37	90	96	62
B	83_{12}	90_{4}	79_{19}	69_{19}	78_{11}	84_{10}	90_{4}	82_{17}	72_{17}	77_{10}	**89**	86	91	82	72
C	70_{6}	71_{4}	71_{10}	71_{6}	71_{6}	**71_{6}**	71_{4}	72_{10}	72_{6}	71_{6}	34	23	72	91	54
D	**88_{3}**	94_{3}	83_{6}	76_{6}	89_{6}	**88_{3}**	94_{2}	83_{6}	75_{6}	90_{5}	38	24	97	98	42
E	**80_{2}**	90_{5}	73_{5}	66_{3}	86_{8}	80_{3}	91_{4}	73_{5}	66_{4}	87_{8}	48	33	92	96	47
OS	**82_{5}**	91_{4}	75_{10}	86_{5}	95_{3}	**82_{5}**	91_{3}	75_{9}	86_{4}	95_{2}	79	69	95	98	83
Median	**82_{5}**	90_{4}	77_{9}	74_{6}	82_{7}	**82_{5}**	91_{4}	79_{8}	74_{6}	82_{7}	50	35	92	96	58
Mean	81_{6}	86_{5}	78_{10}	76_{8}	82_{8}	**81_{5}**	86_{4}	79_{9}	77_{7}	83_{7}	57	45	90	94	60

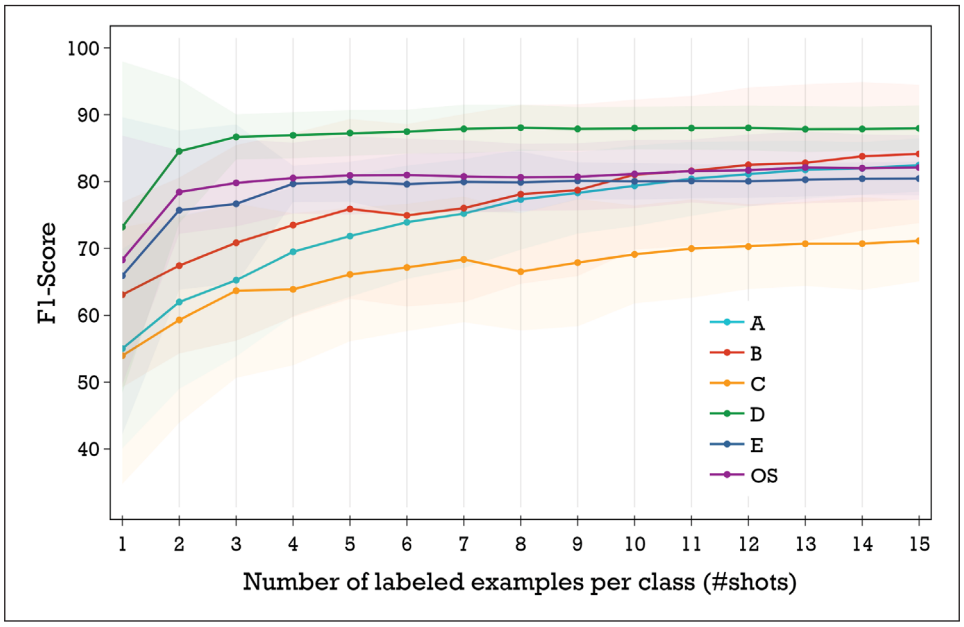


Figure 4.4 Performance of FSL models versus number of manually labeled examples for intermittent job failures detection within studied projects

in Fig. 4.4. For this purpose, we use the non-parametric Wilcoxon rank-sum test (also referred to as the Mann-Whitney U test[13]) to compare the distribution of F1-scores for models trained using 15 shots, with that of models trained using 10, 11, 12, 13, and 14 shots. We obtained respective p-values of 0.003, 0.03, 0.16, 0.45, and 0.78. For 10 and 11 shots, the performance difference is statistically significant with p-values < 0.05 (i.e., 5% margin of error and confidence level of

[13] https://en.wikipedia.org/wiki/Mann-Whitney_U_test

95%). In contrast, starting from 12 shots, all p-values exceed 0.05, indicating no statistically significant difference in performance when training the model with 12 to 14 shots compared to 15. Consequently, we selected 12-shot trained models for all projects, as they achieved the best compromise between performance and number of shots.

FSL models outperform the SOTA models in five of the six projects, with 70-88% F1-scores versus 34-52% in four projects. Even in the one exception project, with the lowest labeling error rate, the performance gap is pretty small (83% vs. 89%), despite our models only requiring 12 shots. As reported in Table 4.3, the SOTA models perform very poorly in projects A, C, D, and E, with respective F1-scores of 52%, 34%, 38%, 48%. Interestingly, these projects exhibit the highest mislabeling error rates, all above 25% (respectively 26.78%, 39.48%, 50.14%, and 42.90%) as shown in Table 4.2. Consequently, the SOTA approach for detecting intermittent job failures is not even viable for practical use in these projects. It only achieves acceptable performance in projects B and OS (89% and 79% F1-scores, respectively), where we observe the lowest mislabeling error rates (16.18% and 16.39%, respectively). In contrast, our 12-shot models achieve satisfying performance (70-88% F1-scores) across all the studied projects, outperforming the SOTA in five projects.

4.6.3 RQ_3. Can a few-shot model for intermittent job failure detection be used on another project

4.6.3.1 Motivation

This RQ investigates whether an FSL model trained to detect intermittent job failures in one project can be reused in another project. Such *cross-project* models can help practitioners, especially at TELUS with a plethora of active projects, reduce the cost of manual data labeling at scale and avoid deploying a separate model for each project.

4.6.3.2 Approach

Similarly to RQ2, we evaluate our FSL-based approach using MCCV. For this RQ, we use $N = 12$ shots as the optimal number of shots identified in RQ2. In line with prior study (Olewicki *et al.*, 2022), we follow a one-to-one approach for cross-project predictions, where a model trained on a project is evaluated on each of the other studied projects.

4.6.3.3 Results

Table 4.4 Performance of FSL models for cross-project predictions

Pred \ Train	A	B	C	D	E	OS	⋆
A	**81_{5}**	$\underline{66_{9}}$	47_{11}	34_{6}	39_{6}	50_{11}	52
B	70_{9}	$\underline{83_{12}}$	36_{18}	48_{17}	18_{7}	55_{3}	**89**
C	53_{11}	$\underline{62_{9}}$	**70_{6}**	44_{7}	34_{3}	49_{8}	34
D	46_{15}	33_{9}	25_{13}	**88_{3}**	20_{2}	$\underline{56_{12}}$	38
E	$\underline{49_{17}}$	30_{6}	23_{5}	30_{12}	**80_{2}**	41_{10}	48
OS	42_{20}	51_{22}	62_{16}	48_{16}	49_{8}	**82_{5}**	$\underline{79}$

⋆ Baseline SOTA

Cross-project predictions using FSL models achieve F1-scores between 18% and 70%, outperforming the baseline SOTA models in 4 out of the 6 studied projects. While promising, cross-project performance is still poor to average in many cases and lower than that of project-specific FSL models, which remain recommended. Table 4.4 presents the performance of cross-project predictions. For each project (in line), we report the F1-scores of predictions using models trained on data from the projects in the columns. As underlined in this table, cross-project predictions achieve the second-best F1-scores in projects A, C, D, and E (66%, 62%, 56%, and 49%, respectively), up to two times higher than the SOTA baselines in these projects. Some cross-project combinations are particularly promising. For instance, predictions in project B using project A achieve 70% F1-score.

However, cross-project performance remains poor in many cases, falling to 18% when predicting project B using project E. Besides, their predictions are far more variable than those of

project-specific models, with standard deviation values reaching up to 22, versus 6 being the highest observed with project-specific models. In conclusion, provided the right job examples, cross-project predictions could be used in some non-critical projects. However, specific models are recommended for better and more reliable performance.

4.7 Discussion

4.7.1 SLID: A Tool for Creating Intermittent Job Failure Detectors

To operationalize the few-shot detection approach presented in this study, we developed SLID[1] (*Small Language models for Intermittent job failures Detection*), an open-source tool for creating and evaluating intermittent job failure detection models (or detectors). The SLID artifact was awarded the "Open Research Object" and "Research Object Reviewed" badges at the ICSME 2025 Artifact Evaluation Track.

Given a set of 12 pairs of manually labeled job logs (i.e., regular vs. intermittent failure logs), SLID fine-tunes a pre-trained sentence transformer and trains a classification head to produce a project-specific binary classification model: the detector. This model can then be called at inference time to classify new job failures as intermittent or regular.

4.7.2 Practical Implications

Practitioners can build and maintain effective detectors with minimal labeling effort, even in cold-start or low-resource scenarios. Unlike the SOTA, which requires large volumes of historical job data and depends on existing rerun practices for automated labeling, SLID requires only 12 manually labeled log pairs per class to train an effective detector. This makes it suitable not only for cold-start scenarios, such as newly created projects or organizations adopting CI/CD monitoring for the first time, but also for projects where intermittent failures are rare yet critical, and where accumulating a large labeled dataset would take an impractical amount of time. Moreover, unlike the SOTA, whose maintenance creates a circular dependency (the labeling

mechanism that sustains detector performance requires the very reruns that detection aims to reduce), SLID models can be periodically retrained using a small set of freshly labeled logs, independently of any rerun policy. This property is essential and presents a key advantage over the SOTA, as it avoids contradicting the broader goal of reducing wasteful manual job reruns.

SLID models can be easily integrated into CI/CD pipelines to provide developers with quick feedback on the nature of job failures, reducing costs. In practice, a detector can be deployed behind an API endpoint called in a post-failure execution step, or as part of an external service triggered via a webhook on job status changes.[14] In both cases, developers receive a near-real-time classification that helps them confirm or dismiss their suspicion of an intermittent failure, reducing sporadic reruns driven by uncertainty. This is particularly valuable for developers who are new to a project or unfamiliar with its environment and failure patterns, as it prevents them from wasting time investigating code changes that are not at fault.

4.7.3 Limitation and Future Directions

Using project-specific models becomes impractical for organizations managing hundreds or thousands of projects. Despite its advantages, the project-specific nature of SLID presents scalability challenges for large organizations. Our cross-project evaluation (RQ3) showed that models trained on one project exhibit highly variable performance when applied to others (18-70% F1-score), making project-specific models the recommended configuration. However, this implies that an organization with n projects must manually label $24 \times n$ logs (12 per class for each project) and maintain n separate classifiers in production, which becomes impractical at the scale of hundreds or thousands of projects, common in large enterprises.

We invite future research to explore training a single reusable detector on logs from diverse projects as an alternative to maintaining multiple project-specific classifiers. Specifically, fine-tuning medium to large-sized language models using a moderately larger but still practical training set (e.g., ≈100 labeled logs per class drawn from diverse projects) could enable a

[14] https://docs.gitlab.com/user/project/integrations/webhook_events/#job-events

single model to serve multiple projects. Such an approach would significantly reduce both the overall labeling effort and the deployment complexity associated with maintaining too many project-specific classifiers.

4.8 Threats to Validity

Construct Validity. The main threat to construct validity concerns the correctness of the manual labeling. To mitigate this threat, we define a ground truth set of intermittent job logs determined using the non-deterministic rerun heuristic (i.e., logs of job failures that were rerun at least once on the same commit and transitioned to success). With this ground truth as a reference and input from TELUS engineers, we relabel job failures as intermittent only when their failure reasons are confirmed in consensus after discussions during weekly meetings. Our replication package (Aïdasso, 2025a) contains (for the OS project) evidence collected during the manual labeling, including similar intermittent jobs in the ground truth set, Stack Overflow resource links and the type of intermittent job failure. These artifacts are being used for further studies on intermittent job failures at TELUS, such as (Aïdasso *et al.*, 2025a).

External Validity. External validity pertains to the generalizability of our findings. Although we conducted this study on industrial projects at TELUS, these projects vary in size, maturity, purpose, programming languages, and intermittent failure ratios. In addition, we studied a popular OS project (Veloren) to support the replication of this study. Unlike the SOTA (Olewicki *et al.*, 2022), our FSL-based approach does not rely on any assumption (i.e., existing policy of rerunning suspicious job failures). It only requires a few (12) manually log examples per class to achieve a satisfying performance in all the studied projects. Therefore, our approach is expected to generalize to projects with a reasonable log verbosity, regardless of any a priori policy for rerunning jobs. Furthermore, we provide a replication package (Aïdasso, 2025a) to foster the reuse of our approach.

4.9 Conclusion

To cut the waste associated with intermittent job failures, prior research developed ML techniques for their detection. However, these techniques require large datasets of job logs and therefore rely on automated labeling based on too strong assumptions about job reruns. In this study, we found that the automated labeling method mislabels on average 32% of intermittent job failures as regular. Such high mislabeling rates undermine the performance of the SOTA, with F1-scores capping at 34-52% in projects with more than 25% mislabeled instances. Hence, we introduced a novel approach for intermittent job failure detection that requires only a few examples of manually labeled job logs per class (i.e., shots) for model training. Our FSL-based approach achieves 70-88% F1-scores with just 12 shots, outperforming the SOTA in all projects except one, where it still achieves 83% F1-score versus 89% for the SOTA. As such, it demonstrates greater effectiveness in practice while requiring only a limited amount of training data. Finally, cross-project predictions appear promising, reaching up to twofold improvement over the SOTA in projects with the highest mislabeling rates. However, these predictions are highly variable and remain average (49-70%), making project-specific FSL models the recommended choice. As future work, it would be valuable to evaluate how the selection of training examples influences model performance and to investigate fine-tuning a single model on logs from multiple projects to enable cross-project prediction. Finally, the next step of this research work is exploring the detection of specific failure categories (i.e., multi-class classification), which will support teams in the automated diagnosis and repair of intermittent job failures.

CHAPTER 5

ON THE DIAGNOSIS OF FLAKY JOB FAILURES: UNDERSTANDING AND PRIORITIZING FAILURE CATEGORIES

Henri Aïdasso[1] , Francis Bordeleau[1] , Ali Tizghadam[2]

[1] Department of Software and IT Engineering, École de Technologie Supérieure,
1100 Notre-Dame Ouest, Montreal, Quebec, Canada H3C 1K3
[2] Network Softwarisation and AI, TELUS,
Toronto, Ontario, Canada

Article published in « IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice » August 2025

5.1 Preamble

Once intermittent (flaky) job failures are detected, the next challenge is understanding the diversity of their root causes and determining which categories require priority attention for further automation. This chapter addresses this challenge by establishing a diagnostic taxonomy and a prioritization framework, published in IEEE/ACM ICSE-SEIP 2025 (Aïdasso *et al.*, 2025a). To complement the original paper, Section 5.6 discusses insights from the top priority categories identified, introduces FlakeRanker[1], the open-source tool developed to operationalize the prioritization approach, and outlines practical implications and future research directions.

5.2 Introduction

Intermittent (or flaky) job failures, stemming from issues unrelated to code changes, pose persistent challenges to software organizations. When uncertain of the result of a job, developers usually rerun it multiple times in the expectation of a change in outcome. These multiple job reruns imply a significant waste of machine resources, but also of developers' time, as they must wait for the reruns to complete (Olewicki *et al.*, 2022). At TELUS, diagnosing and repairing certain flaky job failures even requires the intervention of specialized teams, which causes

[1] https://pypi.org/project/flakeranker/

additional delays. For instance, failures due to infrastructure issues such as overloaded CI servers can only be handled by the *infrastructure team* that is responsible for managing these resources.

Prior studies mainly focused on approaches for automatically detecting flaky jobs to minimize the waste associated with multiple reruns. In particular, Lampel *et al.* (2021) used job telemetry data at Mozilla to train an ML model for detecting flaky job failures. At Ubisoft, Olewicki *et al.* (2022) leveraged job log data to automate the detection of flaky jobs. Later, Aïdasso *et al.* (2025b) proposed a novel approach relying on small language models and few-shot learning for this same purpose. Although multiple studies focused on identifying and predicting flaky test categories (Luo *et al.*, 2014; Bell *et al.*, 2018; Rahman & Shi, 2024; Fatima *et al.*, 2024) for their automated diagnosis and repair, flaky tests are only a part of the issues (Lampel *et al.*, 2021). Despite the difficulties involved in diagnosing and repairing flaky job failures, no prior study has systematically categorized their root causes to inform automated solutions.

Hence, there is a strong need to automate the diagnosis of flaky job failures to support developers in such a time-consuming activity. However, one of the biggest challenges is the wide variety of underlying causes of flaky jobs. Such diversity complicates understanding the root causes and requires a wide range of specific repair solutions. As such, focusing on all categories would be too costly for organizations to implement, with limited benefits on failure categories that rarely occur or occur relatively frequently but do not involve significant waste. Finally, many failure categories pose a challenge for automated diagnosis, as the limited number of examples by category constrains the training of accurate ML models.

A well-defined categorization and prioritization of flaky job failures can help in training accurate ML models to detect specific categories of flaky failures as they occur. At a minimum, such a model will help developers to more quickly determine the category of failure they are facing and, therefore, the most appropriate team to solve the problem. Also, by prioritizing failure categories, one can focus efforts on the most recurrent and wasteful issues to implement targeted solutions for their automated repair (e.g., apply flaky test repair techniques (Bell *et al.*, 2018; Rahman & Shi, 2024) if the job failure falls under the *flaky test* category).

To address this gap, we analyze the different categories of flaky job failures at TELUS and then prioritize them based on Recency, Frequency, and Monetary measures. For this purpose, we identify and label 4,511 flaky job failures, which we use to answer the following research questions (RQs):

RQ_1. What are the main categories of flaky failures? More than 40% of the flaky job failures fall into the top five most frequent failure categories, including *misconfigured env variable* (14.92%), *docker daemon connection failure* (7.2%), *container platform auth failure* (6.96%), *job execution timeout* (6.78%), and *image not found* (4.9%). This result underscores the need for prioritization in addressing flaky job failures.

RQ_2. Which failure categories are the most costly? The five most costly failure categories include two of the most frequent (*misconfigured env variable* and *job execution timeout*) and the most widespread, *container registry server error*, which affects 60% of the projects. We also found that the most frequent categories are not always the most costly, evidenced by the 3rd and 5th most frequent categories ranking only 15th and 14th in cost, respectively.

RQ_3. How do the failure categories evolve over time? Several categories, including the second most frequent (*docker daemon connection failure*), have not surfaced in over a year. In contrast, other categories, like *api gateway deployment error*, have only appeared in the last year, or have become more regular over the past few months (e.g., *container registry server error*). These findings highlight the need to also consider recency when prioritizing flaky job failure categories.

RQ_4. What are the priority flaky failure categories? Using an RFM model and clustering, we identified 14 priority categories for future efforts on automated diagnosis and repair, focusing on the most recent, frequent, and costly issues. In addition, we outlined 14 other categories for our industrial partner to monitor as potential future threats.

Our main contributions are (1) a taxonomy of 46 flaky job failure categories for diagnosis and (2) an RFM-based approach to prioritize these categories. The findings also provide critical insights

into the impact and dynamics of different flaky failure categories at TELUS while laying the foundation for future work on automated diagnosis and repair of flaky jobs. A replication package (Aïdasso, 2025a) has been made publicly available. This package includes FlakeRanker[2], a tool that automatically identifies categories of intermittent job failures and highlights the most wasteful ones for prioritization. It has been awarded the "Available", "Functional", and "Reusable" badges at ICSE 2025 Artifact Evaluation Track.

5.3 Background and Motivation

The frequent occurrence of intermittent job failures, also known as *flaky* job failures, leads to a substantial waste of time and resources invested in their diagnosis and repair. A preliminary analysis of job data collected across 80 TELUS projects reveals that one in four job failures (25%) is identified as flaky upon reruns. In the best cases, flaky failures can be *repaired*, i.e., turned into success, simply by rerunning the job several times without further diagnosis. For instance, a job that fails due to an overloaded CI server can pass if rerun later when the server load has decreased. So, in doubt, developers systematically rerun any job failure with unclear results or that they suspect to be flaky. Such a practice results in inefficient use of machine resources and adds a heavy load to already overloaded CI servers (Olewicki *et al.*, 2022).

In more complicated cases, diagnosing and repairing flaky failures requires the intervention of specialized teams, which can extend over several days. For example, a job failure that appears to be caused by a domain name resolution issue may, in reality, be masking an underlying infrastructure problem. Hence, developers submit a ticket to the team they deem the most appropriate between the *network team* and the *infrastructure team* for diagnosis. As tickets are handled in sequence, this request can take a long time (up to 48 hours, according to interviews) before being processed. In the worst cases, solving the problem requires multiple trial-and-error attempts. The time spent on these failures is even greater if the developers initially direct the ticket to the wrong team.

[2] https://pypi.org/project/flakeranker/

The organization of the CI/CD system at TELUS and interviews with engineers have enabled us to identify three important points that motivate our present study:

1. Our industrial partner TELUS does not have dedicated teams, such as Mozilla's Sheriffs (Lampel *et al.*, 2021), to diagnose flaky job failures upon detection. So, while automated detection approaches proposed in previous studies (Aïdasso *et al.*, 2025b; Lampel *et al.*, 2021; Olewicki *et al.*, 2022; Moriconi *et al.*, 2022) are valuable, they remain insufficient to address these failures. As a result, there is a strong need for automated diagnosis solutions to mitigate the associated time and resource inefficiencies.
2. TELUS uses a wide range of distributed tools for its pipeline jobs, which, combined with the development of network applications (i.e., SDNs), make its CI system more susceptible to specific flaky job failures, such as different types of networking issues.
3. The large variety of root causes behind flaky job failures (extending well beyond flaky tests) makes automated diagnosis and repair challenging, since it is impractical to address all possible causes. Some intermittent failures are relatively rare, which limits the availability of examples for training accurate multi-class classification models, while others require extensive investigation, often spanning several days at TELUS.

For these reasons, we aim to identify and prioritize the various categories of flaky job failures. At the very least, our findings will assist developers in diagnosing flaky job failures more rapidly (e.g., direct a ticket to the right team, resulting in reduced wasted time). At best, the prioritization of flaky failure categories will enable us to build, in future research, a multi-class classifier to automate diagnosis and trigger specific repair actions for each priority category.

5.4 Study Design

5.4.1 Research Questions

The main goal of this study is to prioritize the different categories of flaky job failures to facilitate their automated diagnosis and repair. For this purpose, we adopted an approach based on RFM analysis (Birant, 2011), a data mining method commonly used in marketing to segment and

prioritize customers, based on Recency, Frequency, and Monetary (RFM) measures. Hence, we first identify the different categories of flaky job failures and study their frequency (RQ_1). Then, we evaluate the monetary cost associated with these categories (RQ_2) and analyze how each evolves over time (RQ_3). Next, we build an RFM model to segment the identified categories based on their recency, frequency, and monetary cost measures (RQ_4). This segmentation is analyzed to identify categories that should be prioritized for automated diagnosis and repair in future research. To achieve this, we design the following RQs:

RQ_1. What are the main categories of flaky job failures?

RQ_2. Which failure categories are the most costly?

RQ_3. How do the failure categories evolve over time?

RQ_4. What are the priority flaky failure categories?

Table 5.1 Overview of data collected for the diagnosis study

Metric	Value
Start Date	February 1, 2018
End Date	July 11, 2024
# Projects	80
# Languages	13
# Commits	45,545
# Jobs	157,496
# Success Jobs	109,509 (69.53%)
# Failed Jobs	31,058 (19.72%)
# Skipped Jobs	8,443 (5.36%)
# Canceled Jobs	5,804 (3.69%)
# Manual Jobs	2,588 (1.64%)
# Created Jobs	94 (0.06%)
# Rerun Jobs	25,241 (17.96%)
# Flaky Jobs	16,378 (11.65%)
# Flaky Job Failures	7,763 (25%)

5.4.2 Data Collection

Table 5.1 presents a summary of the collected data. In collaboration with our industrial partner, we selected 14 GitLab groups where developers actively contribute to major TELUS projects. After filtering out inactive and toy (with fewer than 50 commits) projects, we identified 80 projects varying in size, maturity, and purpose (e.g., network applications, data analytics, etc.)

and covering 13 programming languages. We used the GitLab APIs[3] to extract the entire build job history of the 80 projects covering 6.5 years, from February 2018 to July 2024. As a result, we collected 157,496 jobs' JSON metadata and 31,058 failed job logs (1.3GB). Then, we merged the JSON metadata into a CSV file and filtered out non-completed jobs (e.g., *canceled*, *skipped*) as they are not relevant to this study. We obtained 140,567 *success* and *failed* jobs.

We identified flaky jobs based on the following heuristic: if a job is rerun with no changes to the source code (i.e., on the same commit) and its outcome changes between success and failure, it is considered flaky; otherwise, it is treated as a regular job. This approach was used previously (Olewicki *et al.*, 2022; Aïdasso *et al.*, 2025b) and aligns with the development practices at TELUS, as confirmed through interviews. Hence, we grouped the jobs by *project*, *name*, and *commit* and identified the job rerun sequences (i.e., the groups of jobs with at least two executions of the job on the same commit). We saved these rerun sequence data for further analysis. For each job rerun sequence containing at least one *success* and one *failed* job, we labeled its jobs as flaky. As a result, we found 16,378 flaky jobs from which we excluded the jobs with a *success* status since we are only interested in failure reasons from failed job logs.

We obtained 7,763 flaky job failures representing a quarter (25%) of the failed jobs as shown in Table 5.1. Out of these 7,763 flaky job failures, we found that 2,507 (33.29%) jobs have missing log data. A brief investigation revealed that this phenomenon stems from version compatibility issues between self-hosted GitLab and GitLab Runners following major upgrades, which result in job execution logs not being recorded[4]. Engineers at TELUS confirmed that the lost logs resulted from a major migration of GitLab to a public cloud in early 2022. Since our approach to identifying the different categories of flaky failures is mainly based on log data, we excluded the flaky job failures with missing logs.

The remaining 5,256 flaky job failures with log data constitute the initial dataset for the current study. We label this dataset of flaky failures using the approach described below. In

[3] https://docs.gitlab.com/ee/api/jobs.html

[4] https://stackoverflow.com/a/69378998/26508024

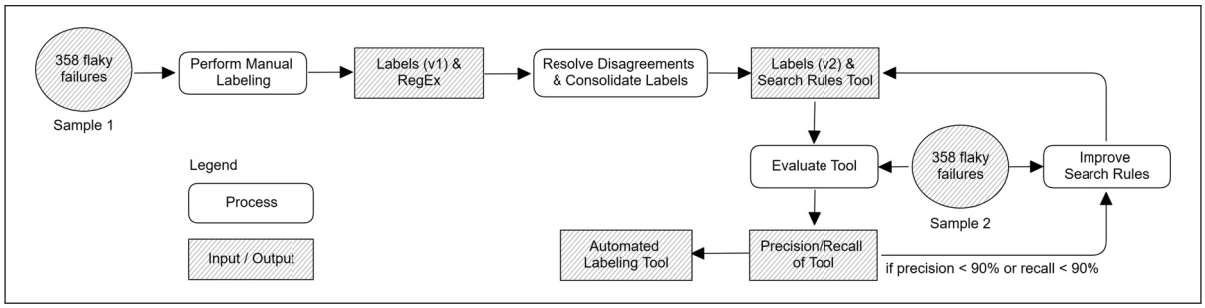


Figure 5.1 Overview of the approach for building the automated flaky job failure labeling tool

addition, we use the rerun sequences data to answer RQ_2. For confidentiality reasons, our replication package does not include the datasets.

5.4.3 Identifying Flaky Job Failure Categories and Automating Labeling

To identify the different categories of flaky job failures (and label the collected job data), we use a semi-automated approach involving (1) the development of a multi-class labeling tool based on search rules and (2) the application of this tool to label the dataset systematically. A labeling tool is essential for this study to complete the labeling of thousands of flaky job failures. It can also be used in future studies to obtain labeled data. Fig. 5.1 presents our approach to developing the automated labeling tool.

Dataset Sampling. Manually analyzing thousands of job logs for labeling is not practical. So, similarly to previous studies (Beller, Bacchelli, Zaidman & Juergens, 2014; Ghaleb *et al.*, 2021), we use two statistically significant samples of 358 job logs each, out of the 5,256 in the initial dataset (with a confidence level of 95% and a 5% margin of error). Sample 1 is used for manual labeling and building the automated labeling tool, while Sample 2 is reserved to evaluate the tool's accuracy. We randomly select from the initial dataset 358 flaky job failures constituting Sample 1. In contrast, the 358 jobs for Sample 2 are drawn from the dataset deprived of Sample 1 jobs to ensure that the tool is validated on unseen data.

Samples Manual Labeling. For each job in Sample 1 that is not yet labeled, we manually analyze the logs to identify the cause of the failure. When the error message is initially unclear, we use StackOverflow (SO) to understand it better. For example, an investigation on SO has enabled us to realize that the `"exit code 13"` error indicates that a container ran out of memory due to the limits specified in its pod.[5] Then, we use open coding (Khandkar, 2009) to assign the flaky job failure to an existing category or create a new one with a name reasonably indicative of the failure root cause (e.g., *container oom error* for the previous error example). Each category name has been validated with TELUS engineers to ensure clarity. If a new category is created, we look in the logs for a regular expression (regex) capable of identifying failures with highly similar logs, as precisely as possible, to maximize the precision of our tool. We then use a script that searches for similar job failures based on the logs using this new regex. The jobs found are then also assigned to the newly created category. To avoid mislabeling, we manually validate each new job labeled with the new category name. Also, the new regex pattern and its associated label are saved for later building our search rules-based tool. We repeat this process until all the jobs in Sample 1 are labeled. As a result, we obtain the first version of labels (v1) and their associated regexes. Later, we manually labeled the jobs in sample 2, following the same process.

Labeling Tool Creation. To develop the labeling tool, we first established the final set of labels (v2) and associated regex through regular meetings with TELUS engineers, during which we discussed and resolved disagreements. In addition, we consolidated similar and homogeneous categories to avoid excessive labels with limited examples, which would undermine efficient data analysis and the training of future ML classifiers. For instance, we merged the *missing env variable* category with the *misconfigured env variable* category since (1) they have similar log error messages, and (2) the diagnosis and repair of both these issues consist of finding and setting the correct CI environment variable value. Each time we merge categories, we combine the corresponding regexes (e.g., using the OR | operator). In the end, we defined an ordered set of 51 heuristics in the form of search rules that associate a regex pattern to one of the 46 final

[5] https://stackoverflow.com/a/69676383/26508024

labels. We then developed a Python script to leverage these search rules to automatically label an input dataset of flaky job failures with the appropriate labels based on the job logs.

Tool Evaluation and Improvement. We evaluate the performance of our labeling tool using two key metrics: *recall* to evaluate its coverage and *precision* to assess its accuracy. Recall is defined as $\frac{n_{labeled}}{N}$, where N is the total size of the input dataset and $n_{labeled}$ is the number of jobs labeled by the tool. Precision is calculated as $\frac{n_{correct}}{n_{labeled}}$, where $n_{correct}$ represents the number of jobs correctly labeled, determined through manual inspection.

On the unseen data from Sample 2, the labeling tool achieved a precision of 99.31% (and a recall of 81.56%), demonstrating the effectiveness of our approach in accurately labeling flaky job failures as intended. Also, the obtained recall of 81.56% on Sample 2 shows that the identified categories are pretty representative of the types of failures in the overall dataset. Although we obtained a satisfying recall, we sought to improve it using the remaining unlabeled data from Sample 2. So, we followed the manual labeling process again, updating the regex patterns and search rules after each iteration. In the end, no new categories were discovered. We stopped after achieving precision and recall scores of over 90%. As a result, we increased the tool's recall to 91.34% on Sample 2.

Automated Labeling. We applied the labeling tool to our 5,256 flaky job failures dataset, successfully labeling 85.83% of the dataset, i.e., 4,511 jobs. We consider these labeled flaky job failures to be representative of the issues plaguing CI at TELUS. Since our study focuses on diagnosing flaky job failures, we limited our analysis to failures where the root cause can be determined from the logs and successfully labeled using the tool. Consequently, the remaining 745 unlabeled failures have been excluded from the analysis.

Summary of Labeling. Our labeling tool achieves a precision of 99.31% (recall of 81.56%) on an unseen random representative sample. Leveraging this tool, we labeled 4,511 flaky job failures ($\approx$ 86% of the initial dataset). We use this labeled dataset to answer our RQs.

5.5 Results

5.5.1 RQ_1. What are the main categories of flaky job failures

5.5.1.1 Motivation

This RQ aims to identify the most prevalent categories of flaky job failures. Despite efforts invested in research on flaky jobs, there is a lack of awareness about specific categories of flaky job failures that can be used for diagnostic purposes. Also, the findings of this RQ will provide valuable insights into the recurrent categories of flaky job failures that we need to consider prioritizing to maximize the impact of automated diagnosis and future repair solutions.

5.5.1.2 Approach

To answer this RQ, we analyze our dataset of 4,511 flaky failures, labeled using the approach described in Section 5.4.3. We report statistics about the number and proportion of flaky job failures in each identified category. In addition, we report the number of distinct projects affected by each failure category to better assess their impact across projects. Also, to facilitate the understanding of the flaky failure categories, we use the card sorting (Spencer, 2009) technique to organize related failure categories into meaningful groups.

5.5.1.3 Results

Table 5.2 presents the 46 identified categories of flaky failures organized into 14 thematic groups.

Table 5.2 Categories of flaky job failures at TELUS

Group	Category	#	%	P*
Environment Variables	misconfigured_env_variable	673	14.92	35
Container Issues	container_already_exists	2	0.04	1
	container_not_found	31	0.69	7
	docker_daemon_connection_failure	325	7.20	13
	image_build_permission_denied	8	0.18	1
	image_build_read_error	17	0.38	5
	image_push_write_error	1	0.02	1
	image_security_scan_failure	8	0.18	7
Unauthorized Access	certificate_verification_failure	56	1.24	16
	container_platform_auth_failure	314	6.96	34
	repository_access_denied	29	0.64	9
Limits Exceeded	docker_pull_limit_reached	30	0.67	5
	cloud_token_limit_exceeded	39	0.86	6
	job_execution_timeout	306	6.78	19
	remote_call_timeout	76	1.68	22
	runner_pod_waiting_timeout	199	4.41	34
	stuck_or_timeout_failure	14	0.31	9
Remote Resource Issues	api_gateway_deployment_error	161	3.57	6
	image_not_found	221	4.90	25
	container_registry_server_error	213	4.72	50
	external_file_invalid_format	125	2.77	8
	http_resource_not_found	38	0.84	6
	service_unavailable	2	0.04	2
Networking Issues	connection_closed_reset_broken	204	4.52	38
	connection_refused	44	0.98	11
	host_resolution_failure	91	2.02	29
	broker_connection_failure	7	0.16	2
	ssl_connection_error	16	0.35	5
Flaky Tests	flaky_test	179	3.97	11
Infrastructure Issues	helm_resource_error	99	2.19	33
	runner_image_pull_failure	99	2.19	28
	runner_instance_error	32	0.71	8
	runner_pod_failure	154	3.41	32
	runner_pod_not_found	10	0.22	7
Dependency Issues	buggy_dependency	13	0.29	4
	dependencies_conflict_error	129	2.86	4
	dependency_installation_failure	124	2.75	22
Transient VCS Errors	git_transient_error	113	2.50	31
Memory Issues	container_oom_error	55	1.22	11
	testing_device_oom_error	6	0.13	5
	static_analysis_tool_oom_error	57	1.26	8
Repository File Issues	repository_file_access_error	60	1.33	10
	repository_file_not_found	37	0.82	10
Internal OS Issues	apt_timezone_issue	66	1.46	7
	os_cmd_execution_error	12	0.27	3
Database Issues	db_table_undefined	16	0.35	1
Total		**4,511**	**100%**	**80**

*Number of projects affected.

For each category, it shows the frequency, the percentage proportion, and the number of projects impacted. Additionally, it highlights in blue the top five categories with the highest frequencies and the top five with the highest number of affected projects.

The most frequent category of flaky job failure is misconfigured environment (env) variable, accounting for ≈ 15% of the flaky job failures. Our analysis reveals that, of the 4,511 labeled flaky job failures, 673 (14.92%) belong to the *misconfigured env variable* category. This category encompasses failures due to missing environment variables, misspelled variable names, and incorrect values of environment variables used during job execution. The *misconfigured env variable* category is also the third most widespread category across the projects, affecting 35 (44%) of the 80 projects studied.

Identifying misconfigured env variables as the most frequent failure category can be explained by two main reasons. Firstly, CI environment variables are not version-controlled, preventing their changes from being tracked via commits. Consequently, failed jobs due to misconfigured variables are identified as flaky when successfully rerun after proper configuration. Second, the build scripts of SDNs at TELUS depend on multiple environment variables that are centrally managed at the GitLab group level by team leaders and simply inherited by individual projects. The lack of direct access to the CI variables leads developers to request their updates and rerun failed jobs several times, until they are correctly configured.

These findings call on CI providers to provide better tooling for CI variable management, such as version control and auto-completion for variable names. They also emphasize the increasing importance of managing build environment resources, such as CI variables, as part of development activities.

Over 40% of the flaky job failures belong to the top five most frequent failure categories, underlining the need for prioritization in addressing them. Following the most frequent failure category, *misconfigured env variable* (14.92%), the next flaky failure categories are *docker daemon connection failure* (7.20%), *container platform auth failure* (6.96%), *job execution timeout* (6.78%), and *image not found* (4.90%). These five most frequent categories collectively

constitute 40.76% of the studied flaky job failures. This finding demonstrates that the majority of flaky job failures are concentrated in a small number of categories, making it more effective to prioritize and address the most wasteful ones.

A *docker daemon connection failure* arises when a job attempts to execute a Docker[6] command but cannot connect to the Docker daemon, either on a local runner pod (e.g., Docker is not properly installed or running) or on a remote host that is unreachable. A *container platform auth failure* occurs when the job fails to authenticate to the container platform used during CI due to an expired token, for example. There is a *job execution timeout* when the duration of the job exceeds the predefined limit. Finally, the *image not found* category encompasses jobs that attempt to use a container image that is not present in the registry of the container platform. Engineers noted during interviews that this often occurs when a job relies on images that have not yet been created in the internal container platform. Once the image becomes available, the job typically runs successfully.

The most popular category across projects is the container registry server error affecting over 60% of the studied projects and ranking 6th most frequent. Our results show that the *container registry server error* is the most widespread failure category across projects, appearing in 50 (62.50%) of the 80 studied projects. It also ranks 6th, just below the top five most frequent categories, accounting for 4.72% of the flaky job failures. This failure category occurs when a server-side issue prevents jobs from successfully pulling (or pushing) images from the shared container registry.

Furthermore, the most popular categories across projects also rank highly in frequency. For instance, two of the top five most popular categories across projects, *misconfigured env variable* and *container platform auth failure*, are among the top five most frequent. The remaining three most popular categories highlighted, i.e., *container registry server error*, *connection closed reset broken*, and *runner pod waiting timeout*, are the 6th, 7th, and 8th most frequent, representing 4.72%, 4.52%, and 4.41% of the flaky job failures, respectively.

[6] https://docs.docker.com/reference/cli/docker/

5.5.2 RQ_2. Which failure categories are the most costly

5.5.2.1 Motivation

The objective of this RQ is to determine which failure categories are the most costly in terms of diagnosis time and machine resources due to reruns. In particular, we sought to determine whether the most frequent categories are also the most costly. The results of this RQ will provide our industrial partner with valuable insights into the most wasteful failure categories, enabling a better prioritization for automated diagnosis and repair solutions.

5.5.2.2 Approach

As discussed before, the waste associated with flaky job failures is related to the multiple job reruns and the (long) time spent by engineers diagnosing the initial failure. Therefore, to evaluate the cost associated with each failure category, we design a cost model including two components: (1) the machine cost and (2) the diagnosis cost. It is inspired by an existing approach for estimating the cost of executing and diagnosing test failures at Microsoft (Herzig, Greiler, Czerwonka & Murphy, 2015).

To estimate the total machine cost ($Cost_{machine}$) associated with a failure category, we calculate the cumulative duration spent by the jobs in the category. Then, we apply the average pricing rate of the infrastructure machines used to execute the jobs. Although this pricing rate can vary based on factors such as region and instance type, for simplicity, we assume all jobs run on the equivalent of a high-memory virtual machine (*n2-highmem-128*) from Google Cloud. We calculated the pricing rate to be \$0.14/min. Finally, the formula for calculating the machine cost is outlined in Eq 5.1 as follows:

Let $E = \{job_1, job_2, ..., job_n\}$ denote the ensemble of n flaky job failures within the category E, each with respective durations $d_1, d_2, ..., d_n$ in minutes (i.e., $d_i = job_i[\text{duration}], i = 1..n$).

$$Cost_{machine}(E) = \left(\sum_{i=1}^{n} d_i \right) \times M \tag{5.1}$$

where M is the constant per-minute price of the infrastructure machines used to run the jobs.

To estimate the total diagnosis cost ($Cost_{diagnosis}$) associated with a category, we hypothesize (from interviews) that the diagnosis process starts after an initial flaky failure and ends with the last rerun (See Fig. 1.6). Hence, we sum up the time delay required to complete the diagnosis of each initial failure in a category. Then, we multiplied this total time delay by the constant S representing the average cost per minute of diagnosing a flaky failure. For comparative purposes, this constant reflects the hourly rate of a single engineer, even though multiple engineers are often involved in the diagnosis. Accordingly, we estimated the constant based on an average engineer's salary of $36 per hour, which translates to $0.6/min. Eq 5.2 shows the formula of the diagnosis cost for a category, described below:

Let $F = \{f_1, f_2, ..., f_m\} \subset E$ $(m < n)$ denote the subset of E containing the initial flaky job failures in their respective rerun sequences $R_1, R_2, ..., R_m$; with $l_1, l_2, ..., l_m$ representing the last executed jobs in each rerun sequence, respectively.

$$Cost_{diagnosis}(E) = \left(\sum_{j=1}^{m} Time_{delay}(R_j) \right) * S \tag{5.2}$$

where $Time_{delay}(Rj) = l_j[\text{finished_at}] - f_j[\text{finished_at}]$ calculated in minutes, and S is the constant average salary of an engineer at the company, estimated per minute.

Finally, we sum the two cost components to calculate the total $Cost$ associated with a category as shown in Eq 5.3.

$$Cost(E) = Cost_{machine}(E) + Cost_{diagnosis}(E) \tag{5.3}$$

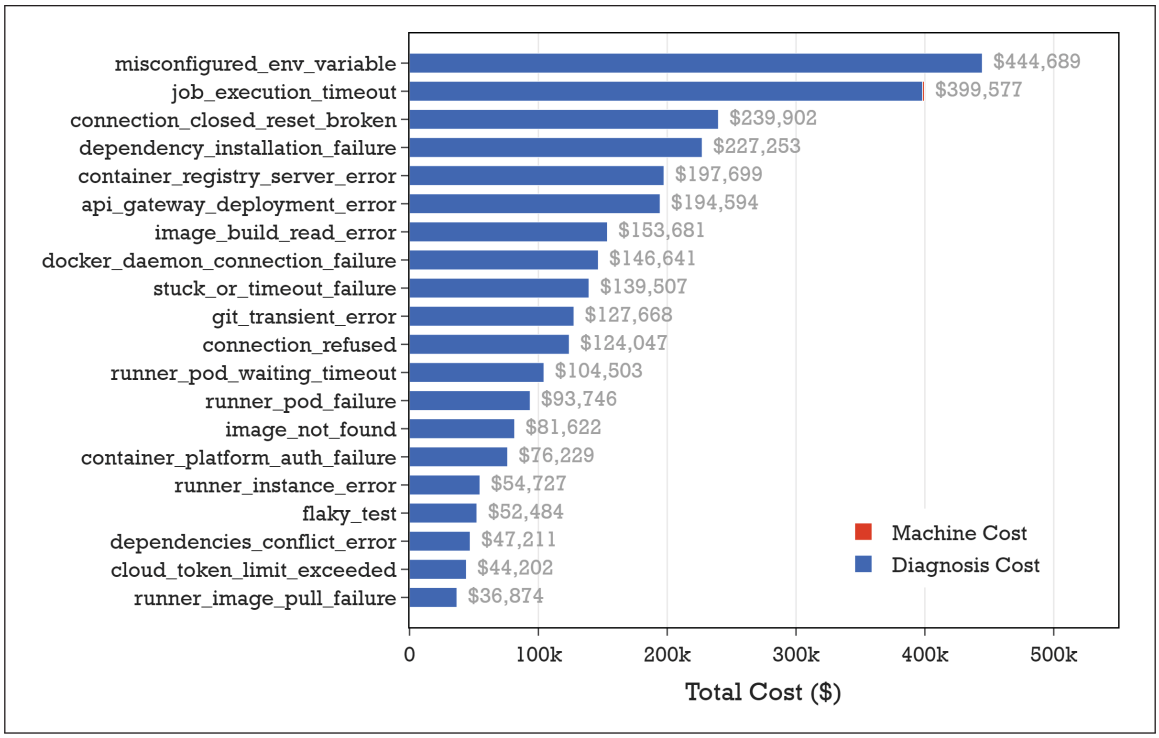


Figure 5.2 Ranking of the 20 most costly flaky job failure categories

To answer our RQ, we rank the failure categories according to the calculated costs. Then, we plot and discuss the 20 most costly categories. Our replication package includes additional figures showing the ranking of the remaining categories.

5.5.2.3 Results

By a significant margin, the most costly failure categories are misconfigured environment variable and job execution timeout, with costs essentially related to diagnosis. Fig. 5.2 shows the 20 costliest flaky failure categories, with the proportions of diagnosis and machine costs for each category highlighted in blue and red, respectively. The most costly category is *misconfigured env variable*, closely followed by *job execution timeout*. These two categories are particularly outstanding given that their costs are nearly twice that of the third costliest (*connection closed reset broken*). They also rank 1st and 4th most frequent categories, as highlighted in Table 5.2.

Besides, almost all the costs associated with the failure categories are related to diagnostic times. As depicted in Fig. 5.2, the diagnosis cost (in blue) constitutes the overwhelming majority of the total cost for each of the failure categories, with cost proportions ranging from 99.63% for *flaky test* to 100% for *runner instance error*. This result underscores the need for automated solutions to address the time-consuming and costly diagnosis (and repair) of flaky job failures.

While cost and frequency show some correlation, the costliest failure categories are not always the most frequent, emphasizing the importance of considering cost alongside frequency in prioritizing failure categories. The two most costly flaky job failure categories, *misconfigured env variable* (1st) and *job execution timeout* (2nd), are followed in the top five most costly respectively by the *connection closed reset broken* (3rd), *dependency installation failure* (4th), and *container registry server error* (5th) categories. The 3rd, 4th, and 5th costliest categories do not figure among the five most frequent ones, ranking instead 7th, 14th, and 6th most frequent, respectively. Similarly, the 3rd and 5th most frequent categories, *container platform auth failure* and *image not found*, only rank 15th and 14th most costly. Nevertheless, the top five costliest categories account together for more than a third (33.69%) of the flaky job failures, with respective proportions of 14.92%, 6.78%, 4.52%, 2.75%, and 4.72%.

5.5.3 RQ_3. How do the failure categories evolve over time

5.5.3.1 Motivation

The objective of this RQ is to explore the evolution of the failure categories over time. Specifically, we aim to determine whether certain categories have ceased to occur or if new ones have emerged recently. The findings of this RQ will provide valuable insights into the recency and the dynamics of the failure categories, helping to pinpoint the most recent issues that affect the CI system and require attention.

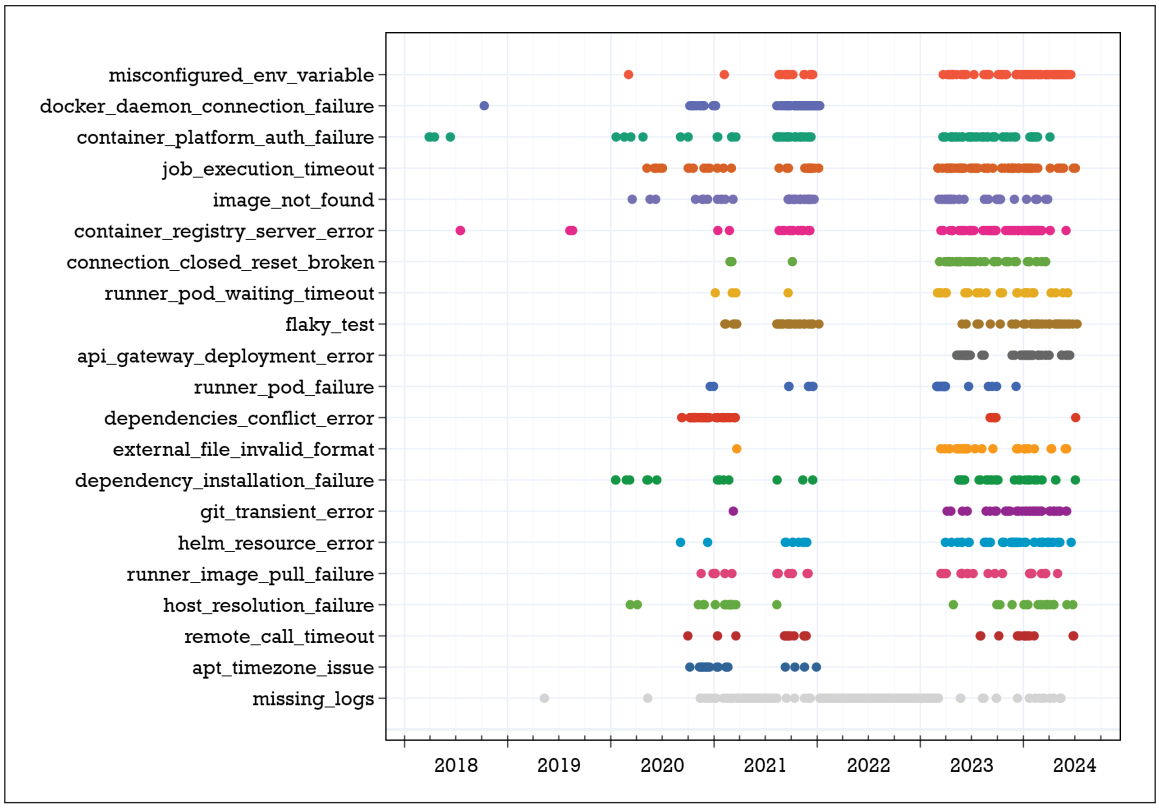


Figure 5.3 Evolution over time of the 20 most frequent flaky job failure categories, arranged from top to bottom. The final scatter plot at the bottom shows the evolution of flaky job failures with missing log data

5.5.3.2 Approach

We identify the distinct creation dates of the jobs in each category. Then, we use a one-dimensional scatter plot to visualize the occurrence dates of the categories across the full 6-year period (indicated in Table 5.1) of the collected data. We intentionally neutralize the daily frequency dimension on the plot since this RQ focuses on temporal trends. To clarify gaps in our visualizations, we also display the evolution of flaky failures with missing logs, which account for a significant proportion of the initial 7,763 flaky job failures. For simplicity of the figure, we only present the 20 most frequent categories in this paper. Complete figures are in the replication package.

5.5.3.3 Results

Fig. 5.3 presents the evolution over time of the 20 most frequent flaky failure categories. There are notable gaps in the timeline across all categories during the second quarter of 2021 and again between the beginning of 2022 and the first quarter of 2023, which we refer to as the *big blank period* in the following. These gaps align with the periods of missing job logs due to major process changes (e.g., GitLab migration).

Several failure categories, including the second most frequent, have not been observed for at least a year. As shown in Fig. 5.3, the categories *docker daemon connection failure* and *apt timezone issue* do not present any data point after the big blank period. Although *docker daemon connection failure* is the second most frequent category, all the flaky job failures identified in this category occurred before February 2022, more than two years before this study. In addition, while spotted recently, the *dependencies conflict error* category was more regularly observed before the second quarter of 2021.

Certain failure categories have only recently emerged or consistently recurred, following major process changes. Fig. 5.3 shows that the *api gateway deployment error* category is identified only after the big blank period. It also reveals that several of the 20 most frequent categories, notably *misconfigured env variable*, *flaky test*, *git transient error*, and *helm resource error* have more consistent apparitions from late 2023 to the second quarter of 2024. These findings suggest that the changes made to the CI process in early 2022 (e.g., the GitLab migration) have resolved specific causes of flakiness, such as *docker daemon connection failure*, while giving rise to new issues such as the *api gateway deployment error*.

While some old categories have resurfaced recently, they have not shown consistent recurrence over the recent period. As depicted in Fig. 5.3, the last occurrence dates for the categories *dependencies conflict error* and *remote call timeout* are around mid-2024. However, the occurrence dates preceding these latest appearances are respectively about nine months earlier, at the end of the 3rd quarter of 2023, and five months earlier, in the middle of the first

quarter of 2024. Such a finding calls for a cautious definition of recency, as a very recent occurrence does not always indicate a recurring current issue.

5.5.4 RQ_4. What are the priority flaky failure categories

5.5.4.1 Motivation

The goal of this RQ is to prioritize flaky failure categories based on their studied characteristics: recency, frequency, and cost. The findings will guide the development of an ML model for automated diagnosis and help our industrial partner focus on priority failure categories requiring specific repair solutions. Additionally, our prioritization framework can serve as a useful approach for researchers addressing a large variety of failure types in various contexts.

5.5.4.2 Approach

We build an RFM model (Birant, 2011) that takes the R-F-M values of all failure categories as input and applies the K-means[7] clustering algorithm to group them into different clusters. RFM model is a widely used method for behavioural segmentation (clustering) (Alves Gomes & Meisen, 2023). The resulting clusters are analyzed to determine which ones should be prioritized. For this purpose, we create a dataset comprising, for each of the 46 failure categories, the R-F-M values defined as follows:

Recency (R). We calculate the average number of days that have elapsed since the last three occurrences of the failure category up to the date of the current study. More recent categories have lower recency values. Although traditional definitions of recency only use the last occurrence, we consider the last three in this study to minimize the relative importance of categories that occurred without consistent recurrence in the recent period. For example, we calculated a recency value of 16 and 195 days for the categories *flaky test* and *dependencies conflict error*, while their last occurrences date back only to 2 and 7 days, respectively (as shown in Fig. 5.3).

[7] https://scikit-learn/stable/modules/generated/sklearn.cluster.KMeans.html

Frequency (F). We use the frequency value of the failure category as identified in RQ_1 and reported in Table 5.2. A higher frequency value indicates that the failure category has been encountered more often by developers.

Monetary (M). We use the estimated monetary cost associated with the failure category as calculated in RQ_2. A higher monetary value indicates (based on our previous findings from RQ_2) that developers spend a significant amount of time diagnosing this failure category.

Table 5.3 Flaky job failure categories with outliers RFM values

Category	R (Days)	F (#)	M ($)	Pattern
misconfigured_env_variable	27	673	444,689	R↑F↑M↑
job_execution_timeout	8	306	399,577	R↑F↑M↑
docker_daemon_connection_failure	915	325	146,641	R↓F↑M↑
testing_device_oom_error	1,265	6	6,377	R↓F↓M↓
image_build_permission_denied	1,415	8	702	R↓F↓M↓

Since K-means is sensitive to outliers, we identify the outliers in our RFM dataset and treat them separately. We use Isolation Forest[8] (IF), an outlier detection model that leverages a random recursive partitioning technique, resulting in stable outcomes across different initializations. We train the IF model using 500 estimators and a contamination ratio of 10% to predict the outliers. We repeat the model training and prediction 100 times to ensure the accurate detection of the outliers. Each iteration consistently identified the same five outlier categories reported in Table 5.3. These outliers are removed from the RFM dataset and discussed in the results.

We perform RFM scoring using the quintile method, a data normalization technique that ensures all features are scaled to a uniform range. We divide the 41 categories into five equal bins according to the quintiles of the R(F-M) measure. For F(M), a score of 5 is assigned to the top quintile and 4, 3, 2, and 1 to the others. For the R score, 5 indicates a recency value in the lowest quintile. As a result, an RFM score of 555 is attributed to a very recent, frequent, and costly category, while a score of 111 indicates an ancient, rare, and cheap category.

[8] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html

We fit a K-Means[9] model to group categories with similar RFM scores. The model uses `k-means++` initialization to optimize centroid selection. In line with previous work (Birant, 2011), we set the number of clusters $k = 8$ to group the categories into eight distinct clusters for analysis. In fact, each cluster is analyzed by comparing its average R, F, and M scores to the overall averages, with an upward ↑ or a downward ↓ arrow assigned based on whether the average score of the cluster surpasses or falls below the corresponding overall average (resulting in 8 possible combinations). For instance, the pattern R↓F↑M↑ means that the cluster has an average recency below the overall average, while the average frequency and monetary values are above the respective total averages.

We evaluate the quality of our resulting clusters using the Sum of Squared Errors[10] (SSE) and the standard deviation. The K-means algorithm was run 500 times, and the model with the lowest SSE value was selected. The clustering results are then reported to answer this RQ.

Table 5.4 K-Means clustering of flaky job failure categories based on RFM scores

Cluster	**Size**	**Std**	**Recency (Avg.)**		**Freq. (Avg.)**		**Monetary (Avg.)**		**Pattern**	**Description**
			Days	**R**	**#**	**F**	**$**	**M**		
C1	6	0.48	35.83	4.83	164.83	4.67	150,700	4.67	R↑F↑M↑	High Priority
C2	6	0.45	55.83	4.50	88.17	3.83	29,341	3.17	R↑F↑M↑	Medium Priority
C3	5	0.55	119.40	3.60	47.20	2.60	10,970	2.40	R↑F↓M↓	Low Priority
C4	4	0.51	120.50	3.75	16.00	1.50	743	1.00	R↑F↓M↓	Emerging Issues
C5	5	0.45	162.20	2.80	204.40	4.80	107,742	4.20	R↓F↑M↑	Idle
C6	4	0.44	229.75	2.00	26.75	2.00	117,991	4.75	R↓F↓M↑	Relic
C7	4	0.61	471.25	1.75	45.00	2.75	7,404	2.00	R↓F↓M↓	Negligible
C8	7	0.30	626.86	1.00	9.43	1.14	3,736	1.43	R↓F↓M↓	Irrelevant
Total	41			3.03		2.91		2.95		

5.5.4.3 Results

Table 5.4 presents the clustering results, describing the eight clusters identified. For each cluster, it shows the size and the average standard deviation (Std) across RFM scores, followed by the average RFM actual values and scores. The overall averages of RFM scores are shown in the

[9] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

[10] https://hlab.stanford.edu/brian/error_sum_of_squares.html

final row. The table also includes RFM patterns and descriptions of the clusters, ranked from *high priority* to *irrelevant* based on the patterns and average RFM scores. The eight clusters are well-balanced, with sizes ranging from 4 (for clusters C4, C6, C7) to 7 (for C8). Table 5.4 also shows that the clusters are statistically distinct, with average standard deviations ranging from 0.61 being the highest to 0.30 being the lowest.

Table 5.5 Identified 14 priority flaky job failure categories

Category	**R (Days)**	**F (#)**	**M ($)**	**Priority**
misconfigured_env_variable	27	673	444,689	Top
job_execution_timeout	8	306	399,577	Top
dependency_installation_failure	54	124	227,253	High
runner_pod_waiting_timeout	35	199	104,503	High
api_gateway_deployment_error	29	161	194,594	High
container_registry_server_error	41	213	197,699	High
git_transient_error	40	113	127,668	High
flaky_test	16	179	52,484	High
external_file_invalid_format	41	125	29,672	Medium
host_resolution_failure	24	91	23,709	Medium
runner_image_pull_failure	99	99	36,874	Medium
cloud_token_limit_exceeded	58	39	44,202	Medium
remote_call_timeout	61	76	25,371	Medium
helm_resource_error	52	99	16,217	Medium

We identified 14 priority flaky failure categories (≈ 30%) as the most recent, frequent, and costly issues, which will be the focus of future research on automated diagnosis and repair, and are summarized in Table 5.5. We prioritize flaky job failure categories with high R, F, and M scores, and therefore, those that are associated with the pattern R↑F↑M↑. As such, we first consider the outliers associated with this pattern. As shown in Table 5.3, *misconfigured env variable* and *job execution timeout* are the top two categories to be prioritized, with the lowest recency values and highest frequency and monetary values. These top priority categories are followed by those in clusters C1 and C2, totaling 12 additional categories as shown in Table 5.4. C1 is ranked as *high* priority due to its average RFM scores being all close to 5 and higher than

those of C2. Consequently, C2 is ranked as *medium* priority although it follows a similar RFM pattern. The 14 priority failure categories account for 30.43% of the 46 identified categories.

A substantial number (≈ 30%) of flaky failure categories require close monitoring as potential recidivists or emerging issues. The clustering results show that C3 (R↑F↓M↓) comprises recent categories (R=3.60) with low frequency and cost, which we consider to be of *low priority*. Categories in C4 (R↑F↓M↓) are even more recent on average (R=3.75) with lower frequency and cost, which indicates this cluster may include *emerging issues* (e.g., *runner pod not found*). The categories in these clusters need to be closely monitored as they reflect current issues that can potentially affect the system more severely over time. In addition, *idle* failure categories in C5 (R↓F↑M↑) also deserve special attention as they represent highly frequent and costly categories that have not surfaced for a little while. In particular, these categories include recently resolved ones (e.g., *runner pod failure*), for which monitoring is essential to ensure they are fully resolved and do not resurface. C3, C4, and C5 total 14 other failure categories.

The remaining failure categories (≈ 39%) can be safely ignored. C6 (R↓F↓M↑) contains *relic* categories that were highly costly but are now bygone. The outlier *docker daemon connection failure* (R↓F↑M↑) also falls into that kind of failure category, which can be analyzed to gain insights into major past issues within the CI system. Clusters C7 and C8 (R↓F↓M↓) group together failure categories described respectively as *negligible* and *irrelevant*. Along with the outliers of the same RFM pattern, i.e., *testing device oom error* and *image build permission denied*, these categories reflect outdated issues that occurred infrequently with minimal costs.

5.6 Discussion

5.6.1 Insights from Priority Categories

The identified priority categories reveal important patterns about the nature of reliability issues in industrial CI/CD environments. We discuss the two most prominent categories.

Environment Variable Misconfiguration. The *misconfigured_env_variable* category emerged as the top priority, accounting for 14.92% of all flaky failures and incurring the highest monetary cost estimated ($444,689). At first glance, environment variable errors might seem deterministic rather than intermittent: a missing or incorrect variable should consistently cause failure. However, our analysis reveals why this category legitimately belongs among intermittent failures. At TELUS, and likely in many large industrial settings, CI/CD pipelines operate within a hierarchical project structure. Environment variables are often defined not by the developers directly responsible for a project, but in parent projects from which they inherit configurations, or by dedicated infrastructure teams managing shared CI/CD resources. This organizational separation creates scenarios where:

- Variables are modified upstream without notification to downstream projects
- Infrastructure changes propagate inconsistently across environments
- Timing-dependent variable resolution leads to non-deterministic outcomes
- Developers lack visibility into (or access to) the configurations affecting their pipelines

For instance, consider a pipeline where multiple parallel jobs inherit environment variables from a dynamically generated artifact or a shared secrets manager. If one job starts before the artifact is fully propagated across runners, or during a variable rotation window in the secrets manager, it may receive an undefined or stale value while sibling jobs succeed. The same pipeline, on the same commit, thus produces different outcomes depending on when each job resolves its variables. This is a typical intermittent job failure pattern that is invisible to developers who lack access to the underlying infrastructure timing.

Besides, unlike source code (which is version-controlled and traceable), environment variable changes in CI/CD systems are generally untracked, making failures appear “flaky” from the developer’s perspective. These findings call for better practices in CI/CD variable management, such as automated validation of variable availability before job execution, spell-checking and value suggestion based on learned patterns from other organizational projects, and version-controlled environment configurations with change notifications to dependent projects.

Job Execution Timeouts. The *job_execution_timeout* category ranked second in priority, with 306 occurrences and $399,577 in estimated costs. While timeouts may appear straightforward (i.e., a job simply exceeds its allocated time), our observations and prior research suggest a more nuanced reality. Timeouts often mask diverse underlying causes beyond mere duration limits. Weeraddana *et al.* (2024a) found that builds may fail due to internal timeouts (e.g., test framework timeouts, network request timeouts) that manifest as job-level failures without clear root cause attribution. At TELUS, we observed timeouts associated with:

- Non-optimized processes consuming excessive resources
- Cascading delays from upstream service latency
- Resource contention (e.g., Kubernetes pods) on shared CI infrastructure
- Intermittent network slowdowns affecting external dependencies loading

These findings suggest that simply increasing timeout thresholds is not always an adequate solution. Instead, organizations need deeper analysis to distinguish between legitimate long-running processes and pathological delays, enabling targeted mitigation strategies. Future research should investigate automated root cause analysis for timeout failures and develop adaptive timeout mechanisms that account for historical patterns and current infrastructure load.

5.6.2 FlakeRanker: A Tool for Automated Labeling and Prioritization

To operationalize the methodology developed in this study, we engineered FlakeRanker, a command-line interface (CLI) tool that automates the identification and prioritization of flaky job failure categories. The tool is openly available on PyPI[1] and was awarded the "Available", "Functional", and "Reusable" badges at the ICSE 2025 Artifact Track (Aïdasso, 2025b,a).

FlakeRanker comprises three modules corresponding to the stages of our methodology:

1. **Labeler**: Identifies flaky jobs and assigns failure categories based on regex patterns derived from our taxonomy. It takes job metadata and execution logs as input and outputs a labeled dataset as a CSV file.

2. **Analyzer**: Computes Recency, Frequency, and Monetary values for each identified category, producing an RFM dataset.
3. **Ranker**: Applies quintile-based RFM scoring and K-means clustering to prioritize categories, outputting ranked categories with cluster assignments and RFM patterns.

These modules can be executed individually for flexibility or combined via a single command (`flakeranker run`) to streamline the entire pipeline.

Integrating FlakeRanker into CI/CD Workflows. FlakeRanker can be integrated into organizational CI/CD monitoring workflows to provide continuous visibility into flaky job failure trends. We envision the following use cases:

Periodic batch analysis. Organizations can periodically run FlakeRanker on static accumulated job failure data (e.g., weekly or monthly) to generate prioritization reports. These reports will inform strategic decisions about which failure categories to address through infrastructure improvements, tooling changes, or automated repair mechanisms.

Dashboard integration. The RFM datasets and clustering outputs can feed into monitoring dashboards (e.g., Grafana) to visualize the evolution of priority categories over time. This enables teams to detect emerging issues (categories with rising recency, frequency, or delays) before they become critical.

Alerting on priority shifts. By comparing successive FlakeRanker outputs, organizations can trigger alerts when previously low-priority categories migrate to higher-priority clusters, signaling potential infrastructure degradation or new systemic issues.

Customizing FlakeRanker for Organizational Context. The default RFM model treats Recency, Frequency, and Monetary dimensions equally, but organizations may have different priorities. A cost-sensitive organization might weight the Monetary dimension more heavily to focus on categories with the greatest negative impact on delays and costs, while a company prioritizing developer experience might emphasize Frequency to address the most commonly encountered frustrations. On the other hand, organizations avoiding technical debt may

favor Recency to resolve currently active issues before they compound. FlakeRanker can accommodate these variations through command-line options that allow users to adjust both RFM weights and underlying formula parameters. For instance, the Monetary calculation depends on infrastructure cost estimates (e.g., hourly compute rates) that vary across organizations; these can be configured to reflect actual operational costs rather than relying on defaults. This flexibility enables teams to tailor the prioritization output to their specific context without modifying the tool's source code.

Limitations of FlakeRanker. Several practical limitations affect the deployment and long-term use of FlakeRanker. The labeling module relies on regex patterns derived from our analysis, which inherently cannot anticipate novel failure types. Failures not matching existing patterns are labeled as "unknown" and require manual investigation to extend the taxonomy. In our dataset, approximately 14% of flaky job failures remained uncategorized, representing a blind spot that organizations should periodically sample and analyze. Moreover, as CI/CD tooling evolves, existing patterns may become stale: a regex matching error message from one version of a container registry client may fail silently after an upgrade. This maintenance burden is an unavoidable consequence of rule-based approaches, though future work on machine learning-based classification (i.e., extending the few-shot learning approach from Chapter 4 to multi-class classification) can address this problem. A related constraint is the cold start for emerging failure categories. The RFM model requires historical data to compute meaningful scores; a newly appearing failure type typically has limited to no frequency or cost history, and thus cannot be prioritized until it accumulates sufficient occurrences. Organizations using FlakeRanker should complement our tool with manual review of very recent "unknown" failures to better anticipate emerging issues.

5.6.3 Practical Implications

CI providers and organizations facing unreliable pipeline failures can adopt strategic prioritization rather than addressing flaky job failures reactively. The 14 priority categories identified in this study account for a disproportionate share of waste, and focusing resolution

efforts on these categories will yield the greatest return. However, flaky job failure patterns are dynamic, meaning that categories that are dormant today may resurface, and new categories can emerge. Continuous monitoring with tools like FlakeRanker enables proactive management rather than perpetual firefighting.

For TELUS, two categories warrant specific attention. Given the prominence of misconfigured environment variables, teams should consider implementing stricter change management and validation for CI/CD variable configurations, particularly in hierarchical project structures where variables are inherited. For timeout-related issues, we recommend pushing the diagnosis of timeout errors beyond duration limits to distinguish between different underlying causes instead of applying blanket timeout increases.

This work opens several avenues for future research, particularly in extending the proposed taxonomy and developing ML-based approaches that generalize beyond predefined regex patterns. We invite researchers to validate and extend our failure category taxonomy across diverse organizational contexts to establish a more comprehensive catalogue. The labeled data produced by FlakeRanker can serve as training data for multi-class classification models that generalize beyond predefined patterns and require less maintenance than rule-based approaches (addressed in Chapter 6). Furthermore, the priority categories identified here warrant deeper investigation into their contributing factors (e.g., disentangling the diverse root causes behind timeout failures could enable more targeted mitigations). Finally, connecting diagnosis to category-specific automated repair remains an open challenge that future work should address to close the loop toward fully autonomous self-healing of pipeline reliability issues.

5.7 Threats to Validity

Internal Validity. Internal threats to validity are related to the representativeness of the data collected. To mitigate these threats, we identified key projects where developers are most active and collected flaky job failure data from the entire history of these projects. However, approximately 30% of the flaky failure logs were unavailable. Although TELUS engineers

confirmed that the logs were lost due to a migration of GitLab, analyzing flaky failures executed during this period of significant change would enable us to better understand the evolution of issues and draw more accurate conclusions.

Construct Validity. The primary threat to construct validity lies in the correctness of the identified flaky job failure categories. To mitigate this, we used open coding to label a representative sample of logs manually, collaborated with TELUS engineers to resolve disagreements, and developed a regex-based labeling tool that achieved $\approx 82\%$ precision on an unseen sample. This tool is further refined before its use on the dataset. Ultimately, engineers confirmed that the identified categories align with their practical experiences. The second threat concerns the estimation of diagnosis costs (RQ2). Lead time, from the first failure to the final rerun, is used as a proxy of diagnosis time, though it may include idle periods like waiting times and weekends. While this may not perfectly represent diagnosis time, it is the most objective measure available. We accept this limitation since precise cost estimations are less critical for this study, which focuses on prioritization through comparative analysis.

External Validity. The threats to external validity relate to the generalizability of our findings. To mitigate these threats, we analyzed flaky job failures from 80 projects varying in size, age, purpose, and programming languages. In addition, TELUS extensively uses open-source and public tools that are widely used across organizations. However, TELUS has a CI environment prone to specific types of flaky failures due to its organization and the nature of its core software products (i.e., SDNs). Therefore, our results may not apply to very different project settings. We open-sourced the labeling and priorization tool (FlakeRanker[11]) along with scripts used to enable the replication of this study in other contexts.

5.8 Conclusion

One of the major challenges affecting CI is flaky job failures. This chapter presents a novel approach to identifying and prioritizing flaky job failure categories for automated diagnosis and

[11] https://github.com/devopsirc/telus-flaky-job-failures-prioritization

resolution. We developed a labeling tool and established a taxonomy of 46 flaky job failure categories identified in the context of TELUS projects. We then analyzed the Recency (R), Frequency (F), and estimated Monetary cost (M) associated with each of these categories. This analysis provides valuable insights into the root causes and how the different failure categories have evolved in the CI system at TELUS. For instance, we observe that about 40% of the flaky job failures fall into the five most frequent categories and about 34% into the five most costly ones. Also, while some categories have only recently emerged, others have not surfaced for over a year. In addition, our findings outlined that R-F-M values are not correlated and should all be considered when prioritizing categories. Finally, we built an RFM clustering model and identified 14 priority categories for future efforts on automated diagnosis and repair and 14 others for monitoring as future potential threats.

CHAPTER 6

PREDICTING INTERMITTENT JOB FAILURE CATEGORIES FOR DIAGNOSIS USING FEW-SHOT FINE-TUNED LANGUAGE MODELS

Henri Aïdasso[1] , Francis Bordeleau[1] , Ali Tizghadam[2]

[1] Department of Software and IT Engineering, École de Technologie Supérieure,
1100 Notre-Dame Ouest, Montreal, Quebec, Canada H3C 1K3
[2] Network Softwarisation and AI, TELUS,
Toronto, Ontario, Canada

Article published in « ACM International Conference on the Foundations of Software Engineering, Industry Track » March 2026

6.1 Preamble

With priority categories identified, the next step is to automate the categorization of detected failures to accelerate triage and diagnosis. This chapter addresses this downstream task by introducing a few-shot multi-class classifier that predicts the priority category to which a detected intermittent failure belongs. This work has been accepted for publication in ACM FSE 2026 (Aïdasso, Bordeleau & Tizghadam, 2026). It builds on the 14 priority categories identified in Chapter 5 and extends the few-shot paradigm introduced in Chapter 4, from binary detection to multi-class classification using FlaXifyer[1].

6.2 Introduction

Intermittent failures root causes vary widely, from transient network issues and resource exhaustion to race conditions, non-deterministic tests, and infrastructure problems (Lampel *et al.*, 2021; Aïdasso *et al.*, 2025a). This diversity poses two challenges: developers waste computational resources repeatedly rerunning jobs expecting they will pass (Aïdasso *et al.*, 2025a; Olewicki *et al.*, 2022); and when reruns fail to resolve the issue, they face substantial

[1] https://github.com/ahenrij/intermittent-job-failure-diagnosis

diagnosis overhead, navigating lengthy logs to pinpoint root causes that often require coordination across teams (Lampel *et al.*, 2021; Olewicki *et al.*, 2022; Aïdasso *et al.*, 2025a).

Prior work has focused primarily on the first challenge: detecting intermittent job failures from regular ones to reduce wasteful reruns. For this purpose, Lampel *et al.* (2021) trained a binary classifier using job telemetry data from Mozilla's CI service. Since telemetry is not always available, Olewicki *et al.* (2022) proposed an approach using TF-IDF representations of job logs combined with rerun metrics. Moriconi *et al.* (2022) explored knowledge graphs with limited results, while substantial work has focused specifically on flaky tests (Lam *et al.*, 2019b; Parry, Kapfhammer, Hilton & McMinn, 2020; Bell *et al.*, 2018; Akli *et al.*, 2023; Alshammari *et al.*, 2021; Fatima *et al.*, 2024; Rahman & Shi, 2024), one cause of intermittent job failures. However, flaky tests account for only a small proportion of such failures (Aïdasso *et al.*, 2025a), whose root causes are far more diverse (Aïdasso *et al.*, 2025a; Lampel *et al.*, 2021; Durieux *et al.*, 2020), and any build job, not just test jobs, can fail intermittently (Lampel *et al.*, 2021; Olewicki *et al.*, 2022; Aïdasso *et al.*, 2025a).

The second challenge, that of diagnosing intermittent job failures, remains largely unaddressed (Aïdasso *et al.*, 2025a). While some transient issues (e.g., network outage) resolve upon job rerun, many failure categories, including infrastructure problems, DNS resolution errors, and credential issues, require substantial diagnosis effort (Lampel *et al.*, 2021; Olewicki *et al.*, 2022; Aïdasso *et al.*, 2025a). This overhead distracts developers from core activities and delays software releases (Aïdasso *et al.*, 2025a). At TELUS, the diagnosis of such failures often requires coordination across network, infrastructure, and application teams, extending resolution time to several days (Aïdasso *et al.*, 2025a).

There is therefore a pressing need for automated methods to categorize intermittent job failures and support diagnosis (Aïdasso *et al.*, 2025a; Lampel *et al.*, 2021). Our prior work (Aïdasso *et al.*, 2025a) contributed a catalogue of 46 failure categories at TELUS and an open-source automated labeling tool (Aïdasso, 2025b). However, this regex-based tool lacks semantic understanding,

cannot generalize to unseen log patterns, and requires manual error-prone maintenance, limiting its practical applicability.

To bridge this gap, we introduce FlaXifyer, an approach for predicting intermittent (*flaky*) job failure categories using few-shot fine-tuned language models. FlaXifyer requires only job execution logs to predict failure categories, making it practical for deployment. Given the heterogeneous nature of job logs (Brandt *et al.*, 2020), we compare two encoder types: BGE (Xiao *et al.*, 2024), a general-purpose text encoder, and CodeBERT (Feng *et al.*, 2020), a code-aware model. Both are fine-tuned using a few-shot learning technique (Tunstall *et al.*, 2022), proven effective under limited labeled data. We evaluate FlaXifyer on a labeled dataset of 2,458 intermittent job failures from industrial projects at TELUS (Aïdasso *et al.*, 2025a) and propose LogSift, a technique for identifying influential log statements to support interpretability. This study addresses the following research questions (RQs):

RQ_1. How well does FlaXifyer predict intermittent job failure categories? Evaluated across different shot settings (2-16) and encoder types (BGE vs. CodeBERT), FlaXifyer achieves 84.3% Macro F1 and 92.0% Top-2 accuracy with BGE at 12 shots, outperforming CodeBERT by 8.1 percentage points. Performance plateaus around 10-12 shots, and general-purpose encoders prove more effective than code-aware models for heterogeneous job logs.

RQ_2. How does model performance evolve as new failure categories are introduced? We assess FlaXifyer's scalability by incrementally extending from 8 core categories to 13. Macro F1 decreases modestly from 92.5% to 84.3%, with most core categories remaining stable. Per-class F1 ranges from 68.4% to 98.8%, revealing categories that are easy (e.g., `runner pod waiting timeout`) and difficult (e.g., `host resolution failure`) to classify.

RQ_3. Can we determine the most influential log statements to support interpretability? We evaluate LogSift, our interpretability technique for identifying influential log statements. LogSift achieves 74.4% mean reduction ratio with 63.1% of its outputs contain 30 lines or fewer. Qualitative analysis confirms that 87% of identified log segments are relevant for diagnosis, surfacing actionable failure information in under one second.

In summary, this chapter makes the following contributions:

- **FlaXifyer**, a few-shot learning approach for predicting intermittent job failure categories using pre-trained language models, achieving 84.3% F1-score.
- **LogSift**, an interpretability technique for identifying influential log statements to support diagnosis.
- **Replication package** (Aïdasso, 2026), including implementation scripts and intermediary results, for verification and replication.

The remainder of this chapter is organized as follows. Section 6.3 presents background and related work. Section 6.4 details FlaXifyer, our approach for predicting intermittent job failure categories. It further presents LogSift, our interpretability technique to identify influential log statements. Section 6.5 describes the study methodology, and Section 6.6 reports experimental results. Section 6.7 discusses findings and implications. Finally, Section 6.8 presents threats to validity, and Section 6.9 concludes the chapter with future directions.

6.3 Background and Motivation

Job Logs. Each CI job execution follows a structured sequence of steps, producing heterogeneous logs. Common steps, such as runner environment setup, repository cloning, dependency installation, artifact caching, and cleanup, are shared across virtually all jobs, while job-specific steps execute the commands defined for that particular task (e.g., compilation, testing, image building, deployment). Each step produces textual logs, resulting in extensive and highly heterogeneous job execution logs that mix natural language, shell commands and outputs, code snippets, error stack traces, package installation traces, and tool-specific outputs (Brandt *et al.*, 2020). Developers rely on these job logs for traceability and failure diagnosis, though their verbosity and heterogeneity make manual inspection challenging (Olewicki *et al.*, 2022; Aïdasso *et al.*, 2025a).

Motivation for Automated Failure Categorization. Diagnosing intermittent job failures is particularly time-consuming, as root causes often lie outside developers' domain of responsibility

(Olewicki *et al.*, 2022; Aïdasso *et al.*, 2025a). At TELUS, the diagnosis of such failures is further complicated by organizational complexity. For instance, failures involving infrastructure-managed resources (e.g., deployment tokens, network devices, container registry) require coordination across multiple teams. Some logs even contain instructions such as *"... please contact infrastructure team..."* buried within thousands of lines that developers must review manually. Automated failure categorization addresses these challenges in three ways: (1) helping developers quickly understand the nature of the issue they face, (2) enabling automated triage by directing failures to the appropriate team, and (3) serving as a prerequisite for category-specific automated repair strategies. Complementing categorization with relevant log segments further accelerates diagnosis by surfacing failure-specific information that can accompany triage notifications, reducing the time teams spend identifying root causes for resolution.

Intermittent Job Failure Categories. In prior work (Aïdasso *et al.*, 2025a), we analyzed intermittent job failures at TELUS and introduced a comprehensive catalogue of 46 failure categories, along with a regex-based labeling tool (Aïdasso, 2025b). Using RFM analysis (Birant, 2011), we identified 14 priority categories that warrant attention for automated diagnosis support. While essential for obtaining labeled data, the existing regex-based tool (Aïdasso, 2025b) has significant limitations for automated categorization. First, handcrafted regex patterns do not account for semantic or structural context and can miss even slight log variations. Second, maintaining the tool is tedious and error-prone, requiring manual updates whenever log patterns change. TELUS engineers emphasized the need for an automated solution that scales reliably across projects and remains easy to maintain.

In summary, existing approaches focus on detecting intermittent failures to reduce wasteful job reruns, but leave the diagnosis challenge largely unaddressed. This study presents the first approach for *predicting intermittent job failure categories* to support practitioners in the diagnosis and repair of such failures.

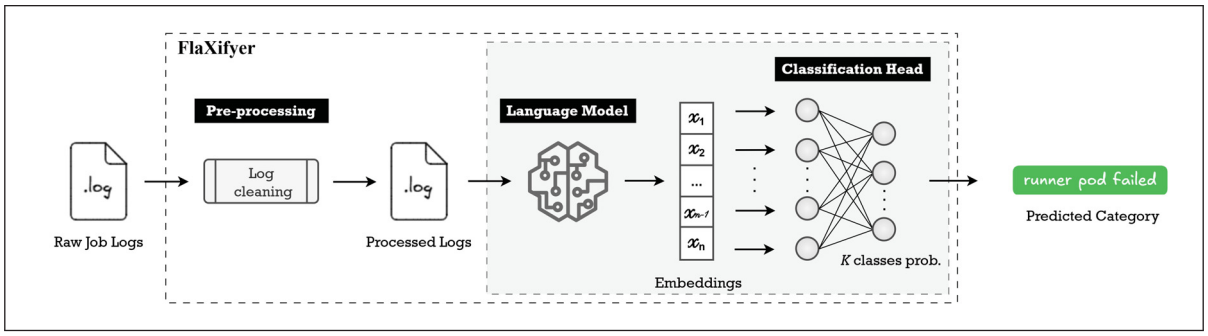


Figure 6.1 Predicting intermittent job failure categories from job logs using a language model and classification head

6.4 FlaXifyer: Predicting and Interpreting Intermittent Job Failure Categories

6.4.1 Overview

FlaXifyer addresses the diagnosis challenge through *multi-class classification* of intermittent job failure categories. We name our approach FlaXifyer to emphasize its ability to classify intermittent (a.k.a *flaky*) job failures. Unlike prior works on flaky tests (Akli *et al.*, 2023; Fatima, Ghaleb & Briand, 2023; Fatima *et al.*, 2024), which address that specific subset of job-related issues, FlaXifyer targets the broader challenge of categorizing diverse flaky job failures across the CI/CD pipeline. Given a job's execution logs, it predicts the failure's root cause category (e.g., `runner pod failure`, `docker daemon connection failure`, `job execution timeout`), enabling developers to quickly route issues to the appropriate team or, in the future, apply category-specific automated repair strategies.

Figure 6.1 illustrates the FlaXifyer pipeline, which comprises three stages: (1) pre-processing raw job logs to reduce noise and size, (2) generating numerical embeddings using a fine-tuned language model (specifically, a text encoder), and (3) predicting the failure category by classifying the embeddings produced in stage (2). FlaXifyer is designed to be practical. It requires only job execution logs as input and leverages few-shot learning to minimize the manual labeling effort for training. At prediction time, given a raw log file, it outputs the predicted failure

category c as:

$$c = C(E(P(\log))) \tag{6.1}$$

where $P(\cdot)$ denotes the log pre-processing function, $E(\cdot)$ the fine-tuned encoder, and $C(\cdot)$ the classification head.

The following subsections detail each stage of implementation.

6.4.2 Log Pre-processing

Job logs are inherently heterogeneous, verbose, and partially repetitive () making them susceptible to the curse of dimensionality when represented as vectors. Moreover, the input length of language models used in the embedding stage is limited to 512 tokens. Pre-processing is therefore essential to reduce log size while preserving information critical for identifying the failure category. Hence, we apply the set of pre-processing rules defined in Section 4.4.1. As a result, these rules reduce raw log size by an average of 61%, which is consistent with prior work (Aïdasso *et al.*, 2025b; Olewicki *et al.*, 2022).

6.4.3 Few-Shot Fine-Tuning

The first step of FlaXifyer's training process involves few-shot fine-tuning of pre-trained embedding models to produce representations that clearly separate intermittent failure categories. Prior work on intermittent job failure detection (Aïdasso *et al.*, 2025b) has shown that fine-tuned embedding models significantly outperform TF-IDF representations for binary job log classification, motivating our choice of embedding models for this multi-class classification task.

To this end, we use the SetFit framework (Tunstall *et al.*, 2022), which enables efficient, prompt-free fine-tuning using only a small number of labeled examples. This capability is essential, as manually annotating job failures with their root-cause categories is costly and time-consuming. While our regex-based labeling tool (Aïdasso, 2025b) can generate an initial labeled dataset, it lacks semantic understanding, does not generalize well to log variations, and requires substantial

(error-prone) manual maintenance to support new failure categories. Few-shot fine-tuning addresses these limitations by learning semantic representations from minimal labeled data.

Specifically, SetFit fine-tunes the pre-trained embedding model using contrastive learning (Koch *et al.*, 2015) to learn a discriminative embedding space. It generates a dataset of triplets from the training set, including (1) *positive triplets* $\{(x_i, x_j, 1)\}$, where log pairs are sampled from the *same* failure category; and (2) *negative triplets* $\{(x_i, x_j, 0)\}$, where pairs are sampled from *different* categories. Using a cosine similarity loss, the encoder learns to position logs from the same category closer together in the embedding space while pushing logs from different categories farther apart. This contrastive objective is well-suited for multi-class problems, as it learns pairwise relationships that generalize across all categories.

Once fine-tuned, the encoder generates embeddings for all training logs, which are then used to train the classifier (see Section 6.4.4 below).

6.4.4 Classification Head

The second training stage fits a classifier on the embeddings produced by the fine-tuned encoder. We use Logistic Regression with multinomial loss as the classification head, mapping 768-dimensional embeddings to failure categories. Because contrastive fine-tuning produces well-separated embeddings, this classifier is both fast to train and effective under limited labeled data, as underlined in prior work (Aïdasso *et al.*, 2025b) and per SetFit's recommended configuration (Tunstall *et al.*, 2022). The classifier outputs a probability distribution over the K failure categories.

At inference time, given a raw job log, FlaXifyer first applies log pre-processing P, then encodes the processed log using the fine-tuned encoder E, and finally predicts a failure category via the trained classifier C. Formally, the predicted category c is given by:

$$c = \underset{k \in \{1,\ldots,K\}}{\arg\max} \left[C\big(E(P(\log))\big) \right]_k \tag{6.2}$$

6.4.5 Interpretability: Identifying Influential Log Statements

Motivation. While accurate prediction of failure categories supports diagnosis, practitioners also need to understand *why* a particular category was predicted. Interpretability is essential for building trust in automated classification systems and enabling developers to verify predictions against their domain knowledge (Ribeiro, Singh & Guestrin, 2016). To address this need, we introduce LogSift, a technique for identifying the log statements most influential to the classifier's decision.

Approach. Our approach draws inspiration from binary search (Knuth, 1998), recursively partitioning the log to isolate the minimal segments that preserve the predicted category. The intuition is that if a log segment contains sufficient information for the classification pipeline to predict the same category as the full log, then the influential statements lie within that segment. Algorithm 6.1 describes the procedure. Given the statements L of a job log and the original category c predicted from the full log, we recursively bisect L into top and bottom halves. When the number of statements is odd, the bottom half receives the extra statement. For each half, we query the classification pipeline f: if the predicted category matches c, we continue the search within that half; otherwise, we backtrack. When both halves independently predict c, influential information is distributed across both segments, and we recursively search each half, aggregating the results. The recursion terminates when the segment size falls below a minimum threshold τ (set to 2 statements to ensure sufficient context) or when neither half preserves the original prediction. In the latter case, the current segment is the minimal chunk containing the influential statements. For practicality, we return a list of (`start`, `end`) index ranges identifying the influential statements, which can then be used to highlight or extract the corresponding log segments for developer review. The proposed algorithm has logarithmic complexity $O(\log n)$ in the number of log statements n when influential information is localized, and linear complexity $O(n)$ in the worst case when information is distributed throughout the log. In practice, important failure-related statements tend to cluster near error messages, yielding efficient execution even for verbose logs.

Algorithm 6.1 LogSift: Influential Log Statements Identification

Require: L: Raw job log as a list of log statements.
Require: f: FlaXifyer classification pipeline
Require: τ: minimum segment size (default: 2)
Ensure: R: list of (start, end) index ranges

Function `LogSift(`L, f, τ`):`
  $c \leftarrow f(L)$ `// Original predicted category`
  return `FindInfluential(`L, c, f, τ, 0`);`
Function `FindInfluential(`L, c, f, τ, *offset*`):`
  if $|L| \leq \tau$ **then**
    return $[(\text{offset},\ \text{offset} + |L| - 1)]$;
  end if
  $m \leftarrow \lfloor |L|/2 \rfloor$;
  $L_{top} \leftarrow L[0 : m]$;
  $L_{bot} \leftarrow L[m : |L|]$ `// Larger half if` $|L|$ `is odd`
  $match_{top} \leftarrow f(L_{top}) = c$;
  $match_{bot} \leftarrow f(L_{bot}) = c$;
  if $match_{top}$ ***and*** $match_{bot}$ **then**
    return `FindInfluential(`L_{top}, c, f, τ, *offset*`)` $\cup$ `FindInfluential(`L_{bot}, c, f, τ, *offset* $+ m$`);`
  end if
  else if $match_{top}$ **then**
    return `FindInfluential(`L_{top}, c, f, τ, *offset*`);`
  end if
  else if $match_{bot}$ **then**
    return `FindInfluential(`L_{bot}, c, f, τ, *offset* $+ m$`);`
  end if
  else
    return $[(\text{offset},\ \text{offset} + |L| - 1)]$ `// Minimal segment preserving` c
  end if

Practical Benefits. This interpretability technique offers several advantages for practitioners. First, it highlights the specific log statements that drove the prediction, allowing developers to quickly verify whether the classification is reasonable. Second, it reduces the cognitive load of manually inspecting lengthy logs by directing attention to the most relevant segments, which include valuable specific information to accelerate the resolution process. Third, when the model makes errors, examining the influential segments can reveal whether the misclassification stems from ambiguous log patterns or model limitations, informing future improvements.

6.5 Study Design

6.5.1 Research Questions

The primary goal of this study is to investigate the effectiveness of FlaXifyer in predicting intermittent job failure categories (RQ_1), its scalability as new categories emerge (RQ_2), and its interpretability for practical use (RQ_3). Therefore, we address the following RQs:

- RQ_1. How well does FlaXifyer predict intermittent job failure categories?
- RQ_2. How does model performance evolve as new failure categories are introduced?
- RQ_3. Can we determine the most influential log statements to support interpretability?

6.5.2 Dataset

Data Source. We build upon the dataset from our prior study on intermittent job failure categories at TELUS (Aïdasso *et al.*, 2025a). This dataset comprises 4,511 labeled intermittent job failures extracted from 80 industrial GitLab projects spanning 6.5 years of build history, from February 2018 to July 2024. The projects vary in size, maturity, and purpose (including software-defined networks, data analytics, APIs, and web portals) and cover 13 programming languages. Each instance is labeled with one of 46 root cause categories identified through semi-automated labeling using a regex-based tool, achieving 91% precision on an unseen representative sample (Aïdasso *et al.*, 2025a).

Category Selection. Multi-class classification with 46 categories is impractical. It presents challenges related to class imbalance and label sparsity, where many categories contain insufficient examples for effective model training. To address this, we focus on the 14 priority categories identified in our prior work through RFM analysis (Aïdasso *et al.*, 2025a). These categories represent the most recent, frequent, and costly failures, i.e., those causing the greatest waste of developer time and CI resources at TELUS. Moreover, they alone account for more than half (55%) of the labeled 4,511 intermittent job failures. Of the 14 priority categories, we exclude `cloud_token_limit_exceeded` (39 examples) due to insufficient samples for

Table 6.1 Selected priority failure categories ranked by RFM priority: top (rank 1-2), high (rank 3-8), medium (rank 9-13)

Rank	Category	Count	%
1	misconfigured_env_variable	673	27.4
2	job_execution_timeout	306	12.4
3	dependency_installation_failure	124	5.0
4	runner_pod_waiting_timeout	199	8.1
5	api_gateway_deployment_error	161	6.5
6	container_registry_server_error	213	8.7
7	git_transient_error	113	4.6
8	flaky_ui_test	179	7.3
9	external_file_invalid_format	125	5.1
10	host_resolution_failure	91	3.7
11	runner_image_pull_failure	99	4.0
12	remote_call_timeout	76	3.1
13	helm_resource_error	99	4.0
	Total	**2,458**	**100.0**

few-shot training at higher shot settings, as detailed in Section 6.5.3. Table 6.1 presents the 13 categories selected, ranked by RFM priority.

Data Labeling and Limitations. Labels were obtained using the regex-based tool (Aïdasso, 2025b) from our prior work (Aïdasso *et al.*, 2025a). While effective for dataset construction, this tool has inherent limitations: regex patterns lack semantic understanding and fail to capture log variations, resulting in incomplete coverage. Manual review revealed that the tool missed approximately 15% of categorizable failures due to minor variations (e.g., `out-of-memory` vs. `insufficient space`). Furthermore, extending the tool to new categories requires substantial manual effort to craft and maintain patterns. These limitations motivate our machine learning approach and the use of few-shot learning, which enables high-quality manual labeling of a small number of examples rather than relying on erroneous automated labeling at scale.

6.5.3 Experimental Setup

We adopt the evaluation methodology from our prior work on binary failure classification (Aïdasso *et al.*, 2025b), extending it to multi-class classification across intermittent failure categories.

Few-Shot Training Configuration. Following SetFit (Tunstall *et al.*, 2022) requirements, the training set TR is constructed by randomly sampling N log examples for each of the K failure categories, yielding a training set of size $N \times K$. In line with recommendations for few-shot text classification, we use Logistic Regression as the classification head (Tunstall *et al.*, 2022). The choice of the underlying embedding model is treated as a separate design decision and is investigated in RQ_1, where we analyze its impact on classification performance.

Cross-Validation Pipeline. To evaluate model performance under a fixed few-shot regime, we use Monte Carlo Cross-Validation (MCCV) (Simon, 2007), presented in Algorithm 4.1.

Each MCCV iteration i proceeds as follows. First, the input dataset D is stratified and randomly split into a learning set L_i (25%), a validation set V_i (25%), and a test set T_i (50%). From L_i, we then sample N labeled examples per failure category to form the few-shot training set TR_i. Model hyperparameters are optimized using V_i, and the resulting best model is evaluated on T_i to obtain the model performance estimate ε_i. Allocating a larger portion of the dataset D to the test set T_i improves the reliability of performance estimates, while the learning set remains sufficiently large to support few-shot sampling. We set $i_{\max} = 30$ independent iterations, and the final model performance is computed as:

$$\bar{\varepsilon} = \frac{1}{i_{max}} \sum_{i=1}^{i_{max}} \varepsilon_i \tag{6.3}$$

Category exclusion. For reliable N-shot sampling across MCCV iterations, each category must contain sufficient examples in the learning set. With a 25% learning split, categories require at least $4N$ total examples to consistently support N-shot training. Among the 14 previously identified priority categories (Aïdasso *et al.*, 2025a), the failure category

`cloud_token_limit_exceeded` (39 examples) does not meet this threshold under higher shot settings evaluated in RQ_1 ($39 < 64$ examples required for 16-shot training) and is therefore excluded. The 13 remaining categories contain between 76 and 673 examples each (Table 6.1), ensuring adequate support for the experiments.

Hyperparameter Optimization. We tune model training hyperparameters using Optuna (Akiba, Sano, Yanase, Ohta & Koyama, 2019) with 5 trials per model configuration, based on a recent study and fine-tuning guidelines (Pareja *et al.*, 2024; Aïdasso *et al.*, 2025b). During each trial, hyperparameter values are sampled from the following hyperparameter space:

- *Body learning rate*: 10^{-6} to 10^{-3}
- *Number of epochs*: 1 or 2
- *Batch size*: 2, 4, or 8
- *Max iter*: 50 to 300, by 50

The hyperparameter values yielding the highest performance on the validation set are selected for final evaluation on the test set.

Computational Environment. Model training was performed on NVIDIA A100 GPUs (40GB), with training time ranging from under 1 minute to 70 minutes per iteration (median: 3.3 minutes). LogSift evaluation and RQs analysis were executed on CPU nodes with 32GB RAM. All experiments used Python 3.11.

6.5.4 Evaluation Metrics

We evaluate FlaXifyer using metrics suited for multi-class classification with imbalanced class distributions, since our category frequencies range from 76 to 673 examples (Table 6.1). All metrics are aggregated across the MCCV iterations and reported as mean ± standard deviation.

Primary Metric. We use **Macro F1-score** as the primary evaluation metric. Macro F1 is the unweighted mean of per-class F1-scores, computed as Macro F1 $= \frac{1}{K} \sum_{k=1}^{K} F1_k$, where K denotes the number of categories and $F1_k = \frac{2\,\mathrm{TP}}{2\,\mathrm{TP}+\mathrm{FP}+\mathrm{FN}}$ is the F1-score for category k. By

treating all categories equally, regardless of frequency, Macro F1 penalizes models that perform well on frequent categories but poorly on rare ones. This aligns with our practical objective of supporting diagnosis as even less frequent failures can incur substantial diagnosis cost when they occur (Aïdasso *et al.*, 2025a).

Secondary Metrics. To provide a comprehensive view of model performance, we report traditional metrics including *precision* ($\frac{TP}{TP+FP}$) and *recall* ($\frac{TP}{TP+FN}$), along with the following:

Matthews Correlation Coefficient (MCC). MCC provides a balanced measure of classification quality that accounts for all entries in the confusion matrix (Matthews, 1975). As a result, it produces high scores only when the model performs well across all categories. This score ranges from -1 (complete disagreement) to $+1$ (perfect prediction), with 0 indicating random performance.

Top-k Accuracy. The proportion of instances where the correct category appears among the model's top k predictions. We report Top-2 and Top-3 accuracy to assess FlaXifyer's practical utility in scenarios where presenting multiple candidate categories is acceptable. Even when the top prediction is incorrect, suggesting the correct category among the top few options can substantially reduce diagnosis time by narrowing the scope of investigation.

6.6 Results

6.6.1 RQ_1. How well does FlaXifyer predict intermittent job failure categories

6.6.1.1 Motivation

Identifying the optimal shot setting and encoder type is essential for practical deployment. On the one hand, identifying the optimal number of shots determines how many examples must be manually labeled to train an effective classifier and, importantly, to extend it with new failure categories in the future. On the other hand, job logs are heterogeneous, containing natural language messages, code fragments, stack traces, and tool-specific outputs. Whether general-

purpose or code-aware pre-trained embedding models better capture this heterogeneity remains unclear. We therefore evaluate our approach, comparing two encoders: BGE (general-purpose) and CodeBERT (code-aware), across shot settings ranging from 2 to 16 to determine the most effective configuration for predicting intermittent job failure categories using the job logs.

6.6.1.2 Approach

To evaluate FlaXifyer's classification performance, we conduct experiments varying two factors: (1) the pre-trained embedding model and (2) the number of shots per category.

Encoders. We compare two encoders with near parameter parity (109M vs. 125M), enabling controlled comparison:

- **BGE**: `BAAI/bge-base-en-v1.5` (109M parameters), a state-of-the-art general-purpose text encoder trained on large-scale web data using contrastive learning (Xiao *et al.*, 2024).
- **CodeBERT**: `microsoft/codebert-base` (125M parameters), a bimodal encoder pre-trained on programming languages and natural language (Feng *et al.*, 2020). CodeBERT has shown promise for flaky test classification tasks (Akli *et al.*, 2023), making it a natural candidate for logs containing test/code snippets.

Shot settings. We evaluate $N \in \{2, 4, 8, 10, 12, 16\}$ shots per category, yielding training sets ranging from 26 to 208 examples with $K = 13$ categories. The values 2, 4, 8, and 16 follow standard few-shot learning conventions (Tunstall *et al.*, 2022; Akli *et al.*, 2023), while 10 and 12 provide finer granularity in the intermediate range.

For each encoder-shot combination, we train and evaluate models using the MCCV procedure described in Section 6.5.3, resulting in 2 encoders $\times$ 6 shot settings $\times$ 5 hpp search trials $\times$ 30 iterations = 1800 trained models. We report Macro F1 as the primary metric, along with MCC, and Top-k accuracy scores.

Table 6.2 Model performance across shot settings for two encoder families. Values are reported as mean_{std} over seeds (%)

	CodeBERT							BGE						
#Shots	f1	mcc	precision	recall	top-1	top-2	top-3	f1	mcc	precision	recall	top-1	top-2	top-3
2	$52.9_{5.4}$	$49.9_{6.1}$	$57.3_{4.5}$	$55.6_{5.1}$	$53.7_{6.2}$	$64.1_{6.3}$	$70.1_{6.2}$	$66.1_{6.2}$	$64.7_{6.4}$	$69.1_{6.5}$	$69.6_{5.2}$	$67.5_{6.3}$	$76.1_{6.3}$	$80.7_{6.3}$
4	$62.3_{3.9}$	$60.3_{4.5}$	$64.0_{3.9}$	$65.6_{3.7}$	$63.6_{4.6}$	$73.2_{4.8}$	$79.1_{4.7}$	$75.3_{4.2}$	$74.4_{4.9}$	$77.1_{4.1}$	$78.5_{3.4}$	$76.5_{4.8}$	$83.5_{5.0}$	$88.1_{4.8}$
8	$71.2_{3.5}$	$69.4_{4.9}$	$71.5_{3.5}$	$74.0_{2.7}$	$72.3_{4.9}$	$81.7_{4.3}$	$86.2_{4.1}$	$82.2_{3.7}$	$81.5_{4.8}$	$83.6_{3.8}$	$84.5_{2.8}$	$83.1_{4.7}$	$89.9_{3.8}$	$93.3_{3.7}$
10	$73.8_{2.9}$	$72.3_{4.2}$	$73.7_{3.0}$	$76.6_{2.3}$	$75.0_{4.2}$	$84.4_{3.6}$	$88.3_{3.5}$	$83.7_{2.7}$	$83.2_{4.0}$	$84.9_{2.6}$	$85.7_{2.2}$	$84.7_{3.9}$	$90.9_{3.5}$	$94.4_{2.9}$
12	$76.2_{2.3}$	$75.0_{3.5}$	$75.9_{2.3}$	$78.4_{1.8}$	$77.6_{3.5}$	$86.9_{2.0}$	$90.2_{2.2}$	$\mathbf{84.3_{1.9}}$	$\mathbf{83.8_{2.9}}$	$\mathbf{85.4_{2.0}}$	$\mathbf{86.3_{1.8}}$	$\mathbf{85.3_{2.8}}$	$\mathbf{92.0_{2.9}}$	$\mathbf{95.3_{2.5}}$
16	$77.6_{2.4}$	$76.7_{3.1}$	$77.1_{2.7}$	$79.8_{1.9}$	$79.1_{3.1}$	$88.0_{2.6}$	$91.1_{2.3}$	$85.4_{2.8}$	$85.1_{4.1}$	$86.6_{2.6}$	$87.1_{2.2}$	$86.4_{4.0}$	$93.3_{3.0}$	$95.8_{2.9}$

6.6.1.3 Results

Table 6.2 presents model performance across both encoders and the six few-shot settings.

FlaXifyer achieves 84-85% Macro F1 at 12-16 shots with BGE, which consistently outperforms CodeBERT across all shot settings, suggesting that general-purpose encoders are better suited for job logs. At 12 shots, BGE reaches 84.3% Macro F1 compared to 76.2% for CodeBERT: a difference of 8.1 percentage points. This gap persists across all metrics: MCC (83.8% vs. 75.0%), precision (85.4% vs. 75.9%), recall (86.3% vs. 78.4%), and Top-2 accuracy (92.0% vs. 86.9%). Unlike flaky test classification, where code-aware models have shown promise (Akli *et al.*, 2023), intermittent job failure logs prove to be better represented with general-purpose language models. This result underlines that job logs are not dominated by code or documentation fragments. Instead, they contain a heterogeneous mix of infrastructure messages, error traces, and tool-specific outputs that general-purpose encoders seem to capture more effectively.

With BGE, performance plateaus around 10-12 shots, with 12 shots achieving the best trade-off between performance, stability, and labeling effort. Using the BGE encoder, F1-score increases from 66.1% at 2 shots to 82.2% at 8 shots, then stabilizes: 83.7% at 10 shots, 84.3% at 12 shots, and 85.4% at 16 shots. The improvement from 12 to 16 shots (1.1 percentage points) is not statistically significant based on the Wilcoxon signed-rank test ($p > 0.05$). Moreover, variance decreases as shots increase up to 12, with standard deviation

dropping from 6.2% at 2 shots to 1.9% at 12 shots. Overall, this suggests that 12 shots represent a practical labeling budget for deploying FLAXIFYER with high and stable performance.

Top-2 accuracy reaches 92.0% at 12 shots, indicating strong practical utility even when the top prediction is incorrect. With BGE at 12 shots, the correct category appears among the top two predictions in 92.0% of cases. This metric provides particularly relevant information for practical diagnosis workflows: when developers are presented with two candidate categories, they can quickly verify against the log content to identify the root cause, and route the failure to the appropriate team or directly proceed with its resolution when possible.

6.6.2 RQ_2. How does model performance evolve as new failure categories are introduced

6.6.2.1 Motivation

In practice, new failure categories emerge as CI/CD environments evolve. At TELUS, infrastructure changes, new tools are adopted, and previously rare issues become prevalent (Aïdasso *et al.*, 2025a). A practical classification model must accommodate new categories without significant degradation on existing ones. This question evaluates FLAXIFYER's scalability by incrementally extending the classifier from 8 core categories to 13, and examines per-class performance to identify which categories are easy or difficult to classify. These findings will inform practitioners about the expected trade-offs when extending the model and highlight categories that may require additional attention.

6.6.2.2 Approach

Our prior study (Aïdasso *et al.*, 2025a) grouped the 13 categories by RFM priority rank. Ranks 1-8 constitute top/high priority categories (representing the most recent, frequent, and costly failures) which require immediate attention, while ranks 9-13 are medium priority categories representing natural candidates for incremental consideration. Hence, we simulate extending a deployed classifier by training models with increasing numbers K of categories:

- $K = 8$: Top and high priority categories only (ranks 1-8)
- $K = 10$: Core categories + 2 medium-priority (ranks 1-10)
- $K = 13$: All 13 selected categories

This design reflects the practical scenario where an organization deploys a classifier for the most critical failures first, then progressively extends coverage to additional categories.

Based on RQ_1 results, we select the best-performing encoder and shot setting (BGE, 12 shots). For each category subset, we train models independently from the base pre-trained encoder using the MCCV procedure described in Section 6.5.3.

We report Macro F1 to assess overall performance trends as K increases. To examine whether adding medium-priority categories degrades performance on the core categories, we compare per-class F1-score ($\overline{F1}_k$) for core categories (ranks 1-8) across all three configurations of K. We also report per-class F1-score for newly added categories to assess how quickly they reach stable performance.

6.6.2.3 Results

Table 6.3 compares per-class F1-scores as the classifier is extended from $K = 8$ core to $K = 13$ total categories.

FlaXifyer scales gracefully, with Macro F1 decreasing from 92.5% to 84.3% when extending from 8 to 13 categories. This 8.2 percentage point decrease is expected as adding categories increases classification complexity. Notably, the transition from K=10 to K=13 incurs minimal degradation (85.6% → 84.3%), suggesting that the primary complexity increase occurs when first introducing medium-priority categories. To understand this degradation, we examine per-class performance across configurations.

Most core categories remain stable as new categories are introduced, though two show notable degradation. Five of the eight core categories maintain F1-scores within 2 percentage points across all K: `runner pod waiting timeout` (98.6% → 98.8%), `flaky`

Table 6.3 Per-class F1-score across category set sizes K

Category	K=8	K=10	K=13
misconfigured_env_variable	$93.5_{3.4}$	$89.6_{8.4}$	$85.4_{7.3}$
job_execution_timeout	$85.0_{9.3}$	$86.7_{8.3}$	$84.6_{8.9}$
dependency_installation_failure	$92.6_{6.1}$	$91.6_{9.0}$	$92.4_{2.9}$
runner_pod_waiting_timeout	$98.6_{1.9}$	$98.2_{2.0}$	$98.8_{1.6}$
api_gateway_deployment_error	$92.0_{3.9}$	$79.3_{10.0}$	$77.8_{8.9}$
container_registry_server_error	$83.5_{7.0}$	$83.4_{6.6}$	$82.5_{5.5}$
git_transient_error	$96.1_{2.8}$	$87.8_{3.1}$	$87.9_{2.7}$
flaky_ui_test	$98.6_{1.1}$	$97.6_{1.4}$	$97.3_{1.1}$
external_file_invalid_format	–	$73.5_{13.8}$	$74.1_{11.7}$
host_resolution_failure	–	$68.2_{6.1}$	$68.4_{5.8}$
runner_image_pull_failure	–	–	$95.4_{3.4}$
remote_call_timeout	–	–	$73.5_{7.8}$
helm_resource_error	–	–	$78.1_{19.8}$
Macro F1	**92.5**	**85.6**	**84.3**

`ui test` (98.6% → 97.3%), `dependency installation failure` (92.6% → 92.4%), `job execution timeout` (85.0% → 84.6%), and `container registry server error` (83.5% → 82.5%). However, `api gateway deployment error` drops from 92.0% to 77.8% (−14.2), and `git transient error` decreases from 96.1% to 87.9% (−8.2). Both categories share semantic overlap with newly added network-related categories (`host resolution failure`, `remote call timeout`), making discrimination more challenging.

At K=13, per-class F1-scores range from 68.4% to 98.8%, revealing distinct easy and difficult categories. The easiest categories exhibit distinctive log patterns: `runner pod waiting timeout` (98.8%), `flaky ui test` (97.3%), `runner image pull failure` (95.4%), and `dependency installation failure` (92.4%), all with low variance (between 1.1% and 3.4%). The most difficult categories are precisely those causing the degradation observed above: `host resolution failure` (68.4%), `remote call timeout` (73.5%), and `external file invalid format` (74.1%). These categories share ambiguous network-related log patterns with others, as discussed in Section 6.7. Notably, `helm resource error` achieves a moderate F1-score (78.1%) but exhibits the highest variance (19.8%), highlighting inconsistent failure patterns within this category.

Table 6.4 Relevance scoring for LogSift evaluation

Score	C1	C2	C3	Interpretation
High	✓	✓	✓	Explains prediction and actionable
Relevant	✓	✓	×	Explains prediction
Partial	✓	×	–	Failure signal only
Not Rel.	×	–	–	No failure information

6.6.3 RQ_3. Can we determine the most influential log statements to support interpretability

6.6.3.1 Motivation

Accurate predictions alone are insufficient for practical adoption: developers need to understand *why* a particular category was predicted to verify the classification, build trust in the system (Ribeiro *et al.*, 2016), and act on it efficiently. This is particularly challenging given the size of job logs, which range from 13 to 27,532 lines in our dataset (mean: 405). Manually sifting through hundreds or thousands of lines to locate failure-relevant information is impractical, especially when developers face multiple failures daily. Effective interpretability addresses both challenges simultaneously. By highlighting the log statements that drive a prediction, developers can (1) verify the classification *and* (2) immediately access the failure-specific details needed for repair, such as the URL for a `host resolution failure` or the variable name for a `misconfigured env variable`. To this end, we evaluate LogSift (Section 6.4.5), assessing whether it identifies meaningful failure indicators while substantially reducing log review effort.

6.6.3.2 Approach

We apply LogSift to a test set split (614 log instances) using the best-performing model (BGE, 12 shots). We set the minimum segment size $\tau = 2$ to ensure sufficient context for interpretation. Unlike classification, LogSift's utility does not depend on prediction correctness. In fact, developers will use highlighted log segments to verify predictions, whether correct or not.

Table 6.5 LogSift quantitative evaluation results

Metric	Value
Mean reduction ratio	74.4% ± 27.6%
30-consistency	63.1%
10-consistency	40.9%
2-consistency	12.0%
Mean execution time	0.98s

Therefore, we assess LogSift outputs along four dimensions: (1) **reduction ratio**, the average proportion of log statements eliminated while preserving the prediction; (2) ***n*-consistency**, the proportion of job instances where LogSift consistently reduces the log to n lines or fewer, measuring practical reviewability; (3) **execution time**, how quickly LogSift highlights relevant segments, informing efficiency compared to manual verification; and (4) **qualitative relevance**, whether identified segments contain meaningful failure indicators.

For qualitative analysis, we randomly sampled 100 jobs and assessed each corresponding LogSift output against three binary criteria: **C1** (failure indicator), contains error messages, exceptions, or stack traces; **C2** (prediction-explanatory), explains the prediction, whether by supporting a correct classification or revealing why a misclassification occurred; and **C3** (actionable), contains specific repair information such as URLs or variable names. Each output is assigned a relevance score following Table 6.4.

6.6.3.3 Results

Table 6.5 summarizes the quantitative evaluation results, while Fig. 6.5 presents the relevance assessment. Figures 6.2, 6.3, and 6.4 illustrate representative LogSift outputs, with each header showing the predicted failure category and reduction ratio.

LogSift effectively isolates influential statements, reducing log review effort by 74.4% on average while maintaining sub-second execution time. With a mean reduction ratio of 74.4%, LogSift reduces the average 405-line log to approximately 104 lines. In the best cases, such as the output example in Fig. 6.2 of a 359-line log, developers need only review 2 log

misconfigured_env_variable 359 → 2 lines

```
Building dependency release=broker-app, chart=app
in ./app.config.yaml: failed to render values files "app.values.
   yaml": failed to render [app.values.yaml], because of template:
   stringTemplate:5:15:...: required env var ‘IMAGE_NAME‘ is not
   set
```

Figure 6.2 LogSift reduces a 359-line log to 2 lines, directly identifying the missing environment variable (`IMAGE_NAME`)

api_gateway_deployment_error 155 → 8 lines

```
upload_zipfile_to_cloud_portal
test791.zip
curl: (26) Failed to open/read local data from file/application
zip file validate API is failed
200
Publish message to PubSub success
{"messageIds":["1403319032815554"]}200
section_end:1705665980:step_script
```

Figure 6.3 LogSift output for a misclassified job (true: `external_file_invalid_format`). Despite the incorrect prediction, the highlighted segment reveals the actual cause: a zip file read failure (`curl: (26)`) during upload, enabling developers to override the prediction

statements (99.4% reduction). More importantly, 30-consistency reaches 63.1%, meaning that nearly two-thirds of logs are reduced to 30 lines or fewer, representing a practical threshold for quick manual review. For 40.9% of log instances, outputs are 10 lines or fewer (a quick glance), and 12.0% are narrowed down to pinpoint 2-line outputs. Mean execution time of 0.98 seconds per log ensures LogSift can be integrated into CI/CD pipelines without introducing any significant latency.

Qualitative analysis confirms that LogSift identifies meaningful failure indicators in 87% of cases. As shown in Fig. 6.5, 69% of LogSift outputs received a *High* relevance score, while

host_resolution_failure 2,296 → 2 lines

```
Creating cache app/resource/cachename...
.cache/pip: found 513 matching files and directories
```

Figure 6.4 Example of LogSift producing a non-relevant segment. Despite high reduction, the identified statements contain no failure indicators or host resolution information

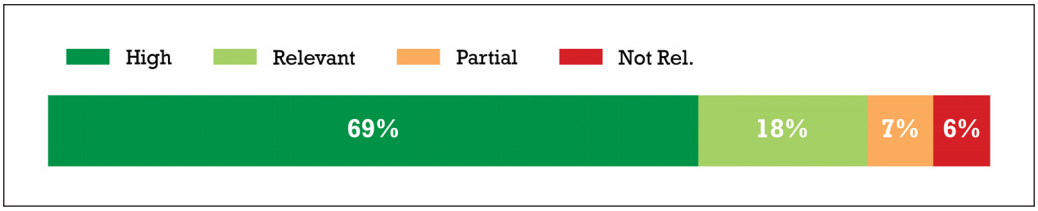


Figure 6.5 LogSift relevance assessment results

an additional 18% were rated *Relevant*. Figure 6.2 illustrates LogSift at its best: for a 359-line log of a `misconfigured env variable` failure, LogSift successfully retrieved the 2 lines pinpointing the missing variable (`IMAGE_NAME`) and its context, precisely the information needed for repair. LogSift also provides relevant information for misclassifications: Figure 6.3 shows a job incorrectly predicted as `api gateway deployment error`, when the true category is `external file invalid format`. Despite the misclassification, LogSift surfaced the `curl` failure reading a zip file, enabling developers to still identify the underlying issue and override the prediction. However, LogSift does not always succeed, yielding 6% outputs rated as *Not relevant*. Figure 6.4 shows a `host resolution failure` job where LogSift identified cache-related statements unrelated to DNS resolution. We hypothesize that such cases (6% of evaluated instances) occur when failure signals are distributed throughout the log or when the model relies on spurious correlations.

6.7 Discussion

FlaXifyer and LogSift offer a practical, low-effort solution for automated failure triage and diagnosis. With BGE at 12 shots, FlaXifyer achieves 84.3% Macro F1 and 92.0% Top-2 accuracy, meaning developers can confidently act on predictions or quickly verify between two candidates. The few-shot approach requires only 12 labeled examples per category, minimizing manual effort when extending to new failure types. If a new category becomes a priority, practitioners can collect 12 representative examples and retrain without extensive data collection. FlaXifyer predictions execute in milliseconds, enabling real-time feedback and automated triage. LogSift complements classification by reducing cognitive load. With a mean execution time under one second, it can run as a post-execution step for failed jobs, surfacing relevant log segments that accompany triage notifications or are saved as job artifacts for review. Together, these tools enable a streamlined workflow where failures are automatically categorized, directed to the appropriate team, and accompanied by actionable diagnostic information.

Semantic overlap between failure categories explains both the value of Top-2 accuracy and LogSift's limitations in certain cases. Several categories exhibit overlapping log patterns: environment variable errors during container operations, file format issues during API gateway deployment, DNS resolution failures during Git operations, and test timeouts during UI rendering. This overlap arises partly from label consolidation in our prior work (Aïdasso *et al.*, 2025a), where related failure patterns were grouped to avoid label explosion. For instance, file read failures were consolidated under `external file invalid format`, and various Helm-related errors under `helm resource error`, resulting in heterogeneous log patterns within these categories. Such overlap explains why Top-2 accuracy (92.0%) substantially exceeds Top-1 accuracy (85.3%), as the correct category often appears among the top two predictions. However, this same overlap causes LogSift to occasionally surface segments that match the predicted rather than the true category, as observed in 6% of evaluated instances. Future work could explore multi-label classification, where logs are mapped to multiple applicable categories, though Top-2 prediction already mitigates this limitation in practice. Additionally, some failures require diagnosis beyond job logs. For example, a file read failure may stem from intermittent

infrastructure issues (e.g., network filesystem timeouts) not captured in application logs. Future studies should investigate mining complementary data sources, such as infrastructure logs or telemetry data, to support end-to-end root cause analysis.

6.8 Threats to Validity

Internal Validity. Our dataset relies on automated labeling using regex patterns, achieving 91% precision (Aïdasso *et al.*, 2025a). While this introduces approximately 9% label noise, few-shot learning has demonstrated robustness to noisy labels (Tunstall *et al.*, 2022). Few-shot performance can also vary depending on which examples are sampled for training. We mitigate this using Monte Carlo Cross-Validation with 30 iterations to ensure stable estimates. Hyperparameter selection using Optuna (Akiba *et al.*, 2019) with 5 trials may not identify globally optimal configurations, though our results demonstrate strong performance across settings. For qualitative evaluation, the first author assessed LogSift outputs using structured binary criteria, with findings validated during weekly meetings with engineers at TELUS. Sample judgments are included in our replication package (Aïdasso, 2026) for verification.

External Validity. Our evaluation focuses on industrial projects at TELUS, which may limit generalizability to other organizations. However, the failure categories studied (e.g., network timeouts, dependency failures, infrastructure errors) are common across CI/CD systems (Lampel *et al.*, 2021; Durieux *et al.*, 2020), and the few-shot approach requires only 12 labeled examples per category, facilitating adaptation to new contexts. To support reproducibility, our replication package (Aïdasso, 2026) includes the complete FlaXifyer and LogSift implementation, experimental scripts for all RQs, and sanitized evaluation results. The TELUS dataset is not included due to confidentiality constraints. For replication in new environments, we include our regex-based labeling tool (Aïdasso, 2025b) along with a sample dataset of labeled intermittent job failures from the `veloren/veloren` GitLab project.

6.9 Conclusion

Intermittent job failures disrupt CI/CD pipelines, wasting computational resources and requiring significant diagnosis effort (Aïdasso *et al.*, 2025a,b; Olewicki *et al.*, 2022; Lampel *et al.*, 2021). In this paper, we introduced FlaXifyer, a few-shot learning approach for predicting intermittent job failure categories using pre-trained language models. Evaluated on industrial projects at TELUS, FlaXifyer achieves 84.3% Macro F1 and 92.0% Top-2 accuracy with only 156 labeled log examples for 13 categories (12 per category), and scales gracefully when extending to new failure categories. We also proposed LogSift, an interpretability technique that reduces log review effort by 74.4% while surfacing relevant failure information in 87% of cases. Together, these tools enable automated triage, reduce cognitive load, and accelerate failure resolution. As part of our future work, we will explore multi-label classification to better handle failures exhibiting overlapping log patterns, along with more refined category definitions to reduce semantic ambiguity. We also plan to investigate AI agents for automated failure resolution, leveraging FlaXifyer's predictions and LogSift's highlighted segments to trigger category-specific repair actions without human intervention.

CHAPTER 7

ON THE ILLUSION OF SUCCESS: AN EMPIRICAL STUDY OF JOB RERUNS AND SILENT FAILURES IN INDUSTRIAL CI

Henri Aïdasso[1] , Francis Bordeleau[1] , Ali Tizghadam[2]

[1] Department of Software and IT Engineering, École de Technologie Supérieure,
1100 Notre-Dame Ouest, Montreal, Quebec, Canada H3C 1K3
[2] Network Softwarisation and AI, TELUS,
Toronto, Ontario, Canada

Article submitted to « IEEE Transactions on Software Engineering » September 2025

7.1 Preamble

This chapter presents our empirical study of silent failures in industrial CI, submitted to IEEE TSE (Aïdasso, Bordeleau & Tizghadam, 2025c). While Chapters 4 through 6 focus on intermittent job failures (i.e., unreliable and misleading job failures), this chapter investigates the complementary pipeline reliability problem: the phenomenon of silent job failures (i.e., false positives where jobs report success despite failing to perform their intended tasks).

7.2 Introduction

Compounding the problem of intermittent job failures is the phenomenon of *silent failures*, where a pipeline job signals a success after its execution (i.e., exit code 0 and green check mark), but in reality fails, at least partially, to carry out its intended tasks. These silent failures typically remain undetected until a developer notices a problem, such as a missing job artifact or the absence in production of a newly deployed software version. One such example is reported as an issue in the open-source GitLab Runner project[1], where the cache upload step failed silently in a job appearing successful, without any trace of error or warning in the logs.

As a consequence, silent failures cause delayed feedback to developers, erode their trust in job successes, and lead to ineffective CI processes. Indeed, when faced with unclear or unexpected

[1] https://gitlab.com/gitlab-org/gitlab-runner/-/issues/2291

pipeline outcomes, developers often rerun the suspected jobs multiple times (see Fig. 7.1) just to be sure and in the hope that the job will ultimately run as expected (Olewicki *et al.*, 2022; Aïdasso *et al.*, 2025a). Such frequent reruns place additional strain on already limited CI server resources, increase developer waiting times, and quickly escalate CI costs for organizations (Aïdasso *et al.*, 2025a; Olewicki *et al.*, 2022; Aïdasso *et al.*, 2025f). Prior work (Durieux *et al.*, 2020; Olewicki *et al.*, 2022; Aïdasso *et al.*, 2025a,b) studying the practice of rerunning jobs (both in open-source and industrial contexts) has focused on intermittent job failures, where explicitly failed jobs are rerun until they pass. More recently, Maipradit *et al.* (2024) and Brus, Maipradit, Barr & McIntosh (2025) studied rechecks of *failing* CI runs in OpenStack, modeling the likelihood of a recheck resolving a failure. In contrast, the complementary phenomenon of silent failures, where jobs *appear* successful but fail to complete their intended tasks, has received little attention, most likely due to the challenge of identifying them without explicit error traces.

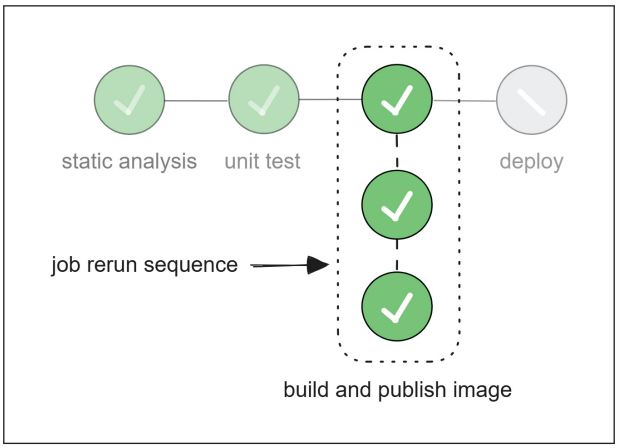


Figure 7.1 Illustration of a typical CI pipeline of four jobs, highlighting the rerun sequence of the third job responsible for building and publishing the container image. This job is re-executed twice due to silent failures, resulting in three total executions of this same job without any code or CI configuration file change

Therefore, silent failures pose a critical challenge to the reliability of CI processes, which can no longer be disregarded. Indeed, they (1) undermine automation goals by increasing manual interventions and hindering reproducibility, (2) cause delayed detection of bugs and release issues, and (3) lead to wasteful reruns of pipeline jobs. Hence, there is a strong need for our

industrial partner to understand the circumstances under which silent failures occur, in order to raise awareness among teams and support their effective identification and resolution. These findings will also benefit practitioners aiming to improve the reliability of their CI processes and open new research directions on reliability for CI researchers.

To this end, we first assess the prevalence and cost of rerunning (successful) jobs by analyzing 142,387 jobs collected across 81 projects at TELUS and spanning 6 years. Then, we investigate the most important factors associated with the rerun of successful jobs to better understand the circumstances under which such reruns occur. Finally, we examine 92 publicly available issues related to silent failures to identify the most common causes and existing solutions. Specifically, we answer the following research questions (RQs):

RQ_1. How often do developers rerun successful jobs? What is the impact of rerunning successful jobs? In the studied projects, about 11% of successful jobs (11%) have been rerun. These reruns account for more than half (51%) of all reruns without code changes, even more than reruns of failed jobs (44%). While reruns of failed jobs typically occur within 15 minutes, over 35% of the reruns of successful jobs are triggered after 24 hours, with delays even exceeding 72 hours. Moreover, reruns of successful jobs consume nearly half (48%) of the total server time spent on all job reruns. These findings highlight the prevalence of silent issues prompting reruns and their substantial impact on CI costs.

RQ_2. What are the most important circumstances under which developers rerun successful jobs? We studied the influence of 32 independent variables, grouped into three families, on the likelihood of rerunning a successful job using mixed-effects logistic regression models. The results revealed influential factors associated with a higher probability across all three families: job execution context (e.g., testing and static analysis jobs), project and its history (e.g., languages with limited IDE support, such as Shell, and a larger number of commits), and the author's recent tendency to rerun jobs. Inversely, some factors are significantly associated with a lower probability, such as executions on feature branches where developers work in isolation, projects with higher pipeline failure ratios, and authorship from more experienced developers.

RQ_3. What types of failures occur silently within jobs marked as successful? We manually examined 92 public issues where practitioners reported cases of silent failures in jobs appearing successful. We identified 11 themes, with the top three most common being artifact operation errors (28%), caching errors (23%), and ignored non-zero exit codes (18%). These themes align with the identified influential factors. In fact, testing and static analysis jobs often handle artifacts (e.g., test reports) and caches (e.g., dependencies), while ignored exit codes are common in scripting-heavy jobs (e.g., Shell). Finally, we summarized the practical solutions proposed for these issues to facilitate their reuse by practitioners.

A replication package (Aïdasso, Bordeleau & Tizghadam, 2025d) have been made publicly available and include: **(1) Datasets** of 142,387 jobs from 81 industrial projects, 198,146 jobs from 8 popular open-source projects, and 92 identified public issues related to silent failures; **(2) Source Code**, notebooks, along with intermediate results to enable verification and replication of our study; and an **(3) Appendix IV** providing additional study details and results of our replication study on the 8 open-source projects, supporting the external validity of our findings.

7.3 Background and Motivation

Silent Failures. During interviews conducted at TELUS, engineers have reported cases where build jobs are marked as successful despite failing to complete (or only partially completing) their specified tasks. We refer to this phenomenon as silent failures, which often go unnoticed for extended periods and erode developers' trust in the success of the CI/CD process. One such example is presented in this public issue[2] on GitLab, describing the case of a silent (license scanning) job failure, where ". . . the job succeeds, but actually fails silently without reporting anything [...] without any warning or error. . . ".

Another public issue[3] reports successful jobs that fail to upload declared artifacts. The issue author underlined that ". . . as a result [of silent failures in successful jobs], one can introduce a

[2] https://gitlab.com/gitlab-org/gitlab/-/issues/224187

[3] https://gitlab.com/gitlab-org/gitlab/-/issues/22711

regression in its build process without discovering it before days or client reporting incapacity to download artifact. . . ".

Other examples noted at TELUS include container build jobs that signal success while no new image version is actually available in the registry, testing jobs that silently skip buggy files due to caching strategies, and many other such problems. These silent failures lead to significant waste in the software development lifecycle in terms of delayed detection of issues or code defects, long manual verifications, delayed releases, and wasteful job reruns.

Limitations of Logs. Developers typically leverage job execution logs to investigate and diagnose the root causes of failures. Resulting from numerous command and tool executions, these logs are highly heterogeneous and often extremely large, reaching thousands of lines and sizes of up to 50 MB (Brandt *et al.*, 2020). Such volumes significantly make diagnosing job failures difficult. But worst, silent failures typically show no clear signs in the logs.[3]

```
1  Running with gitlab-runner...
2  ...
3  Creating cache [CACHE_KEY]...
4  Created cache
5  Job succeeded
```

Figure 7.2 Log excerpt of a silent cache upload failure in a successful job

In many cases, it is very difficult (if not impossible) to identify silent failures in CI jobs only based on logs. Fig. 7.2 shows an excerpt of execution logs of a successful job containing a cache upload failure as reported in this issue.[4] As can be observed, the logs contain no errors or warnings about the failure, although developers, as noted in the issue description, would have preferred to be notified of the problem. Another example is shown in Fig. 7.3, from an issue[5] on the GitLab project, where a `notify pipeline failure` job appears successful but silently fails to send pipeline incident report files to the teams. As shown in the figure, the logs appear to indicate complete success of the job. On closer inspection, however, one can observe the small

[4] https://gitlab.com/gitlab-org/gitlab-runner/-/issues/2291

[5] https://gitlab.com/gitlab-org/gitlab/-/issues/384171

chunk `invalid_blocks`, which reveals failure of the previous command. In large logs (e.g., exceeding 500 lines[6]), such a short and unclear error trace is typically overlooked, particularly by less-experienced developers.

```
1  Running with gitlab-runner 15.6.0...
2  ...
3  $ scripts/generate-failed-pipeline-slack-message.rb -i
       ${BROKEN_MASTER_INCIDENT_JSON} -f
       ${FAILED_PIPELINE_SLACK_MESSAGE_FILE};
4    % Total % Received % Xferd Average Speed Time Time Time Current
5                                 Dload Upload Total Spent Left Speed
6  100 6483 0 14 100 6469 112 51752 --:--:-- --:--:-- --:--:-- 51864
7  invalid_blocks
8  Uploading artifacts for successful job
9  Uploading artifacts...
10 /.../incident.json: found 1 matching files and directories
11 ...
12 Uploading artifacts as "archive" to coordinator... 201 Created...
13 Cleaning up project directory and file based variables
14 Job succeeded
```

Figure 7.3 Log excerpt of a silent HTTP request failure in a successful job

Rerun Practice and Terminology. When suspicious about a build job outcome, developers tend to rerun the job multiple times without any change to the source code or CI configuration file (Olewicki *et al.*, 2022; Aïdasso *et al.*, 2025a), hoping for either (1) an actual job failure indicating the issues or (2) a true success with job tasks being actually completed (Hilton *et al.*, 2016). Fig. 7.1 illustrates a job rerun scenario, where the first two successful executions of the `build and publish image` job are rerun, respectively leading to an additional execution of that job. For clarity, we define below the terminology used about job reruns throughout this paper.

Let $J_0^i, J_1^i, J_2^i, \ldots, J_n^i$ denote the $n+1$ consecutive executions of the same job J on an identical code version, identified by the i^{th} commit c_i in a given project. Each subsequent job execution J_{k+1}^i ($k \geq 0$) results from a manual rerun of the preceding execution J_k^i. For simplicity, we refer to each execution instance simply as a *job*. A **job rerun sequence** is the set $\{J_0^i, J_1^i, J_2^i, \ldots, J_n^i\}$ consisting of all (re)executions of a job J on a given commit c_i. In each rerun sequence, we

[6] https://gitlab.com/gitlab-org/gitlab/-/jobs/3398283094

refer to the first job J_0^i and the last job J_n^i respectively as the **original** and **final jobs**. For each pair ($J_k^i \rightarrow J_{k+1}^i$) in the rerun sequence, we refer to J_k^i as the **rerun-initiating job**, while J_{k+1}^i denotes the **rerun job** resulting from manually rerunning J_k^i. Note that the original job is only a rerun-initiating job (and not a rerun job). Similarly, the final job is only a rerun job and not a rerun-initiating one. As a result, there is an equal number n of rerun-initiating and rerun jobs in a job rerun sequence of $n + 1$ jobs. In an ideal wasteless CI/CD process, only the original job of a rerun sequence is executed. So, the additional rerun jobs in the sequence (i.e., all except the original job) are considered wasteful jobs. Finally, in the example of Fig. 7.1, silent job failures correspond to the first and second executions of the `build and publish image` job.

Although there is no official policy on rerunning suspicious jobs at TELUS, developers have confirmed the application of this practice in the event of silent failures. Therefore, in this paper, we study silent job failures by investigating jobs that were manually rerun (i.e., rerun-initiating jobs) despite having previously completed with a success signal. Repeatedly rerunning jobs increases the load on already constrained CI servers and leads to significant developer wait times, ultimately delaying software delivery (Olewicki *et al.*, 2022; Aïdasso *et al.*, 2025a). As a result, silent job failures lead to significant waste for organizations, in particular TELUS, in terms of infrastructure and developer time.

Therefore, there is a strong need to better understand these silent job failures in order to support engineers in cutting their associated costs. Our findings will also provide valuable insights and guidelines that practitioners and CI providers can leverage to improve the reliability of build processes.

7.4 Study Design

7.4.1 Research Questions

Our study aims to investigate the prevalence and impact of rerunning successful jobs due to silent failures (RQ_1), identify the factors associated with rerunning successful jobs and thereby with

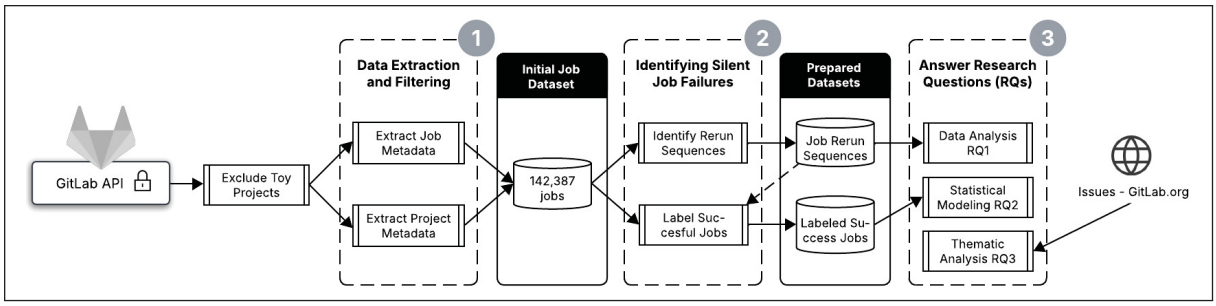


Figure 7.4 Overview of our approach for (1) data collection and (2) preparation, including data pre-processing and labeling, to (3) answer our research questions

silent failures (RQ_2), and finally uncover the categories of silent failures that occur in successful jobs (RQ_3) to better understand this phenomenon. For this purpose, we seek to answer the following research questions (RQs):

RQ_1. How often do developers rerun successful jobs? What is the impact of rerunning successful jobs?

RQ_2. What are the most important circumstances under which developers rerun successful jobs?

RQ_3. What types of failures occur silently within jobs marked as successful?

7.4.2 Data Collection

Project Selection. We identify with our industrial partner 14 main groups of actively maintained projects. From these groups, we filter out projects with fewer than 50 commits to exclude toy and experimental projects. As a result, we identified 81 microservice projects, from which we extracted the entire history of pipeline jobs up to July 11, 2024. Data collected from the oldest project span six years, while data from the youngest project span four months. These projects vary in size, purpose (e.g., backend APIs, network controllers, web portals, data analytics), and cover 13 programming languages, including Python, Go, C, TypeScript, JavaScript, and Shell. Appendix II lists the 81 selected projects and their characteristics, including the number of commits, age, main programming language, purpose, and job execution statistics. Project names are omitted for privacy reasons.

Data Extraction. We collect raw metadata from pipeline jobs in the 81 identified projects, using the GitLab REST APIs[7]. Specifically, we retrieve the raw metadata JSON files from a total of 153,374 jobs available in these projects. The JSON files are then flattened and consolidated into a CSV file. We only keep the data fields needed to compute the features listed in Table 7.2, which are discussed in detail in Section 7.4.4. Additionally, we collect project-level metadata, including the main programming language, project creation date, and the total number of commits.

Table 7.1 Overview of the collected data for studying silent failures

Metric	Value
# Projects	81
# Languages	13
# Commits	45,401
# Builds	53,851
# **Jobs**	**153,374**
# **Executed Jobs**	**142,387 (92.83%)**
# Success Jobs	106,678 (69.55%)
# Failed Jobs	29,983 (19.55%)
# Canceled Jobs	5,726 (3.73%)
# **Non-executed Jobs**	**10,987 (7.16%)**
# Skipped Jobs	8,443 (5.50%)
# Manual Jobs	2,450 (1.60%)
# Created Jobs	94 (0.06%)
Date range of data collection	Feb 2018 to July 2024

Table 7.1 presents a summary of the collected data. We observe a ratio of 2.85 (i.e., 2 to 3) executed jobs per pipeline across the studied projects. Out of the 153,374 collected jobs, about 70% completed successfully, underscoring the critical importance of ensuring the reliability of *success* outcomes. The *failed*, *canceled*, *skipped*, *manual*, and *created* jobs represent respectively 19.55%, 3.73%, 5.50%, 1.60%, and 0.06%.

Data Filtering. We filter out jobs with statuses *skipped*, *manual*, and *created*, since these statuses indicate that the jobs were never executed[8] and thus provide no valuable information for the present study focusing on pipeline outcomes' reliability. Besides, these jobs only represent

[7] https://docs.gitlab.com/ee/api/jobs.html

[8] https://docs.gitlab.com/ci/jobs/#available-job-statuses

about 7% of the collected data. We sort the remaining 142,387 executed jobs by date of creation (from the oldest to the most recent), forming the initial dataset of studied jobs.

7.4.3 Identifying Silent Job Failures

In this study, we use the rerun of successful jobs as a *proxy* for silent job failures. Rerun-initiating jobs having a *success* status are therefore treated as candidate silent failures. As discussed in Section 7.3, systematically identifying silent failures from the logs of successful jobs is impractical, as they often leave little to no noticeable trace of error (see Fig. 7.2 and Fig. 7.3). Thus, we study silent job failures through the lens of successful jobs that developers have rerun. Indeed, based on interviews with TELUS engineers who confirmed the practice of rerunning suspected successful jobs, we hypothesize that successful jobs are rerun because they fail to complete their intended tasks, whether entirely or partially. This approach follows the well-established practice of studying reruns of *failed* jobs to characterize intermittent job failures (Durieux *et al.*, 2020; Olewicki *et al.*, 2022; Aïdasso *et al.*, 2025a), which we extend to successful jobs.

Identifying Rerun Sequences. We start by identifying jobs that were re-executed without any change to the source code or CI configuration file. To this end, we group the jobs by *name*, *commit* ID, and *project*, in line with existing works (Olewicki *et al.*, 2022; Aïdasso *et al.*, 2025b,a). In GitLab CI, job names are unique within a project's pipeline configuration. This grouping, therefore, uniquely identifies a specific job definition executed multiple times on the same code version, with no change to any version-controlled file, including the CI configuration file (`.gitlab-ci.yml`). From these groups, we identify job rerun sequences (see Fig. 7.1) as the job groups including more than one job, i.e., groups containing at least one additional execution of the same job on the same code version. As a result, we obtained a total of 13,380 job rerun sequences, each containing jobs in chronological order of execution. We use the dataset of job rerun sequences to answer our RQs. Specific approaches are detailed in Section 7.5.

Labeling Successful Jobs. To address RQ_2, which investigates the factors distinguishing rerun successful jobs (silent failures) from non-rerun successful jobs (true successes), we focus on the dataset of 106,678 successful jobs. We label this dataset using the 13,380 job rerun sequences by identifying the IDs of rerun-initiating jobs having a `success` status. This filtering on the `success` status ensures that jobs that were accidentally canceled and subsequently rerun are excluded from the RQ2 analysis by design. The successful jobs matching these IDs in the dataset are labeled as having been rerun (`rerun = 1`), while all others are labeled as non-rerun (`rerun = 0`). As such, the variable `rerun` indicates whether a successful job was rerun or not and is the dependent variable that we seek to explain in RQ_2. The results of this labeling process, which provide insights into the prevalence of silent failures leading to re-executed successful jobs, are discussed in RQ_1.

7.4.4 Independent Explanatory Variables

Similar to prior studies (Weeraddana *et al.*, 2024a; Benjamin *et al.*, 2023; Hong *et al.*, 2024), we identify a set of independent variables aimed at uncovering the most influential factors associated with the rerun of successful jobs, i.e., of silent job failures (RQ_2). Table 7.2 presents the initial list of 14 variables with their descriptions and the rationale for studying them. These variables are grouped into three families. First, the *job execution context* family includes job-level factors potentially associated with silent issues that prompt developers to rerun successful jobs. Inspired by previous studies (Benjamin *et al.*, 2023; Ghaleb *et al.*, 2019; Kola-Olawuyi *et al.*, 2024), the other two variable families, *project factors* and *developer factors*, enable us to assess how project characteristics and developer profiles relate to the rerun of successful jobs. The insights from these associations can help managers identify and focus efforts on the most critical types of projects and help raise awareness among teams.

Table 7.2 Factors used as independent variables. “F." stands for Family

F.	Description/Rationale
Job Execution Context	**job_type**: Type of executed job derived from the job name (e.g., static analysis, testing, deployment). Rationale: Inspired by Rausch *et al.* (2017), reruns of successful jobs might be associated with some specific types of jobs (e.g., deployment jobs).
	branch_type: The type of branch on which the job was triggered (e.g., main, develop, feature, hotfix). Rationale: Reruns of successful jobs might be associated with jobs triggered on specific branches, since the stability expectations are different across branches (Islam & Zibran, 2017).
	month_of_year, **day_of_week**, **hour_of_day**: The moment when the job is triggered. Values range in [1–12], [1-7], and [0-23]. Rationale: Inspired by this issue (noa, 2019) and prior work (Weeraddana *et al.*, 2024a; Kola-Olawuyi *et al.*, 2024; Ghaleb *et al.*, 2019), different load periods on the CI server can explain success reruns.
Project (History) Factors	**language**: The main programming language of the project. Rationale: Successful job reruns might be more or less common in projects written in specific programming languages (Ghaleb *et al.*, 2019; Benjamin *et al.*, 2023).
	age: Age of the project in days when the job is triggered. Rationale: Inspired by Benjamin *et al.* (2023); Kola-Olawuyi *et al.* (2024), older or younger projects may be more or less prone to instabilities and reruns of successful jobs.
	commits: Number of commits at the time the job is triggered. Rationale: Mature or complex projects may more easily include silent regressions explaining successful job reruns (Benjamin *et al.*, 2023).
	build_frequency: Average #jobs per day when the job is triggered. Rationale: A high job frequency may stress CI resources, increasing the likelihood of silent issues and of rerunning successful jobs.
	failure_ratio: Percentage of failed jobs when the job is triggered. Rationale: Projects with lower failure rates might still suffer from silent issues and thus prompt to rerunning successful jobs (Kola-Olawuyi *et al.*, 2024).
	rerun_ratio: Percentage of rerun jobs, when the job is triggered. Rationale: Inspired by Olewicki *et al.* (2022), a high proportion of prior rerun jobs may suggest reliability issues within the project, increasing the likelihood of rerunning successful jobs.
Developer Factors	**author_exp_days:** Experience (number of days since first contribution) of the developer who authored the triggering commit. Rationale: Less experienced developers may have a less thorough understanding of projects (Ghaleb *et al.*, 2019; Rausch *et al.*, 2017) and are more likely to rerun successful jobs when unsure of the outcome.
	author_exp_commits: Experience (ratio of total commits in the project) of the developer who authored the triggering commit. Rationale: Inspired by Ghaleb *et al.* (2019); Rausch *et al.* (2017); Kola-Olawuyi *et al.* (2024), the ratio of commits is another estimation of the author’s experience, which, similarly to author_exp_days, might explain rerun of successful jobs.
	author_recent_rerun_history: Proportion of rerun jobs among the last five jobs triggered by the author of the current job. Rationale: Inspired by Weeraddana *et al.* (2024a); Kola-Olawuyi *et al.* (2024), the author’s tendency to rerun jobs suggests they might be working on change sets associated with reliability issues, increasing the odds of rerunning successful jobs.

The independent variables used in our model draw on feature families commonly used across CI factor modeling studies, where they have been applied to different dependent variables such as build duration (Ghaleb *et al.*, 2019), build timeouts (Weeraddana *et al.*, 2024a), build failures (Rausch *et al.*, 2017; Islam & Zibran, 2017), and recheck outcomes (Brus *et al.*, 2025). In our study, the dependent variable is whether a *successful* job is rerun, a proxy for silent failure. Reusing established feature families to study a new outcome enables us to build on factors whose practical relevance has been demonstrated in prior CI research.

In Section 1 of Appendix IV, we describe the feature engineering of the categorical independent variables (i.e., `job_type`, `branch_type`, and `language`) and show the distribution of categories for each of these variables. In particular, we assigned branch types based on the GitFlow naming convention (Atlassian, 2024) adopted in the studied projects, which includes five primary types: `main`, `develop`, `feature`, `release`, and `hotfix`. In particular, we identified 7 job types (`testing`, `deployment_and_release`, `static_code_analysis`, `image_security_scan`, `code_and image_build`, `notification`, and `other`) via open-coding (Corbin & Strauss) and label consolidating, 5 branch types (`develop`, `feature`, `release`, `hotfix`, and `main`) based on the Gitflow[9] naming convention, and 10 main languages (Python, Gherkin, Go, HCL[10], HTML, JS, Jinja, Shell, TSX, and TS). The remaining numerical independent variables are directly computed from the collected data fields as described in Table 7.2.

7.4.5 One-Hot Encoding of Categorical Variables

We use one-hot encoding[11] to transform the categorical variables (`job_type`, `branch_type`, and `language`) into binary dummy variables (Kumar, 2021). For each variable, one-hot encoding creates multiple dummies, one for each of its category levels. A dummy variable having a value of 1 indicates that the observation (i.e., the job) belongs to the corresponding category level, while 0 indicates it does not. As a result, we derived 7 dummy variables for job

[9] https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

[10] HCL is a domain-specific language for defining Infrastructure-as-Code

[11] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html

types (e.g., `job_type_testing`), 5 for branch types (e.g., `branch_type_feature`), and 10 for the language variable (e.g., `lang_python`). In statistical modeling, one dummy is typically removed for each categorical variable to prevent multicollinearity and serve as a reference category for interpreting the remaining dummies' log-odd coefficients (Kumar, 2021; Ghaleb *et al.*, 2019). The choice of the reference to drop is described further in Section 7.5.2. At this stage, we obtained 33 numerical (independent) variables and one dependent binary variable for the 106,678 successful jobs in our dataset for RQ_2.

7.4.6 Cyclical Encoding of Periodic Variables

A subset of our variables captures time periods (i.e., `hour_of_day`, `day_of_week`, and `month_of_year` with 23, 7, and 12 category levels, respectively). However, treating these variables as categorical as Ghaleb *et al.* (2019) would (1) involve creating 42 additional dummy variables (23 + 7 + 12), increasing the complexity of the model, and (2) imply creating artificial discontinuities at period boundaries (e.g., between hour 23 and 0). In fact, one-hot encoding ignores the natural cyclical patterns of temporal variables, which could nonetheless offer more meaningful insights across broader periods (e.g., end of the year) rather than fixed categories like individual months. To capture the cyclical patterns in these variables, we apply cyclical encoding (Bansal, Balaji & Lalani, 2025), which transforms each variable t into two sinusoidal components as follows:

$$\begin{cases} x_{\sin}(t) = \sin\left(2\pi \cdot \frac{t}{P}\right) \\ x_{\cos}(t) = \cos\left(2\pi \cdot \frac{t}{P}\right) \end{cases} \quad \text{where } P = \max(t) \tag{7.1}$$

As a result, each temporal variable is encoded into sine and cosine variables representing their (x, y) coordinates on a circle, where period boundaries (e.g., hour 23 and 0, or December and January) are closer. Therefore, we derive six $(2 * 3)$ new variables in place of the three temporal variables, resulting in a total of 36 independent variables.

Summary of Datasets. We use the 13,380 job rerun sequences (36,489 jobs) for RQ1, and the 106,678 successful jobs across 81 projects for RQ2, each with 36 numerical independent variables and one dependent variable for statistical modeling.

7.5 Results and Discussion

7.5.1 RQ_1. How often do developers rerun successful jobs? What is the impact of rerunning successful jobs

7.5.1.1 Motivation

The goal of this first RQ is to evaluate the prevalence and cost of the practice of rerunning jobs at TELUS, especially jobs that had already been completed successfully. Indeed, we leverage reruns of successful pipeline jobs as an indicator of silent failures, based on developers' experience with silent failures as reported during interviews. However, since there is no explicit policy at TELUS recommending reruns in response to unreliable outcomes or silent failures, we first seek to examine the prevalence and impact of such practice. The findings from this RQ will offer valuable insights into the practice of rerunning successful jobs, laying the groundwork and motivation for a deeper investigation of silent failures and associated job reruns.

7.5.1.2 Approach

To answer RQ_1, we identify jobs that were rerun without any modifications to the source code or CI configuration file. Following the method described in Section 7.4.3, we identify the job rerun sequences (illustrated in Fig. 7.1), resulting in a dataset of 13,380 sequences. In a rerun sequence containing n jobs, $n - 1$ are rerun jobs, i.e., all but the original job. Similarly, $n - 1$ are rerun-initiating jobs, i.e., all but the final job. Using the dataset of job rerun sequences, we compute statistics on rerun jobs (overall and by rerun-initiating job status) and analyze the distribution of rerun counts (i.e., rerun sequence lengths minus one) across projects. In addition, similarly to prior work (Durieux *et al.*, 2020), we investigate the distribution of status change in

Table 7.3 Statistics of job reruns at TELUS

Metric	Value
# Projects including rerun jobs	81 (100%)
# Rerun jobs	23,109 (16.23%)
# Rerun sequences	13,380
# Sequence of 1 rerun	9,317 (69.63%)
# Sequence of 2 reruns	2,263 (16.91%)
# Sequence of 3 reruns	865 (6.46%)
# Sequence of 4 reruns	379 (2.83%)
# Sequence of 5 reruns	186 (1.39%)
# Sequence of ≥ 6 reruns	370 (2.76%)
Avg # of jobs per sequence	≈ 2
Max # of reruns for a single job	29

pairs of rerun-initiating jobs and subsequent rerun jobs to better assess how often developers rerun successful jobs versus failed or canceled ones. Furthermore, we analyze the distribution of time gaps between the start of each rerun-initiating job and its subsequent rerun within the sequences, by rerun-initiating job status, in order to assess the delays before noticing (silent vs. actual) failures and diagnosing them. Finally, we evaluate the amount of machine time spent running wasteful rerun jobs that would be saved if only the original jobs had been executed in each rerun sequence.

7.5.1.3 Results

Table 7.3 presents job rerun statistics for the 81 studied projects, including the distribution of rerun counts and the number of reruns by rerun-initiating job status. Fig. 7.5 shows the distribution of status transitions between each job and its immediate rerun in the sequences.

A significant proportion (approximately 16%) of the studied jobs have been rerun, highlighting the prevalence of this practice. On average, these reruns occur in sequences involving two additional runs of the same job, with the number of reruns reaching up to 29. As reported in Table 7.3, we found 23,109 rerun jobs, accounting for 16.23% of all 142,387

studied jobs (including passed, failed, and canceled ones). The rerun ratio (16.23%) is strikingly high in the industrial setting, almost ten times that (1.7%) reported in a prior open-source study (Durieux *et al.*, 2020) and more than double the ratio (7.95%) observed in our replication study on open-source projects detailed in Section 4 of Appendix IV. We note that direct comparison of rerun ratios across studies requires caution, as they operate at different granularities and in different ecosystems. Durieux *et al.* (2020) report 1.72% of restarted *builds* on Travis CI (pipeline level), while our ratio of 16.23% measures reruns at the *job* level within a pipeline, which is a finer granularity. A single pipeline typically contains multiple jobs, making these ratios not directly comparable. Besides, rerun jobs were found in all 81 studied projects, further emphasizing the widespread nature of this practice across the industrial projects. The 23,109 rerun jobs are found in a total of 13,380 rerun sequences, averaging 1.73 reruns per sequence (i.e., typically one to two reruns). Specifically, the majority of rerun sequences (69.63% ≈ 70%) consist of a single rerun. The remaining 17%, 6%, 3%, 1%, and 3% of job rerun sequences involve 2, 3, 4, 5, and more than 5 (up to 29) re-executions of the same job, respectively.

Figure 7.5 Evolution of job statuses between rerun-initiating jobs (x-axis) and the immediate rerun jobs (y-axis) in the identified job rerun sequences

More than half (51%) of the rerun-initiating jobs were successful, while failed ones account for 44%. This result indicates a significant presence of silent job failures (≈ 11%) among all jobs marked as successful, and emphasizes the need for further investigation to understand their associated factors and impact. As previously described, 23,109 rerun jobs imply 23,109 rerun-initiating jobs identified in the rerun sequences. Of these 23,109 rerun-initiating jobs, 11,808 (i.e., 51.08% or more than half of the rerun-initiating jobs) are jobs that have initially completed their execution with a *success* signal. These jobs represent 11% of all the studied successful jobs. In contrast, failed jobs represent 44% of the rerun-initiating jobs, while canceled jobs account for the remaining 5%. The notably high proportion of successful jobs in the rerun-initiating ones, surpassing that of failed jobs, is particularly intriguing. Intuitively, one would expect failed jobs to be the primary trigger for reruns, as they more clearly warrant developer intervention. Nevertheless, this finding aligns with developers' reports during interviews, where they described frequently encountering silent failures hidden under signals of job success. It also emphasizes the need to investigate the factors associated with the reruns of successful jobs, in order to better understand this underexplored phenomenon.

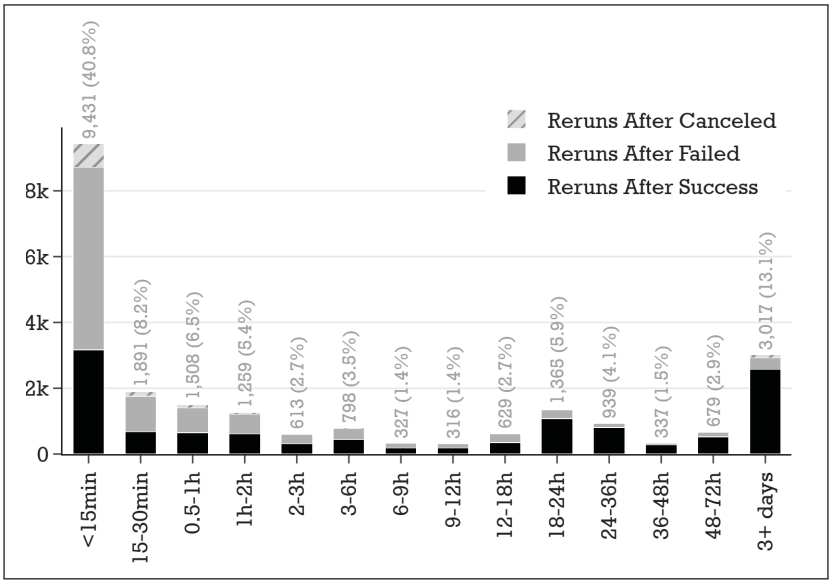


Figure 7.6 Distribution of time delays between rerun-initiating jobs and job reruns, by status of the initiating jobs. Each bar is annotated with the number of rerun jobs in that interval, with the corresponding proportion in parentheses

Reruns after successful jobs happen much later (median 6.9 hours, 4.6 days on average) compared to reruns after failed jobs (median 11.4 minutes, 13.3 hours on average) and canceled jobs (median 8.4 minutes, 1.3 days on average). This shows that silent failures often go unnoticed for long periods and suggests they require longer diagnostic times. Fig. 7.6 shows the distribution of delays between the creation times of each rerun-initiating job and its immediate rerun. It highlights in black the distribution of time delays for instances where the rerun-initiating jobs were successful, i.e., delays before rerunning successful jobs. As shown in the figure, a substantial portion (40.8%) of reruns takes place within 15 minutes, regardless of the rerun-initiating job status. Notably, over half of all reruns (55.5%) occur within one hour, including the vast majority (73.3%) of reruns initiated following failed jobs. These results indicate that developers respond quickly to explicit failures. In contrast, reruns following successful jobs follow two different patterns, as can be seen in Fig. 7.6. Indeed, 31.4% of these reruns are initiated within 15 minutes, while a further 30.6% are initiated only after 2 days. Moreover, the majority (52.03%) of reruns following successful jobs take place only after a one-day delay. Discussions with developers at TELUS suggest that this is because, on the one hand, some silent job failures (e.g., deployed container not running in production) are noticed quite quickly when developers already suspect the job and are actively awaiting its effects via manual verifications in production, for instance. In these cases, reruns are performed immediately in the hope that the job will actually complete its task. Alternatively, the silent failures are either only noticed very late on or require the intervention of infrastructure teams to diagnose them, thus extending the time delay for re-executing the job to ensure that it finally works as intended.

Rerun jobs following success account for nearly half (48%) of the total wasted server time from all rerun jobs. This finding emphasizes that silent failures significantly contribute to increased CI costs. The median duration of rerun jobs following a successful execution (1.75 minutes) is higher than that of rerun jobs after a failure (1.5 minutes). As a result, rerun jobs after success spent a substantial amount of server time, i.e., 29 days and 13 hours, which accounts for 47.5% of the total server time of 62 days and 6 hours for all reruns. To put this

in perspective, the total CI compute time across all 142,387 studied jobs is approximately 359 days. The 62 days spent on all reruns therefore represent 17.3% of the total CI compute budget, and the 29.5 days spent on reruns of successful jobs alone account for 8.2%. Thus, much of the wasteful server usage due to reruns, avoidable if the original executions had sufficed, stems from silent failures that prompt the rerun of successful jobs.

Summary of RQ_1. About 11% of successful jobs are rerun, accounting for more than half (51%) of all rerun-initiating jobs. The reruns occurred after longer time delays than those of failed and canceled jobs, and also consumed 48% of the total server time spent on all reruns. These results highlight the prevalence and stealthy nature of silent failures (causing late detection) and their high contribution to CI costs.

Replication of RQ_1 on Open-Source Projects. To assess the generalizability of these findings, we conducted a replication study on 198,146 jobs across 8 popular open-source GitLab projects (details in Section 4 of Appendix IV). We found that although the overall rerun ratio is lower in OSS projects (7.95% vs. 16.23%), reruns of successful jobs are even more predominant, accounting for approximately 76% of all job reruns (vs. 51% in the TELUS industrial setting). Furthermore, similar to the industrial context, reruns of successful jobs account for almost all reruns occurring after 6 hours. These results provide initial evidence that the practice of rerunning successful jobs and its association with silent failures extend beyond the studied industrial context.

7.5.2 RQ_2. What are the most important circumstances under which developers rerun successful jobs

7.5.2.1 Motivation

We identified in RQ_1 a significant number of initially successful jobs that have been later rerun. Hence, the goal of this RQ is to investigate whether such reruns can be explained by project and job characteristics. If so, we further seek to identify the key factors that most strongly influence the rerun of successful jobs. These findings will provide practitioners with valuable insights into

the circumstances surrounding silent failures and help raise awareness among teams to support their effective identification and mitigation.

7.5.2.2 Approach

To identify the most important factors associated with the rerun of successful jobs, we model the likelihood of developers rerunning successful jobs based on various independent variables listed in Table 7.2. For this purpose, we use mixed-effects logistic regression models[12], a type of generalized linear mixed-effects model (GLMM), that emphasizes interpretability and captures both fixed and random effects across groups (i.e., projects in our case) (Faraway, 2016).

$$\texttt{logit}(P(Y_g = 1)) = \beta_0 + \sum_{i=1}^{n} \beta_i X_i + \theta_g \tag{7.2}$$

Eq. 7.2 shows the equation of a mixed-effects logistic regression model. It estimates the log-odds of the binary outcome Y_g as a linear combination of two types of effects: fixed effects $\boldsymbol{\beta_i}$ and random effects θ_g. The fixed effects ($\boldsymbol{\beta_i}$), constant across all groups, are the coefficients of the independent variables X_i, while the random effects (θ_g) capture group-specific variations (Faraway, 2016; Ghaleb *et al.*, 2019), such as differences across projects. β_0 denotes the constant intercept representing the baseline log-odds of the outcome when all predictors are zero.

In line with prior work (Ghaleb *et al.*, 2019, 2022; Weeraddana *et al.*, 2024a), the choice of a statistical inference model (over traditional ML models is driven by our primary objective of explanation rather than prediction. Indeed, statistical models provide interpretable coefficients, allowing us to quantify the influence of each variable on the predicted outcome, unlike ML models, which typically operate as black boxes.[13]

[12] https://stats.oarc.ucla.edu/r/dae/mixed-effects-logistic-regression/

[13] https://www.fharrell.com/post/stat-ml

The initial input dataset of our approach consists of the 106,678 successful jobs, including 36 independent variables and one dependent variable (i.e., `rerun`), as detailed in Sections 7.4.3 and 7.4.4. Our dataset is imbalanced with only 11% of positive examples, as discussed in RQ_1. While class balancing techniques such as SMOTE (Chawla, Bowyer, Hall & Kegelmeyer, 2002) are commonly used in ML to improve the recall of the minority class, according to the recent findings of Weeraddana *et al.* (2024a), this technique does not lead to substantial differences in the outcomes of statistical models. Therefore, we do not apply class balancing to our dataset, which also enables our model to more accurately reflect real-world scenarios in line with Weeraddana *et al.* (2024a). While additional sensitivity analyses, such as downsampling the majority class, could be explored, our model's EPV ratio of 369 far exceeds the recommended threshold of 10 (Peduzzi, Concato, Kemper, Holford & Feinstein, 1996), and is well-calibrated as discussed in the results. These indicators provide strong evidence against overfitting and suggest that class imbalance is unlikely to affect our results. To create the mixed-effects model, we follow the guidelines for statistical regression modeling provided by Harrell (2015), which we describe in the five steps below.

Step 1. Correlation Analysis. Highly correlated variables are known to distort statistical models (Weeraddana *et al.*, 2024a; Ghaleb *et al.*, 2022), undermining the interpretation of contributions of each independent variable to the dependent variable. Therefore, we sought to address high correlations among variables using Spearman's rank correlation (ρ), which (in contrast to Pearson's) captures both linear and non-linear relationships (Weeraddana *et al.*, 2024a). For this purpose, we use the `varclus` function from the `Hmisc`[14] package in R. In line with prior work (Weeraddana *et al.*, 2024a), we use a threshold of $|\rho| = 0.7$ to determine pairs of highly correlated variables. As a result, we identify one pair of highly correlated variables (`age`, `commits`), from which we remove one variable to mitigate the collinearity (Weeraddana *et al.*, 2024a; Ghaleb *et al.*, 2019; Kola-Olawuyi *et al.*, 2024; Kumar, 2021). Specifically, we remove the variable `age`, as `commits` offers a more meaningful estimate of project maturity and complexity. Results of our correlation analysis are provided in Appendix IV.

[14] https://cran.r-project.org/web/packages/Hmisc

Step 2. Redundancy Analysis. We perform a multicollinearity analysis on the 35 remaining independent variables to identify and mitigate redundant features. Multicollinearity occurs when a feature can be linearly explained by the combination of other features (e.g., a dummy variable that can be derived from the others (Kumar, 2021)). For this purpose, we use the `redun` function from the `Hmisc`[14] package in R, which fits 35 separate models, each predicting one feature based on the remaining 34. Similar to prior work (Weeraddana *et al.*, 2024a; Ghaleb *et al.*, 2019), we use an R^2 threshold of 0.9 to identify features recommended for exclusion due to redundancy. As a result, we identify and remove the redundant variables: `branch_type_main`, `lang_python`, and `job_type_deployment_and_release`. In fact, these variables are dummies for the original variables `branch_type`, `lang`, and `job_type`, respectively, and therefore serve as reference categories for these variables. At the end of the redundancy analysis, our dataset includes 32 independent variables, which we use in our model.

Step 3. Mixed Effects Model Fitting. We fit our mixed-effects logistic regression model using the `glmer` function from the `lme4`[15] package in R. It models the binary outcome `rerun` using the 32 independent numerical variables as fixed effects and the job's project ID as the grouping factor for random effects. Following Ghaleb *et al.* (2019), we specify the *binomial* distribution, *Laplace* approximation, and *bobyqa* optimizer as parameters of the `glmer` function. Furthermore, we calculate the Events Per Variable (EPV) ratio to assess the risk of overfitting in our model (Peduzzi *et al.*, 1996). An EPV above 10 indicates a low risk of overfitting in logistic models (Peduzzi *et al.*, 1996; Ghaleb *et al.*, 2019).

Step 4. Performance Evaluation. We evaluate our model using five standard metrics: ROC AUC (Hanley & McNeil, 1982), AUC-PR (Saito & Rehmsmeier, 2015), Brier Score (Brier, 1950), Marginal R^2 (McFadden, 1978), and Conditional R^2. Definitions and interpretation guidelines for each metric are provided in Appendix IV.

Step 5. Significant Factors Identification. We estimate the importance of each independent variable using the maximum likelihood χ^2 (Chi-Squared) test, also known as the likelihood ratio

[15] https://cran.r-project.org/web/packages/lme4

χ^2 test, used in prior studies (Weeraddana *et al.*, 2024a; Ghaleb *et al.*, 2019). An independent variable is significant if it has $Pr(< \chi^2) < 0.05$, where $Pr(< \chi^2)$ is the p-value from the χ^2 statistical test. Its associated χ^2 value indicates whether the model is statistically different from the same model in the absence of the independent variable. A larger χ^2 value indicates that the independent variable has a stronger explanatory power of the rerun of successful jobs.

7.5.2.3 Results

We report the model performance in Table 7.4. Table 7.5 shows, for each independent variable, the model coefficient (i.e., the log-odds), the χ^2 test results, and the odds ratio (OR = $e^{log\ odds}$). Significant variables are marked with asterisks. Positive or negative log-odds indicate that the variable has a direct relationship (i.e., an increase in the variable increases the likelihood of rerunning successful jobs) or an inverse relationship, respectively. In the following, we discuss the performance of our model, and for synthesis, the most significant and practically relevant factors that are identified.

Table 7.4 Mixed-effects logistic regression model performance

ROC AUC	**AUC-PR**	**Brier Score**	R^2**m**	R^2**c**
0.851	0.496	0.074	0.306	0.367

Our model demonstrates strong discriminatory power, a balanced precision–recall trade-off, and well-calibrated predicted risk estimates. As shown in Table 7.4, our mixed-effects model achieves an excellent ROC AUC of 0.85, far exceeding the 0.5 expected from random guessing, indicating a strong ability to distinguish between rerun and non-rerun successful jobs. Its AUC-PR of 0.496 substantially exceeds the baseline value of 0.1107, demonstrating effective identification of positive instances while limiting false positives, particularly important in our imbalanced dataset context. Additionally, the low Brier score of 0.074 reflects well-calibrated predictions. Moreover, the marginal R^2 of 0.306 (above 0.2) indicates an excellent model fit. This value approximately represents 83% of the conditional R^2 of 0.367, meaning that fixed effects alone account for 83% of the explained variance in our model. Furthermore, we obtained

an EPV of 369, a value far exceeding the threshold of 10, which provides strong evidence that the risk of model overfitting is negligible.

Table 7.5 Results of our mixed-effects logistic model

F.	Factor	Coef.	χ^2	Pr($<\chi^2$)	S.[+]	OR
Job Execution Context	(Intercept)	−2.806	234.71	$< 2.2e^{-16}$	***	0.06
	job_type_code_and_image_build	0.124	65.9	$4.763e^{-16}$	***	1.13
	job_type_image_security_scan	−0.281	229.1	$< 2.2e^{-16}$	***	0.75
	job_type_notification	0.091	43.9	$3.403e^{-11}$	***	1.10
	job_type_other	0.031	5.8	0.0163	**	1.03
	job_type_static_code_analysis	0.399	500.0	$< 2.2e^{-16}$	***	1.49
	job_type_testing	0.636	1394.7	$< 2.2e^{-16}$	***	1.89
	branch_type_feature	−0.642	2063.2	$< 2.2e^{-16}$	***	0.53
	branch_type_hotfix	−0.343	730.9	$< 2.2e^{-16}$	***	0.71
	branch_type_main	−0.011	1.1	0.3019		0.99
	branch_type_release	−0.243	387.5	$< 2.2e^{-16}$	***	0.78
	month_of_year_sin	0.013	1.2	0.27296		1.01
	month_of_year_cos	0.082	42.1	$8.627e^{-11}$	***	1.09
	day_of_week_sin	−0.026	4.8	0.0277	*	0.97
	day_of_week_cos	−0.010	0.6	0.4218		0.99
	hour_of_day_sin	0.003	0.1	0.7917		1.00
	hour_of_day_cos	−0.007	0.3	0.5781		0.99
Project (History) Factors	lang_gherkin	−0.036	0.2	0.6563		0.96
	lang_go	0.004	0.0	0.9055		1.00
	lang_hcl	0.048	5.8	0.0160	*	1.05
	lang_html	−0.153	4.1	0.0429	*	0.86
	lang_javascript	0.054	1.4	0.2421		1.06
	lang_jinja	−0.077	0.6	0.4451		0.93
	lang_shell	0.156	11.5	0.0007	***	1.17
	lang_tsx	0.073	0.5	0.4916		1.08
	lang_typescript	−0.177	5.1	0.0236	*	0.84
	rerun_ratio	0.273	123.5	$< 2.2e^{-16}$	***	1.31
	failure_ratio	−0.069	7.0	0.0082	**	0.93
	build_frequency	0.001	0.0	0.9644		1.00
	commits	0.120	28.3	$1.026e^{-07}$	***	1.13
Dev.	author_exp_days	0.008	0.3	0.5891		1.01
	author_exp_commits	−0.151	85.3	$< 2.2e^{-16}$	***	0.86
	author_recent_rerun_history	0.774	6239.2	$< 2.2e^{-16}$	***	2.17

[+]Significance codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

The most important factors influencing the likelihood of rerunning successful jobs and associated with higher odds include job type (especially testing and static code analysis jobs) and projects written in scripting-heavy languages such as Shell. As shown in

Table 7.5, common wisdom factors (i.e., `author recent rerun history`, `rerun ratio`, and `#commits`) present highly significant evidence (p-value < 0.001) of association with the rerun of successful jobs. In particular, the `author recent rerun history` has the highest χ^2 value among all the studied independent variables. Its odd ratio suggests that for every one-unit increase, there is a predicted 117% average increase in the odds of rerunning successful jobs, controlling for all other independent variables.

More interestingly, we observe that several other less obvious factors have a significant influence on the likelihood of rerunning successful jobs. In fact, nearly all dummy variables for job type and branch type show strong evidence of association with success reruns. Specifically, testing and static code analysis jobs, along with feature and hotfix branches, have the four highest χ^2 values among the dummy variables, indicating very high significance. Based on the odds ratios, successful testing and static code analysis jobs are predicted to have, on average, 89% and 49% higher odds, respectively, of being rerun compared to deployment and release jobs (the reference category), controlling for all other independent variables. Conversely, successful jobs executed on feature and hotfix branches are 47% and 29%, respectively, less likely to be rerun compared to jobs executed on develop branches.

Among the periodic variables, the cosine component of the month of year is highly significant (p-value < 0.001), suggesting a seasonal trend. To interpret its influence, we plot in Fig. 7.7 its response curve to analyze how the predicted probability of rerunning successful jobs evolves as the months vary, controlling for all other variables. As can be observed, successful jobs executed at the end or beginning of the year are predicted to have higher odds of being rerun, on average, compared to those executed in the middle of the year. While this seasonal trend is statistically significant ($\chi^2 = 42.1$, p-value < 0.001), we note that its practical effect size is modest (OR = 1.09) compared to factors such as job type (OR up to 1.89) or author recent rerun history (OR = 2.17). We retain temporal features because they reveal a meaningful pattern for monitoring purposes, but practitioners should prioritize addressing job-type and author-level factors for greater impact.

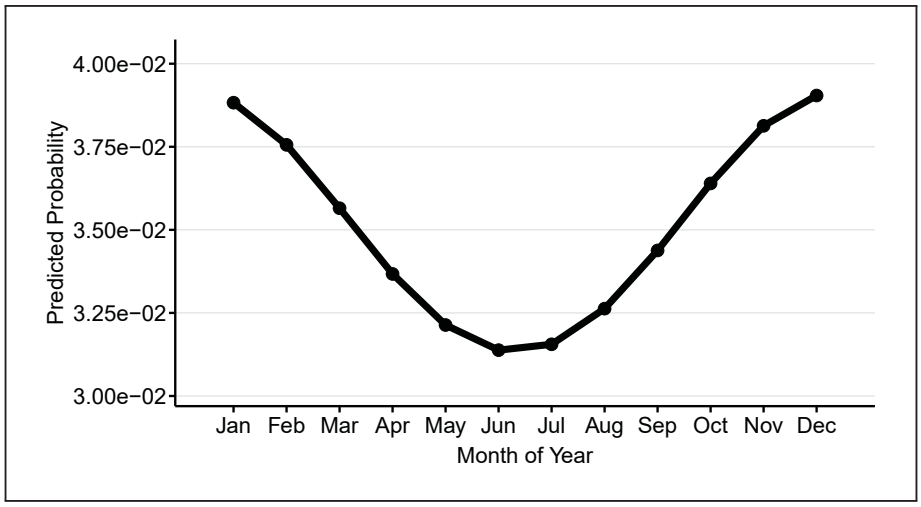


Figure 7.7 Month of year vs. the probability of rerunning successful jobs

At the project level, we note that both Shell and HCL languages are significant and associated with higher odds. In particular, successful jobs executed in Shell and HCL projects are predicted to have, on average, respectively 17% and 5% higher odds of being rerun than those in Python projects, controlling for all other independent variables. The higher odds for Shell and IaC-heavy projects could reflect multiple factors, including their limited IDE support for catching bugs, their prevalence in tasks that interface with external systems (caches, artifact repositories, deployment environments), and the common scripting practice of not enforcing strict exit code checking (e.g., omitting `set -e`) that allow failures to go undetected. These explanations remain hypothetical and warrant further investigation to establish causality.

Other key factors associated with lower odds of rerunning successful jobs include execution context (e.g., feature, hotfix, and release branches), project failure ratio, and developer experience in commits contribution. As shown in Table 7.5, successful jobs executed on feature, hotfix, and release branches are highly significant (p-value < 0.001) and predicted to have, on average, respectively 47%, 29% and 22% lower odds of being rerun compared to those executed on develop branches. These results are consistent with the GitFlow branching model adopted at TELUS. As the central integration branch, `develop` is where all feature branches converge, leading to a higher frequency of CI jobs and a greater exposure to unexpected issues

such as caching conflicts, artifact inconsistencies from parallel contributions, and environmental instabilities. In contrast, `feature` branches represent more isolated development contexts, where a single developer typically works independently on a specific change. This isolation reduces exposure to the types of intermittent and integration-related problems that can cause silent failures, which may explain the 47% lower odds observed for feature branches. Similarly, `hotfix` and `release` branches focus on well-defined, targeted changes (urgent fixes and stabilization, respectively), further reducing the likelihood of encountering unexpected silent issues compared to the more dynamic `develop` branch. Similarly, for every one-unit increase in the project's failure ratio (p-value < 0.05) there is a predicted 7% average lower odds of successful jobs being rerun. This association may indicate that projects with higher failure ratios benefit from more explicit error signals, potentially reducing the occurrence of unnoticed failures that would otherwise prompt the rerun of successful jobs. However, this interpretation remains a hypothesis that warrants further investigation. Finally, successful jobs authored by developers with greater commit contributions to the project, tend to have lower odds of being rerun. Specifically, for each one-unit increase in the commit ratio of a successful job's author, the odds of its rerun are predicted to decrease on average by 14%, holding all other independent variables constant.

Summary of RQ_2. Successful jobs are more likely to be rerun when they are testing and static code analysis jobs, or belong to projects dominated by scripting and infrastructure-as-code languages such as Shell and HCL. In contrast, the odds of rerunning successful jobs decrease when they are on feature, hotfix, or release branches, in projects with higher failure ratios, or triggered by more experienced contributors.

Replication of RQ_2 on Open-Source Projects. Our replication on 8 OSS projects (Section 4 of Appendix IV) yielded a model with even stronger discriminatory power (ROC AUC of 0.976, AUC-PR of 0.849, Brier Score of 0.021), confirming that the rerunning of successful jobs is driven by distinct factors. Several variables exhibit consistent trends across both contexts, including testing and static code analysis job types being associated with higher rerun odds. However, some variables show opposite effects (e.g., `commits`), which we attribute to contextual differences between industrial microservices and larger OSS projects. These results support

the generalizability of our core finding that specific, identifiable factors are associated with the rerun of successful jobs.

7.5.3 RQ_3. What types of failures occur silently within jobs marked as successful

7.5.3.1 Motivation

In this RQ, we seek to explore the root cause of silent job failures and existing solutions discussed by developers. The findings will provide valuable insights, complementary to the key influential factors identified in RQ_2, to help practitioners better understand the nature of silent failures that occur during job executions marked as successful and find initial solutions to these issues.

7.5.3.2 Approach

To answer this RQ, we conduct a thematic analysis of practitioner-reported issues on GitLab. We focus on publicly reported issues to gain broader insights beyond the studied industrial context and to ensure the generalizability of our findings. Inspired by prior work (Weeraddana *et al.*, 2024a), we first systematically search for relevant public issues related to silent failures.

The search queries used include:

- `silent (job OR build OR pipeline OR CI) failure`
- `(job OR build) succeeds but fails`
- `(job OR build) fails but still succeeds`

These queries capture our definition of silent job failures corresponding to jobs that are marked as successful but in reality fail to complete their task. Our search was conducted on the official GitLab issue tracker[16], which covers both the GitLab platform and the GitLab Runner projects. We focused on this single data source because (1) our study is centered on GitLab CI, the most widely adopted self-hosted CI platform, (2) practitioners commonly report CI-related issues

[16] https://gitlab.com/gitlab-org/gitlab/-/issues

on the official tracker, and (3) restricting to one platform ensures consistency in the taxonomy. We intentionally run our search on all issues regardless of status (i.e., open/closed) or recency. Indeed, practitioners are often reluctant to make major migrations (Aïdasso *et al.*, 2025a) and continue to rely on older versions of tools; therefore, they may encounter or learn from previously resolved problems. Hence, we manually screen issues in the search results, examining their titles and descriptions to retain only those explicitly describing scenarios where failures occurred during CI job execution, but the job was ultimately marked as successful.

This filtering process resulted in 92 relevant issues identified, including 1,199 human-written comments. Then, using the open coding (Khandkar, 2009) technique, we examine in a first iteration, the title, description, and comment threads of each issue to identify recurring themes (or codes) related to the causes and contexts of the reported silent failures. At the end, we further refine the identified themes by grouping similar topics and renaming them to account for their new scope. In a second iteration, the first two authors independently assigned themes to issues, resolving disagreements through discussion. To assess the consistency of our categorization, we calculated Cohen's Kappa score (Cohen, 1968), which yielded a value of 0.913, indicating almost perfect agreement. In addition, we identified and summarized the solutions or workarounds proposed by practitioners and CI provider engineers for each theme. Hence, we provide insights into both the problem space and available remediation strategies. The detailed list of the 92 issues with their links is included in our replication package (Aïdasso *et al.*, 2025d).

In addition, we coded each issue for the job type(s) associated, using the same seven types defined in Section 7.4.4. Job types were identified from explicit mentions in issue descriptions and comments, or inferred from referenced artifacts (e.g., coverage reports → testing, DAST reports → security scanning). Issues where no specific job type could be determined were marked as "unspecified." An issue could be assigned multiple job types when applicable.

7.5.3.3 Results

We identified 11 themes for silent job failures, listed in Table 7.6, along with brief descriptions, concrete examples drawn from the referenced public issues, and proposed solutions or workarounds. We elaborate on the themes below.

T_1. Artifact Operation Errors. This theme, the most commonly observed, includes failures related to artifact handling, such as uploading, downloading, and unexpected issues like retrieving outdated artifact versions. As noted in issues[17][18], developers expect these errors to cause immediate job failure rather than being suppressed, since they can lead to serious problems ranging from downstream job failures to bugs reaching production.[17] Artifacts often include critical resources such as unit tests or security reports, making it essential that upload or retrieval failures are noticed right away.[18]

T_2. Caching Errors. The second most common theme is cache-related failures (such as connection errors, ignored empty keys, and issues with expired or incomplete cache files) that occur without feedback to developers. These problems often stem from local cache misconfigurations[19], connection issues with external storage like Google Cloud Storage[20], and concurrency conflicts in parallel jobs sharing the same cache.[21] They cause downstream failures due to missing or corrupted caches, forcing manual reruns of early cache-producing jobs; a process that involves up to 4 wasted minutes on each cache creation step.[21]

T_3. Ignored Non-Zero Exit Codes. Another common theme includes cases where, either a job script returns an error code[22] (i.e., a non-zero code) or an explicit failure command is executed[23] (e.g., `exit code 1`), but the job ignores this non-zero exit code and still reports success to the

[17] https://gitlab.com/gitlab-org/gitlab/-/issues/22711

[18] https://gitlab.com/gitlab-org/gitlab/-/issues/33566

[19] https://gitlab.com/gitlab-org/gitlab/-/issues/291052

[20] https://gitlab.com/gitlab-org/gitlab-runner/-/issues/4127

[21] https://gitlab.com/gitlab-org/gitlab/-/issues/291052

[22] https://gitlab.com/gitlab-org/gitlab-runner/-/issues/26877

[23] https://gitlab.com/gitlab-org/gitlab-runner/-/issues/26538

Table 7.6 Themes and solutions identified for silent job failures

ID	Theme (#)	Solutions/workarounds
T_1	Artifact Operation Errors (26)	• Add dependencies between artifact-producing and consuming jobs • Increase the maximum artifact size for uploads in settings • Fix local paths and permanent storage configurations • Check artifacts' presence and explicitly fail the job
T_2	Caching Errors (21)	• Check storage access and retention policy configurations • Clean up cache storage disk space • Commit cache files separately to avoid overrides on same commit • Upgrade the cache management system version
T_3	Ignored Non-Zero Exit Codes (17)	• Set `GIT_STRATEGY=clone` in build configuration to prevent repository cloning issues • Avoid `exit` calls in *if-else* blocks of Powershell scripts • Add `set -e` to scripts to enforce exit code checks • Change `./script.sh` with `source ./script.sh` in Dockerfiles
T_4	Infrastructure Config. Issues (10)	• Increase the worker timeout • Check or increase worker task queue size limits • Set pod limits and requests in k8s managed runners
T_5	Security Scan Failures (7)	• Install missing *analyzer* dependencies before running the script • Upgrade or change the scanning tool or library
T_6	Skipped Script Execution (7)	• Check overrides of container image entrypoints • Update runner with a non-buggy version (noa, 2021)
T_7	Post-deployment Issues (6)	• Check deployment environment configs. (e.g., env. name, URL) • Add post-deployment steps to check container health
T_8	Ignored Test Failures (3)	• Implement post-testing steps to verify test failures from reports • Explicitly set proper error level in *WinBatch* test scripts
T_9	Silent Curl Errors (3)	• Remove the `-silent` or `-s` option silencing curl errors • Add an `OR` operator to the problematic script to send email notification on failure
T_{10}	Invalid CI Variables (2)	• Check variable scopes (e.g., protected branches only) • Enable or disable variable expansion
T_{11}	Syntax Errors (1)	• Avoid including partial CI configurations in separate files

developers. Such an issue has been reported as "driving [. . .] team crazy after weeks searching for a fix"[23] and can lead to highly problematic scenarios where deployments are unsuccessful but remain unnoticed[22] for long periods. Migrating to a non-buggy runner version has also been suggested.[24]

T_4. Infrastructure Config. Issues. This theme captures issues rooted in the configuration of the underlying CI infrastructure. Examples include setting overly restrictive size limits on runner

[24] https://gitlab.com/gitlab-org/gitlab-runner/-/issues/36928

workers, which prevents jobs from executing; or Kubernetes (k8s) cluster issues that abruptly terminate jobs mid-execution.[25] This latter issue is particularly wasteful to organizations since "every day, several times an hour, [developers] have to manually restart the pipelines to solve the problem".[25] Overall, the infrastructure-related problems often overlap with other themes. For instance, an incorrect runner setup can lead to cache upload failures.[26]

T_5. Security Scan Failures. In some job cases, scans of software dependencies, licences, or image vulnerabilities fail (i.e., indicate issues), while the job still signals success. Such issues can lead to important code vulnerabilities being introduced into production. As for the causes, missing dependencies in the analyzer tool itself[27] have been reported, which can be installed in the `before_script` section of the CI configuration file. Other issues highlighted buggy scanning tools that developers should upgrade or change to solve the problem.

T_6. Skipped Script Execution. Described as a problem "which can [have] quite disastrous results [in production]"[28], this theme encompasses cases where job scripts or triggers are either partially or completely skipped, yet the job is still marked as successful. Under the illusion of success, teams may falsely certify code quality, let major bugs slip through to production, or announce releases without any actual deployment. These can have major consequences, from substantial costs to address the issues later on to a loss of credibility with end-users. The root causes mainly include misconfigured container image entrypoints and buggy runner versions.

T_7. Post-deployment Issues. This theme refers to problems that silently occur after a deployment job is marked as successful. It notably includes failures in creating or updating deployment environments[29], as well as cases where containers are marked as successfully deployed but are not actually available in the target environment.[30] In particular, problems with environment

[25] https://gitlab.com/gitlab-org/gitlab-runner/-/issues/4119

[26] https://gitlab.com/gitlab-org/gitlab-runner/-/issues/37513

[27] https://gitlab.com/gitlab-org/gitlab/-/issues/347479

[28] https://gitlab.com/gitlab-org/gitlab/-/issues/13000

[29] https://gitlab.com/gitlab-org/gitlab/-/issues/16586

[30] https://gitlab.com/gitlab-org/gitlab-runner/-/issues/25988

creation or updates are often the result of configuration errors (e.g., in the environment name or URL).

T_8. Ignored Test Failures. This theme is a subset of silent failures due to ignored non-zero exit codes, but is primarily related to test jobs. It includes cases where unit or integration tests fail, but the test results (and the non-zero exit code) are ignored[31], allowing the job to be marked as successful despite the failing test cases. Such silent failures pose a critical risk, as they may lead to the deployment of buggy code versions into production, which can, in turn, result in severe negative consequences for organizations.

T_9. Silent Curl Errors. We identified some instances of improper use of the `curl` command in the job specifications. A typical example[32] is the use of the silent option (`-s` or `--silent`) of the command, which causes jobs to report success despite the occurrence of underlying errors that should have been exposed (e.g., failure of an artifact upload via `curl`).

T_{10}. Invalid CI Variables. This theme regroups silent failures related to misconfigured and invalid CI/CD variables. Examples include silent issues that arise from incorrect variable settings (e.g., scopes limited only to protected branches) or from unexpected behavior when using self-referencing variables without enabling variable expansion.[33]

T_{11}. Syntax Errors. Issues associated with this theme concern errors that arise when parts of the main CI configuration file (i.e., partial build configurations) are saved in separate files.[34] Such partial build configurations are therefore not validated at submission before execution, allowing syntax issues to slip through unnoticed and cause silent failures. These CI misconfiguration issues highlight the need for more intelligent developer support tools to manage the growing complexity of CI configurations and prevent reliability issues.

---

[31] https://gitlab.com/gitlab-org/gitlab/-/issues/501738

[32] https://gitlab.com/gitlab-org/charts/gitlab/-/issues/3145

[33] https://gitlab.com/gitlab-org/gitlab/-/issues/556631

[34] https://gitlab.com/gitlab-org/gitlab/-/issues/340541

Table 7.7 Cross-tabulation of silent failure themes (RQ_3) and job types identified from issue descriptions

Job type	**T_1**	**T_2**	**T_3**	**T_4**	**T_5**	**T_6**	**T_7**	**T_8**	**T_9**	**T_{10}**	**T_{11}**	**Total**
code_and_image_build	15	9	6	2	–	3	–	–	1	–	–	36
testing	7	8	8	1	–	1	–	3	–	–	–	28
deployment_and_release	–	1	1	1	–	1	6	–	–	1	–	11
image_security_scan	1	–	–	–	7	–	–	–	–	–	–	8
static_code_analysis	3	2	–	1	–	–	–	–	–	–	–	6
notification	–	–	–	–	–	–	–	–	1	–	–	1
unspecified	8	8	3	6	–	2	–	–	1	1	1	30
Total	**26**	**21**	**17**	**10**	**7**	**7**	**6**	**3**	**3**	**2**	**1**	**103**

To explore the connection between the identified silent failure themes and the influential factors of successful job reruns from RQ_2, we cross-tabulate themes with the job types reported in each issue. We focus on job type because it is the most significant categorical factor in RQ_2 and the most reliably identifiable from issue reports, unlike factors such as branch type or developer experience. Job type was identifiable in 64 of 92 issues (70%). Table 7.7 presents the results.

Several patterns emerge that converge with the RQ_2 findings. Build and testing jobs concentrate in the three most frequent themes: T_1. Artifact Operation Errors (22 of 26), T_2. Caching Errors (17 of 21), and T_3. Ignored Non-zero Exit Codes (14 of 17). This is consistent with the RQ_2 finding, where testing and static code analysis jobs exhibit the highest rerun odds (OR of 1.89 and 1.49, respectively), as these job types are the primary producers and consumers of artifacts and caches. As expected, Security scan failures (T_5) map exclusively to security scan jobs (7 of 7), Post-deployment issues (T_7) exclusively to deployment and release jobs (6 of 6), and Ignored test failures (T_8) exclusively to testing jobs (3 of 3). Nevertheless, Infrastructure config. issues (T_4) are predominantly unspecified (6 of 10), reflecting their cross-cutting nature as platform-level problems. Overall, the concentration of build and testing jobs in the most frequent silent failure themes aligns with their higher rerun odds in RQ_2, providing empirical support for the connection between the quantitative and qualitative findings.

Summary of RQ$_3$. We identified 11 themes related to silent failures, including cache and artifact operation issues (T_1, T_2); failures within CI configuration files, such as ignored test and security scans (T_3, T_5, T_8); infrastructure problems that disrupt job completion (T_4, T_6); and misconfigurations or misuse of CI tools (T_7, T_9, T_{10}, T_{11}) leading to missed alerts, escaped bugs, and unnoticed deployment failures.

7.6 Discussion

Practitioners should include post-execution verification tasks as a standard step in CI pipelines to validate the success of business-critical jobs that produce artifacts or perform environment changes such as deployments. As determined in RQ$_2$, the most important factors influencing the rerun of successful jobs include specific job types that either produce artifacts (e.g., testing and static code analysis jobs) or perform environmental changes (e.g., notification and image build jobs). These findings align perfectly with the most common causes of silent failures identified in RQ$_3$, including artifact operation errors and caching errors, but also ignored test failures and post-deployment issues, which do not result in actual job failures, even though developers expect such issues to be explicitly surfaced. Indeed, these silent failures can have catastrophic consequences for organizations. As an example, falsely successful testing jobs can lead to bugs escaping into production, affecting software quality, user experience, and ultimately the organization's reputation. To anticipate these problems, we recommend that practitioners systematically add verification steps (e.g., jobs) immediately after jobs whose successful and reliable executions are critical to the business, in order to ensure the presence of the expected artifacts or environmental changes (e.g., test reports, cache updates, new container image or software versions). In this way, these downstream jobs can detect silent failures and provide timely feedback to developers through explicit failures.

For the specific studied industrial context, we observed additional factors that can help raise awareness of silent failures across teams to anticipate and mitigate them. In particular, the association between Shell/HCL projects and higher rerun odds may stem from multiple factors, including limited IDE assistance for early error detection and common scripting pitfalls such

as missing `set -e` directives and ignored exit codes in *if-else* blocks (see T_3 and T_9 in RQ_3). Teams working with scripting and infrastructure-as-code languages should review their scripts for such patterns to improve reliability.

In summary, based on the findings of this study, we recommend that practitioners adopt the following measures to mitigate silent failures:

- **Validate job outputs.** Add post-execution steps after artifact-producing and deployment jobs (including build, testing, and static analysis jobs) to validate expected outputs.
- **Enforce strict error handling.** Use `set -e` in Shell scripts to ensure immediate exit on first command failure and review exit code handling in conditional blocks.
- **Audit CI configurations.** Periodically review cache configurations and artifact retention policies on self-hosted CI instances.
- **Prioritize high-risk jobs.** Monitor testing and static analysis jobs closely to understand better silent failure modes related to these jobs, as they have the highest observed rerun odds.
- **Invest in developer training.** Support developers, especially less experienced ones, through onboarding and regular training on CI reliability practices.

Our findings highlight important root causes that CI providers should prioritize to prevent silent failures and enhance the reliability of CI systems. We identified several infrastructure-related factors that influence the rerun of successful jobs, as well as the most common causes of silent failures that prompt developers to trigger such reruns. In particular, caching issues, artifact operation errors, and ignored failure exit codes are the most common. Still, they can be mitigated through enhanced validation mechanisms that we recommend CI providers implement in future versions. For example, CI providers could introduce a "required artifacts" directive in the CI configuration, which would trigger automatic post-execution checks and raise an explicit failure when expected artifacts are missing. Similarly, CI providers should provide configuration options that clearly signal failures related to cache or artifact repository storage access, rather than silencing them in seemingly successful jobs. As part of the root causes, we also identified several CI configuration smells (e.g., short worker timeout, small queue size, or

partial configuration file saved outside the CI configuration file). We recommend that these smells be leveraged to improve validation of CI configuration files, both within IDEs and at submission time, to prevent silent failures. In this regard, several of our RQ_3 themes, particularly Ignored Non-zero Exit Codes (T_3), Invalid CI Variables (T_{10}), and Syntax Errors (T_{11}), can be viewed as CI configuration smells that produce fake successes, as conceptualized by Vassallo et al. Vassallo, Proksch, Jancso, Gall & Di Penta (2020b). Extending linting tools such as CD-Linter Vassallo *et al.* (2020b) with detection rules targeting these patterns can help anticipate a subset of the silent failures we identified.

Researchers should propose approaches for automated detection of silent failures, to assist developers in identifying them early, so as to reduce the costs of job reruns. Our mixed-effects model of jobs from 81 projects shows strong discriminatory power (ROC AUC of 85%), is well calibrated (Brier score of 0.074) and achieves an excellent goodness-of-fit score (marginal R^2 of 0.306). These results suggest that the studied variables capture meaningful variation between rerun and non-rerun successful jobs. We emphasize that the primary goal of our model is *explanatory*: i.e., to identify the most influential factors associated with the rerun of successful jobs. We did not design, tune, or evaluate the model for predictive deployment (e.g., no train/test split or temporal validation was used). Nonetheless, our explanatory findings could serve as a starting point for future predictive work. We invite future studies to leverage the identified factors and root causes to develop dedicated detection models for silent failures, using appropriate predictive evaluation protocols (e.g., temporal cross-validation).

7.7 Threats to Validity

Construct Validity. Threats to construct validity concern the degree to which our study measures what we claim to be analyzing. The most important threat of this kind is the identification of silent job failures, which we have studied in this paper, through successful jobs that have been rerun. Our main hypothesis is that developers rerun successful jobs only when these jobs have not fully or correctly completed their intended task. Such a hypothesis disregards cases where a truly successful job is still rerun, such as, for instance, the rerun of a deployment job after a

scheduled environment cleanup. Nevertheless, TELUS engineers have confirmed our hypothesis in the studied industrial context, and we expect these cases to be very rare in other contexts since rerunning successful jobs does not align with effective CI practices (Elazhary *et al.*, 2022).

Our grouping strategy by (job name, commit SHA, project ID) may also capture reruns unrelated to silent failures, such as jobs re-executed after infrastructure maintenance operations. Since our definition requires the same commit SHA, maintenance operations that modify any version-controlled file, including the CI configuration file, would produce a new commit and not be grouped as reruns. A developer could still manually rerun a successful job to verify the environment or redeploy services after an infrastructure intervention. However, at TELUS, infrastructure is managed by a specialized team separate from the developers who typically rerun jobs, making such overlapping cases infrequent.

Another threat to construct validity comes from our use of open coding, which involves manual inspection and subjective judgment. In RQ_2, we derived job types from job names using regex-based classification. While this process may introduce errors, we carefully selected keywords that clearly indicate the job's category, thereby reducing the risk of mislabeling. To mitigate this threat in RQ_3, where the assignment of themes was more interpretive, the first two researchers independently classified the GitLab issues. We then calculated an agreement score of 0.913, which shows almost perfect agreement. We make all related artifacts publicly available (Aïdasso *et al.*, 2025d), enabling verification and supporting replication of our results.

Finally, the 92 public issues (1,199 comments) analyzed in RQ_3 were retrieved exclusively from the GitLab issue tracker, which may over-represent problems specific to the GitLab ecosystem and issues that are sufficiently severe or visible to be publicly reported. Silent failures that are resolved internally without a public report are not captured. Nevertheless, publicly reported issues represent real problems encountered by practitioners, lending practical relevance to our findings. Furthermore, the platform-agnostic nature of most identified themes (e.g., ignored exit codes, caching errors, artifact issues) suggests broader applicability beyond GitLab.

Internal Validity. The main internal threat to validity lies in the omission of confounding factors that may influence our interpretation. Indeed, we did not explore all the factors that could be associated with the re-execution of successful jobs, such as the CI server characteristics or variables indicating whether the job used cache. Such variables could better explain the likelihood of rerunning successful jobs. However, these data are either not openly available or require feature engineering too specific to the CI service, which would pose a threat to generalizability. Besides, code change intent before reruns could influence the prevalence of silent failures and warrants investigation in future commit-level analyses. Nevertheless, we selected 14 primary factors (resulting in 32 independent variables) across three dimensions based on existing literature studying factors associated with build failures (Islam & Zibran, 2017; Rausch *et al.*, 2017; Dimitropoulos *et al.*, 2017; Barrak *et al.*, 2021) and build performance (Weeraddana *et al.*, 2024a; Ghaleb *et al.*, 2019; Silva *et al.*, 2023; Kola-Olawuyi *et al.*, 2024). Also, the purpose of this study is to highlight the factors associated with rerunning successful jobs so as to raise awareness within industrial development teams. We encourage future research to investigate the causal relationships among these factors.

On the other hand, the variable selection following correlation analysis may have influenced the results of our mixed-effects models. To mitigate this concern, we provided a detailed rationale for the choice of variable in each pair of correlated variables. Moreover, we provide a replication package (Aïdasso *et al.*, 2025d) to facilitate the reproducibility of our results.

External Validity. We built our mixed-effects models with data from projects using GitLab CI. As such, some of our findings are inherently tied to GitLab CI's design, such as the specific artifact and cache handling mechanisms (e.g., `artifacts:when` semantics, cache key strategies) and infrastructure issues specific to certain runner configurations (e.g., Windows-based runners). However, the core findings of this study extend beyond any single CI platform. Silent failures, by nature, arise from the gap between a job's reported status and its actual outcome, a problem common to all CI systems, although we argue that silent failures might be more prevalent in industrial settings where CI services and tools (e.g., artifact repositories) are self-hosted. Also, testing and artifact-producing jobs exist on every CI platform and remain susceptible

to silent issues due to their reliance on external resources such as caches, artifact stores, and test frameworks. Likewise, root causes such as ignored exit codes, scripting pitfalls, and misconfigured infrastructure are platform-agnostic. Furthermore, the variables used in our study (Table 7.2) are CI service-agnostic and computable from metadata available in most CI platforms. To facilitate replication on other CI services such as Jenkins or GitHub Actions, we have made our replication package publicly available (Aïdasso *et al.*, 2025d).

In addition, the focus on industrial projects poses another threat to external validity. To address this, we analyzed 81 projects that vary in size, age, and programming language (13 in total, including widely used ones such as Python, JavaScript, and Go). These projects serve diverse purposes, ranging from backend systems (e.g., APIs, API gateways, and network controllers) to front-end applications (e.g., business intelligence dashboards and web portals). TELUS also makes extensive use of widely adopted open-source tools such as Docker and Kubernetes. We believe that such diversity of project context and tools enhances the generalizability of our results beyond the context of this study. Additional details on the studied projects, including their purpose, age, dominant language, number of commits, jobs, etc., are provided in Appendix II. We have released the industrial dataset of 142,387 jobs to support open science. We also note that excluding projects with fewer than 50 commits may bias the findings toward more mature projects with established CI practices.

Nonetheless, we conducted a replication study on 198,146 jobs collected across 8 popular GitLab open-source projects to ensure the external validity of our findings. This dataset has been extracted from the GLBUILDS dataset[35] introduced in (Aïdasso, Bordeleau & Tizghadam, 2025e). Key replication results summarized in RQ1 and RQ2 (Sections 7.5.1 and 7.5.2), confirmed many of our findings, including the distinction between silent failures and true successes (AUC ROC of 97%, AUC PR of 85%) and the influence of factors such as testing and static code analysis jobs. Complete results and comparative discussions are provided in Appendix IV.

[35] https://doi.org/10.6084/m9.figshare.27641388.v1

7.8 Conclusion

One major challenge in CI systems is ensuring the reliability of outcomes, both failures and successes. While prior work (Aïdasso *et al.*, 2025a,b; Olewicki *et al.*, 2022; Lampel *et al.*, 2021; Durieux *et al.*, 2020) has focused on intermittent build failures, the problem of false job successes has been largely overlooked. These "silent failures" are often hard or impossible to trace using logs, and are potentially more common in industrial contexts with self-hosted CI. In this paper, we investigate silent failures through successful jobs that were later rerun. We found that over half of reruns follow successes, often after long delays (up to days), suggesting delayed detection of silent issues. Reruns of successful jobs are more likely for testing and static code analysis jobs, and in projects using languages with limited IDE support, such as Shell or HCL. Moreover, a thematic analysis of 92 related publicly available issues highlights caching and artifact errors as frequent causes of silent failures, typically in testing and static analysis jobs. Most common themes also include shell scripting issues (e.g., ignored non-zero exit codes) and CI misconfigurations, which link poor language support to silent failures. These findings provide insights to raise developers' awareness of silent failures, which go unnoticed while contributing to increased CI costs in organizations. We also summarized practical solutions to help practitioners mitigate these silent failures.

CHAPTER 8

TOWARDS BUILD OPTIMIZATION USING DIGITAL TWINS

Henri Aïdasso[1] , Francis Bordeleau[1] , Ali Tizghadam[2]

[1] Department of Software and IT Engineering, École de Technologie Supérieure,
1100 Notre-Dame Ouest, Montreal, Quebec, Canada H3C 1K3
[2] Network Softwarisation and AI, TELUS,
Toronto, Ontario, Canada

Article published in « ACM International Conference on Predictive Models and Data Analytics in Software Engineering » June 2025

8.1 Preamble

This chapter presents the CBDT Framework, a digital twin-inspired architecture for the continuous monitoring and improvement of CI build processes, published in ACM PROMISE 2025 (Aïdasso *et al.*, 2025e). While Chapters 4 through 7 investigate and develop individual solutions for detecting and diagnosing pipeline reliability issues, this chapter addresses the need for a unifying framework that orchestrates such solutions in production CI/CD environments. The CBDT Framework thus serves as the architectural backbone envisioned in the thesis introduction for integrating detection, diagnosis, and repair services into a cohesive multi-component system.

8.2 Introduction

The use and evolution of continuous integration and continuous deployment (CI/CD) pipelines come with several challenges that prior research addressed individually. CI challenges are mainly related to long build duration, frequent build failures, and intermittent non-deterministic builds (Aïdasso *et al.*, 2025f; Hilton *et al.*, 2017). These issues incur significant waste to organizations, including high CI costs, reduced developer productivity, and delayed software releases (Hilton *et al.*, 2017; Lampel *et al.*, 2021; Olewicki *et al.*, 2022). To address CI challenges, prior studies have proposed techniques for improving specific build performance metrics in isolation. For instance, Gallaba *et al.* (2022a) investigated strategies to accelerate build duration, Lou *et al.*

(2019) developed an automated repair technique to reduce build failures, while Olewicki *et al.* (2022) focused on detecting flaky builds to minimize costs associated with build flakiness.

Despite the proven benefits of the CI-improving solutions, practitioners remain reluctant to adopt them due to uncertainties associated with build process changes (Senapathi, Buchan & Osman, 2018). In fact, build performance metrics (e.g., duration, failure, flakiness) are interrelated in such a way that process changes made to improve one may inadvertently degrade another. As evidence, Ghaleb *et al.* (2022) found that while enabling cache may speed up build duration, it can also lead to failures due to dependencies becoming obsolete. Hence, there is a strong need for practitioners to **(1) monitor** the initially sought performance improvement when adopting a solution, **(2) analyze** the impact of process changes on other performance metrics, and **(3) optimize** multiple performance metrics in a single framework. Such challenges align with Lehman's law stipulating that software systems are "*...multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement...*" (Lehman, 1979). Yet, to our knowledge, no prior study has approached the continuous improvement of CI build processes from a holistic standpoint, which makes the need to fill this gap long overdue.

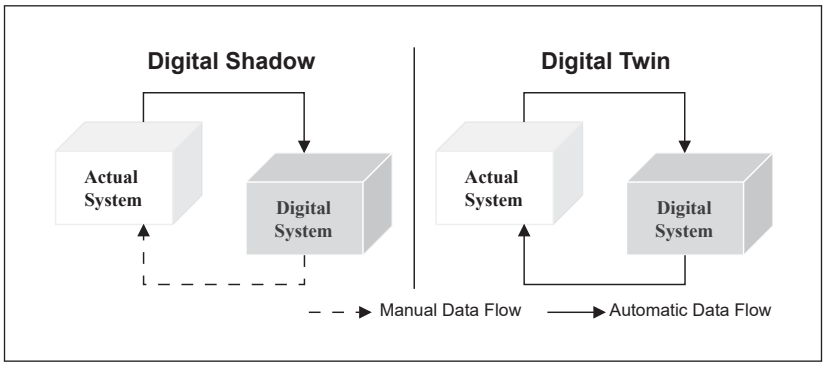


Figure 8.1 Concepts of Digital Shadow and Digital Twin

Digital Twins (DTs) have emerged as a key technology that enables real-time monitoring and optimization of complex systems, referred to as the Actual Twin (AT). Glaessgen & Stargel (2012) defined a DT as a multiscale and dynamic representation of a complex system whose functions are to mirror the life of its corresponding AT. As such, the synchronization between the DT and its AT is fundamental and is achieved through the automated integration of real-time data

from the AT. As illustrated in Fig. 8.1, this level of implementation forms the Digital Shadow (DS), which supports continuous monitoring and enables manual improvement feedback to the AT. In addition, the DT includes mathematical models that provide intelligent capacities used to automate the feedback loop for improvement of the AT. As a result of their effectiveness, DTs have been used to optimize a broad range of cyber-physical systems such as aircraft, autonomous vehicles, wind farms, and manufacturing processes. Furthermore, several DT frameworks such as Eclipse Ditto have been developed and open-sourced to facilitate the implementation of DTs, but mainly in IoT (Gil, Mikkelsen, Gomes & Larsen, 2024).

This chapter presents the following contributions: (1) a novel idea of developing CI Build process Digital Twins (CBDTs) to continuously monitor and improve continuous integration build processes across multiple performance metrics, (2) a context-agnostic framework with digital shadowing capabilities to demonstrate the feasibility of this idea and facilitate CBDTs' development, and (3) guidelines and challenges for developing CBDTs in practice. A replication package including the CBDT framework's source code and a 25 GB dataset of build jobs integrated from the 10 most popular GitLab projects is made publicly available.[1]

8.3 CI Build Digital Twin (CBDT)

Fig. 8.2 illustrates the architecture of the proposed CBDT for continuous monitoring and global improvement of the CI Build Process (AT). In organizations, this AT mainly encompasses the Version Control System (VCS) and the associated CI Platform. The objectives and conceptual design of the CBDT are presented below.

8.3.1 Optimization Objectives

A digital twin is developed with clearly defined optimization objectives, namely the key performance metrics that guide improvement efforts. Based on a recent systematic literature

[1] https://doi.org/10.6084/m9.figshare.27641388

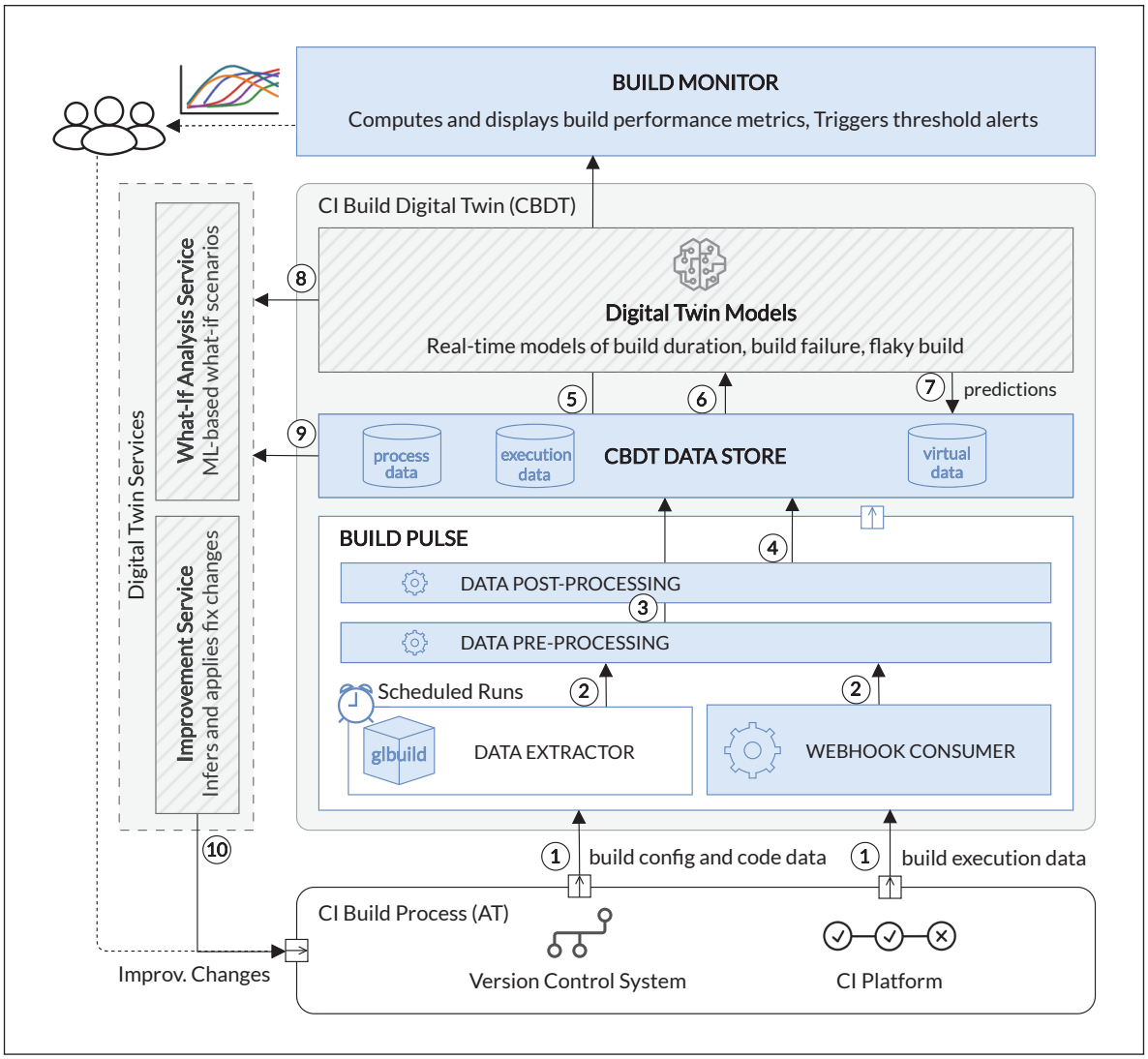


Figure 8.2 Architecture of the CI Build Digital Twin (CBDT)

review (Aïdasso *et al.*, 2025f), we identify four primary performance metrics for global build optimization, presented below.

- **Execution frequency** measures how often builds are triggered. A higher frequency indicates more frequent software delivery but also higher infrastructure costs.
- **Mean duration** captures the average time from build trigger to completion. Lower values indicate faster feedback, which is essential for developer productivity. An optimal baseline is approximately 10 minutes (Hilton *et al.*, 2017).

- **Failure ratio** is the proportion of failed jobs among all completed jobs. A lower ratio is desirable, as failures consume diagnosis time and delay software delivery.
- **Intermittent failure ratio** represents the percentage of failed jobs that succeeded upon retry without code changes. A higher ratio points to environmental instabilities and, therefore, associated waste.

These metrics should all be minimized for effective build optimization. More specific proxies (e.g., execution cost, energy consumption, carbon footprint) can be defined to align with organization-specific objectives.

8.3.2 Interactions and Collaboration

This section presents our conception of how the different components of the CBDT interact with the build process (AT) to enable its continuous monitoring and global improvement.

Fig. 8.2 highlights arrows annotated from ① to ⑩, indicating the main sequence of interactions between the CBDT components. First, data collection tools (in BUILD PULSE) continuously observe the AT via webhook events, and ① collect raw build execution and code-related data using APIs each time the AT state changes. The collected data are ② pre-processed before being ③ stored in the CBDT DATA STORE. After data integration, additional data fields required for ML models and visualization are ④ calculated at the post-processing stage and stored. Every time new build data are integrated, the newly available data are ⑤ queried by the monitoring service BUILD MONITOR to refresh the visualizations, and ⑥ by the DT Models for continual learning (Wang, Zhang, Su & Zhu, 2024c) of the ML models. These ML models make real-time predictions, which are also ⑦ stored for monitoring and analytics purposes discussed in Section 8.5. Furthermore, ⑧ replicas of the models and ⑨ the CBDT data are used to provide the additional DT services, including data-centric what-if analysis (Grafberger, Groth & Schelter, 2022) and improvement services such as ML-based build repair techniques that infer patches from historical failure repair patterns (Lou *et al.*, 2019). It is worth noting that data flows ⑥ and ⑨ follow a publish-subscribe messaging pattern, where BUILD PULSE acts as

the publisher, notifying data subscriber components via the CBDT Message Broker each time new data is integrated. For simplicity, this message broker is not illustrated. Finally, (10) the improvement changes (e.g., code patches) are automatically applied to the AT through its APIs for continuous improvement.

8.4 CBDT Framework

To validate the feasibility of our concept, we developed the CBDT framework as a minimum viable product (MVP) composed of the blue components in Fig. 8.2. The basic features of these components are context-agnostic and can be easily reused for different specific CBDT development objectives. The remaining components in **gray** require contextual implementations discussed in Sections 8.5 and 8.6.

At a minimum, the CBDT framework can be used by researchers and practitioners to obtain build data for their predictive models and analytics use cases. Unlike historical build data collection tools like TravisTorrent (Beller *et al.*, 2017b), this framework incorporates real-time data acquisition and distribution capabilities, facilitating the development of real-time prediction models. In its intended form, the CBDT framework provides basic digital shadow functionalities (i.e., real-time data synchronization and continuous monitoring) and forms the backbone for implementing and using CBDT models and services in diverse software project contexts.

8.4.1 Implementation Setup

To implement the CBDT framework, we focused on GitLab, a unified DevOps platform that incorporates both VCS and CI Platform and is used in many organizations worldwide. We verified the CBDT framework's functionalities by digitally shadowing the build process of the 10 most popular GitLab projects over the second half of 2024. As a result, we accumulated 25GB of data from 1.7 million build jobs. This dataset, along with the source code of the CBDT framework, including `Docker` images and `docker-compose.yaml` files to run the framework, is made publicly available in the replication package to foster reuse and development of CBDTs.

8.4.2 Framework Components

The CBDT framework is composed of five main microservices.

Build Pulse is the core Python application responsible for synchronizing data with the AT and distributing this data to the other CBDT components. To achieve data synchronization, it includes modules for historical and real-time build data acquisition.

Historical data integration is necessary to get a baseline of past performance and behaviors. For this purpose, we developed a lightweight Python library *glbuild*[2] that collects historical data from a set of GitLab projects. It retrieves and stores data from the complete build history, or up to a configurable limit on the number of historical jobs. Since highly active or long-lived projects can accumulate a vast amount of build data, this limit avoids the time and storage costs of collecting the oldest jobs. In addition, *glbuild* accepts an object parameter implementing the abstract class `AbstractStorage`, which handles data pre-processing and storage during collection. Pre-processing steps involve parsing raw data to defined data models, data cleaning and formatting (e.g., uniforming timestamps to UTC timezone). Post-processing steps are run to compute additional data fields for ML training. Currently, only the `flaky` field value is computed based on non-deterministic job reruns (Olewicki *et al.*, 2022).

Real-time data integration ensures that the CBDT remains synchronized with the AT. For this purpose, we implemented a webhook consumer endpoint to listen to the job webhook events[3] triggered every time a build job status changes. The consumer endpoint is secured using a token-based middleware to ensure that data is received only from the GitLab webhooks. Once the consumer receives a job event, API calls are made using the job ID to collect all the required build data. The obtained data then undergo the same pre-processing steps described for historical data before being stored, and the same post-processing steps afterward.

---

[2] https://pypi.org/project/glbuild/

[3] https://docs.gitlab.com/user/project/integrations/webhook_events

Configuring job webhook events in GitLab requires administrator access to the projects. To circumvent this requirement when working on open-source projects where such access is not available, we implemented scheduled integrations of historical data as illustrated in Fig. 8.2. These scheduled runs are activated by configuring the `DATA_REFRESH_INTERVAL` environment variable in Build Pulse, which specifies the time interval in seconds at which *glbuild* reruns to fetch the newly available build job data since its last run.

Each time Build Pulse integrates newly available build data, it notifies all the data subscriber components via the CBDT Message Broker. Finally, it exposes REST APIs for searching through the integrated data using criteria on various data fields.

Build Monitor is the data visualization application enabling real-time build process monitoring. For its implementation, we extended Grafana OSS[4] with a provisioned data source, dynamic SQL queries, and dashboard setups for real-time visualization of the build performance metrics defined in Section 8.3.1. Via the queries, these performance metrics are calculated and refreshed on the dashboard at a configurable time interval. The monitoring dashboard enables metrics aggregation at different time intervals (i.e., hourly, daily, weekly, monthly, and yearly) and data granularity levels (i.e., single versus multiple projects). Build Monitor also enables threshold-based alerts (e.g., via e-mail or Slack), allowing practitioners to take prompt action to address issues before they escalate.

Build Data Subscriber is the boilerplate Python application designed to subscribe to DT data updates. It includes minimal functions required to listen to data availability signals that Build Pulse emits via the CBDT Message Broker. This boilerplate application serves as a foundation for building contextual data subscriber components (with gray and hashed background in Fig 8.2) that require the build data stream to synchronize their states and automate actions.

CBDT Data Store is responsible for storing and providing access to the DT data. We use PostgreSQL, an open-source database system that efficiently stores large volumes of structured and unstructured data and offers high querying performance. While the current CBDT

[4] https://grafana.com/oss/grafana

framework version uses PostgreSQL, the framework can easily be customized (by implementing the aforementioned `AbstractStorage` class) to interact with other storage solutions, such as Logstash for execution logs.

CBDT Message Broker is the publish-subscribe software used to notify the data subscribers of the availability of new build data from BUILD PULSE. For simplicity, this broker is not explicitly shown in the CBDT architecture. We use RabbitMQ[5], a very popular message broker that offers persistence and high availability.

8.5 Leveraging Machine Learning to Build CBDTs

Creating DT models is essential for **real-time predictions** and **anomaly detection**. Based on the target performance metrics defined in Section 8.3.1, we propose creating ML models of three key build process aspects: failure, duration, and flakiness. Beyond the typically intended use case of these models, which is to assist developers in managing their schedules and proactively tackling failures (Aïdasso *et al.*, 2025f), the continuous monitoring of model outcomes and actual process states can help practitioners better understand the models' decisions while enabling real-time anomaly detection by identifying deviations from predicted (i.e., expected) states. Concretely, the idea is to have real-time monitoring of the predicted and actual states so that sudden spikes in actual failures, duration, or flakiness (compared to predictions) could be easily linked to recent process changes. Similarly, improvements resulting from process changes can be measured by a decline in these metrics.

To create the DT models, one can leverage existing techniques for build failure prediction, duration prediction, and flaky build detection reported by Aïdasso *et al.* (2025f). Algorithms used in state-of-the-art techniques mainly include tree-based models like XGBoost and Random Forest, which offer easy interpretability, as well as deep learning models like Long Short-Term Memory networks, which effectively capture temporal patterns in data. In addition, modeling build processes involves transforming collected data into features associated with code changes,

[5] https://www.rabbitmq.com

CI performance, infrastructure telemetry, developers and project-specific characteristics, as previously reported (Aïdasso *et al.*, 2025f). Here, LLMs can be valuable for processing textual data such as code and logs.

Challenges of leveraging ML in the context of CBDTs are two-fold: (1) creating several multi-scale models within the same framework, including project-specific models, and cross-project models that leverage data from other projects to improve predictions within a single project; and (2) continuously using the numerous real-time models while maintaining high availability and minimal latency, and updating them online (e.g., Ni & Li (2018)) to avoid concept drift.

8.6 Build Optimization with the CBDT

8.6.1 What-if Analysis Service

In the context of CBDTs, what-if analyses refer to exploring how different build process changes (i.e., scenarios) impact each build performance metric. Such analyses will enable practitioners to explore alternative scenarios and make informed decisions by identifying process changes that maximize benefits while minimizing negative impacts on other performance metrics. The what-if analysis service can be implemented using ML models as discussed in (Grafberger *et al.*, 2022). The idea is to use replicas of the DT models to evaluate the sensitivity of the build performance metrics to specific changes in the build configuration (e.g., enabling cache or updating the testing framework version). To achieve this, DT models should integrate process-related features, such as numerical representations of build configuration files (e.g., `.gitlab-ci.yml`), which can be obtained as embeddings from language models. Moreover, effective what-if analysis requires learning from large enough historical build data from various projects to reliably estimate how process modifications influence the different build performance metrics.

8.6.2 Improvement Services

[6] https://docs.gitlab.com/ee/api/instance_level_ci_variables.html

Various services can be implemented to infer improving changes and automate actions for the continuous improvement of the AT. For example, when the DT duration model predicts a long build time, build acceleration techniques like caching (Gallaba *et al.*, 2022a) can be applied. If a build failure is predicted, ML-based automated repair techniques (Lou *et al.*, 2019) can be used to infer and apply patch candidates via the AT's update API endpoints (e.g., Repository files API[6] and CI/CD variables API[6]). While efforts have been made toward the diagnosis of flaky builds (Aïdasso *et al.*, 2025a; Lampel *et al.*, 2021), techniques for automating the repair of such builds are yet to be developed to minimize the associated waste. Overall, the envisioned idea is that such recurring and automated improvement actions, executed in concert with the DT models' predictions and informed decision-making about process changes, will ensure continuous and global improvement of build processes.

8.7 Conclusion

This paper introduces a novel idea of developing CI Build Digital Twins (CBDTs) for holistic and continuous optimization of build processes. Essentially based on continuous data mining, the CBDT aims to provide various services, including real-time monitoring, anomaly detection, what-if analysis, and automated improvement of multiple build process performance metrics like failure rate, duration, and flakiness. To support such an idea, we developed the CBDT framework, which, in its current state, enables digital shadowing. This MVP framework is part of ongoing research work toward developing CBDTs with industrial partners who see great value in this DT-based approach for global build optimization.

[6] https://docs.gitlab.com/api/repository_files/#update-existing-file-in-repository

CHAPTER 9

GENERAL DISCUSSION

9.1 Preamble

Beyond the empirical findings discussed within each chapter, this chapter reflects on the practical tool contributions presented in Chapters 4 through 8. It discusses how the proposed tools chain together into an end-to-end automation pipeline to improve CI/CD reliability, the lessons learned from industrial deployment at TELUS, the generalizability of these tools beyond this specific industrial context, and the research roadmap toward fully autonomous self-healing pipelines for reliability issues.

9.2 From Detection to Repair: A Suite of Tools for Improving Pipeline Reliability

The tools developed in this thesis, namely SLID, FlakeRanker, FlaXifyer, and LogSift, form an integrated suite that can be deployed within CI/CD pipelines to systematically address intermittent job failures. Figure 9.1 illustrates how these tools chain together into a practical automation pipeline operating within the CBDT Framework. In parallel, the empirical findings on silent failures (Chapter 7) provide the groundwork for extending this automation chain to jobs that falsely report success. Each tool addresses a distinct stage of the automation chain, from detection through diagnosis to repair.

Detection of intermittent job failures with SLID models serves two purposes. First, it prevents wasteful reruns and saves developer time by immediately confirming or dismissing their suspicions. Second, and equally important, it acts as a first-level triage that filters intermittent failures from regular ones (which follow the standard developer debugging workflow) before multi-class categorization. This filtering is essential: without it, a multi-class classifier would face severe class imbalance, as regular failures vastly outnumber any specific intermittent failure category. By routing only detected intermittent failures to FlaXifyer, the pipeline maintains balanced prediction performance across categories.

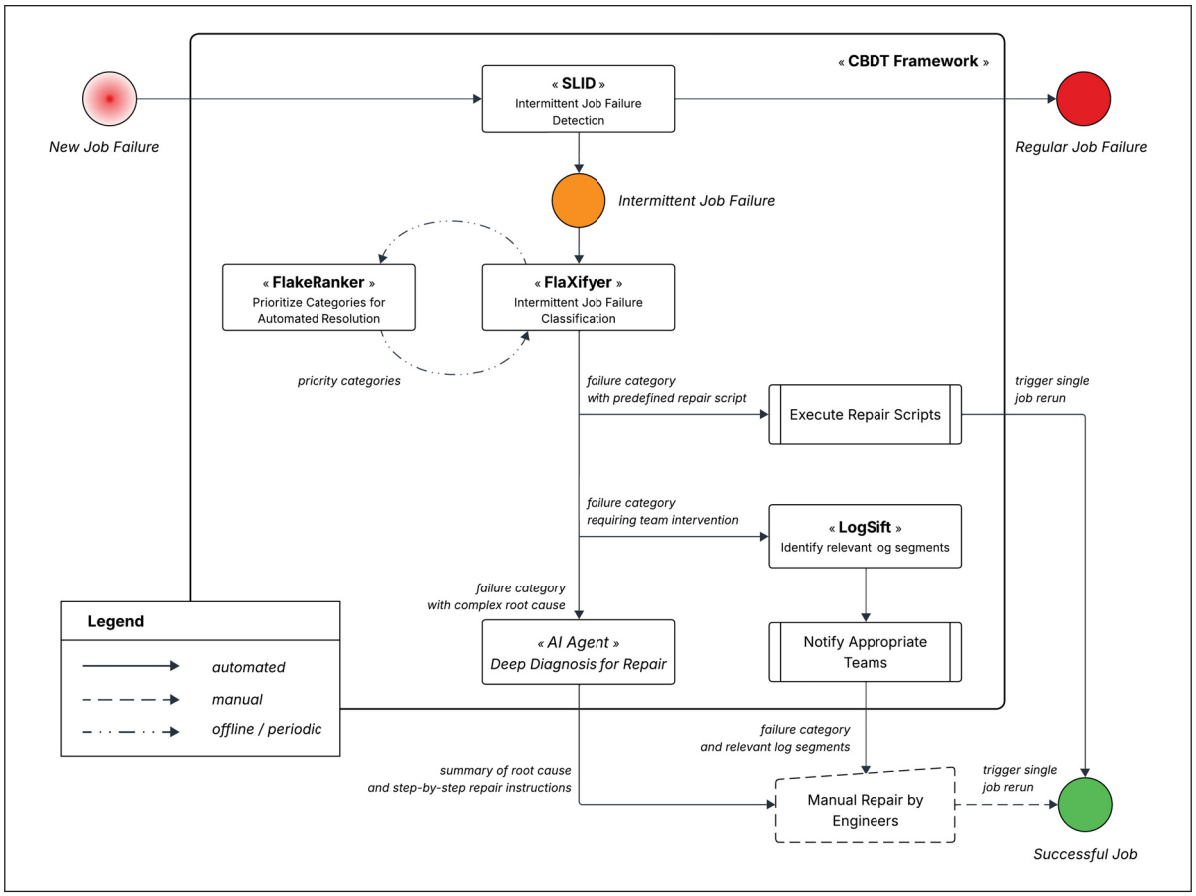


Figure 9.1 Integrated pipeline for the automated detection, diagnosis, and repair of intermittent job failures, operating within the CBDT Framework. The pipeline orchestrates SLID model for detection, FlakeRanker for prioritization, FlaXifyer model for classification, and LogSift for interpretability, supporting faster diagnosis. Repair paths vary by category complexity: automated scripts for well-understood issues, team notification with relevant log segments for issues requiring human expertise, and AI-assisted diagnosis for categories with complex root causes

FlakeRanker complements detection by providing the diagnostic taxonomy and prioritization framework that informs downstream automation. Beyond its initial use in establishing the 46 failure categories and identifying the 14 priority ones (Chapter 5), FlakeRanker supports continuous monitoring in production. Organizations can periodically run it on accumulated failure data to track whether priority categories are being effectively addressed, detect shifts in category prevalence, and surface emerging issues before they become critical. This monitoring ensures that automation efforts remain aligned with the most impactful reliability challenges.

Once a SLID model detects an intermittent job failure, FLAXIFYER determines its specific category, and the path to repair depends on the nature of that category. For categories with known or deterministic repair patterns, lightweight automation scripts can be triggered directly. For instance, a `misconfigured_env_variable` failure can trigger a script that searches parent projects for similarly named variables via API[1] and suggests the most recently updated variable matches. Similarly, a `job_execution_timeout` failure can trigger a default script that increases the configured limit via an API call[2], though as discussed in Chapter 5, deeper diagnosis is often warranted for this category. Such repair automation scripts can be organized in a category-to-script mapping structure and executed automatically whenever FLAXIFYER predicts the corresponding category for a detected intermittent job failure.

For categories with more complex underlying causes, simple scripted repairs fall short, and Agentic AI (Wang *et al.*, 2024b) offers a promising path forward. Categories such as `transient_git_error` or `container_registry_server_error` can stem from underlying reasons that vary across occurrences. For these, we developed an AI agent using a Large Language Model (LLM) as a reasoning backbone to accelerate diagnosis and repair. Given an intermittent job failure log and its predicted category, the agent builds targeted search queries from the log content and leverages tools to retrieve root cause and repair information from GitLab Issues, Stack Overflow, and other internal knowledge bases where developers report and discuss pipeline reliability issues. It then synthesizes this information into actionable step-by-step summaries for developers. This agent is currently undergoing systematic evaluation with practitioners at TELUS and shows promising results in supporting the repair of complex intermittent failures.

While the current tools focus on intermittent job failures (i.e., false alarms), the complementary challenge of silent failures (i.e., false successes) remains an open frontier. Chapter 7 identified key factors associated with such failures, particularly in testing and static analysis jobs that produce critical artifacts, and established a taxonomy of 11 silent failure themes. These findings,

[1] https://docs.gitlab.com/api/project_level_variables/

[2] https://docs.gitlab.com/api/runners/#update-runners-details

combined with the statistical explanatory model developed in that study, lay the groundwork for future detection and diagnosis tools targeting silent failures. We envision extending the automation chain with a silent failure detector that flags suspicious job successes before they propagate defects into production, followed by category-specific verification scripts that validate expected artifacts or environmental changes.

9.3 Lessons Learned from Industrial Deployment

Ongoing work for deploying the proposed solutions at TELUS revealed practical challenges for organizations, from data access to model deployment and organizational adoption.

Data Access and Availability. Our detection and classification approaches rely primarily on job execution logs, which are generally available across CI services. Nevertheless, retention policies vary. GitHub Actions, for instance, retains logs for only up to 90 days in public repositories and 400 days in private ones[3]. Such limited retention hinders longitudinal analysis, including the RFM-based prioritization that FlakeRanker enables (Chapter 5), which benefits from historical data spanning months or years to accurately capture recency, frequency, and cost trends. For model training, however, our few-shot approaches require only 12 examples per class: i.e., 24 log instances for binary detection (Chapter 4) and up to 156 log instances for 13-category classification (Chapter 6). These represent a modest requirement for any organization where intermittent job failures have become problematic enough to warrant attention.

We also identified additional data sources, such as infrastructure telemetry and runner logs, that could deepen diagnostic capabilities beyond what job logs alone provide. For instance, a failure attributed to a crashed pod does not reveal why the pod failed: information that could be located in runner logs. However, these sources typically require elevated access privileges and contain more sensitive information, limiting their practical use in organizations. CI providers are well-positioned to leverage such data internally to deliver more reliable services to their users.

[3] https://docs.github.com/en/organizations/managing-organization-settings/configuring-the-retention-period-for-github-actions-artifacts-and-logs-in-your-organization

Sensitive Data and LLM Application. Job logs themselves often contain sensitive information such as developer emails, internal paths, and occasionally tokens or credentials. Hence, although LLMs have been widely adopted for software engineering tasks (Hou *et al.*, 2024), the presence of such data in logs precludes the use of public LLM APIs, requiring instead private organizational accounts or in-house instances (used at TELUS). Even with internal deployment, sensitive data exposure remains a concern. Future work should develop automated log sanitization techniques to enable safer exploration of LLM-based approaches for improving pipeline reliability.

Model Deployment and Efficiency. Deploying detection and classification models in production revealed challenges related to model size. Indeed, the small pre-trained language models resulted in container images exceeding 30GB per model, which becomes impractical when maintaining project-specific classifiers at scale. We addressed this through quantization and conversion to the Open Neural Network Exchange (ONNX) format[4], reducing model sizes to about 2-4GB with faster inference. As discussed in Chapter 4, a promising direction regarding detection models is to train larger models that generalize across many projects, replacing the current one-model-per-project configuration and simplifying deployment.

9.4 Generalizability to Other Software Organizations

The approaches and tools developed in this thesis, including SLID, FlakeRanker, FlaXifyer, and LogSift, are designed to be reusable across organizations. The methodologies for detecting intermittent failures with few-shot learning (Chapter 4), systematically categorizing and prioritizing them (Chapter 5), and predicting categories with interpretable outputs (Chapter 6) transfer directly to any CI/CD environment with accessible job logs. Moreover, similarities between the failure categories observed at TELUS and those reported at Ubisoft (Olewicki *et al.*, 2022) and Mozilla (Lampel *et al.*, 2021) suggest that many categories generalize across industrial contexts. Notably, infrastructure problems, network instabilities, timeout issues, and non-deterministic tests are common across organizations operating complex CI/CD systems (Olewicki *et al.*, 2022; Lampel *et al.*, 2021; Weeraddana *et al.*, 2024a; Moriconi *et al.*, 2022).

[4] https://onnx.ai/

Nevertheless, the specific failure patterns within each category and their associated repair strategies are inherently context-specific, depending on the CI service, toolchain, organizational policies, and infrastructure in use. The 46 categories identified at TELUS serve as a reference taxonomy, but organizations adopting our proposed tools should expect to discover patterns unique to their environment. The tools and approaches are precisely designed to enable systematic discovery and prioritization of context-specific reliability issues.

9.5 Toward Fully Autonomous Resolution of Pipeline Reliability Issues

The CBDT Framework introduced in Chapter 8 provides the architectural foundation for integrating the tools developed in this thesis into a unified, autonomous system. This section outlines how the current contributions fit within the framework and the research directions required to achieve fully autonomous self-healing pipelines.

Integrating Detection and Diagnosis Models. Models created with SLID and FlaXifyer can be deployed as prediction services within the CBDT's Digital Twin Services layer. Upon each job failure event, the detection model classifies the failure as intermittent or regular; if intermittent, the categorization model predicts its failure category. LogSift can operate alongside these models as an interpretability service, extracting the most influential log segments for each prediction. The combination of predicted category and relevant log statements can then be routed to appropriate teams via a configurable mapping table (category to notification channel, such as email or Slack), enabling automated triage.

Integrating Repair Services. The category-to-script mapping and the LLM-based repair agent described earlier can be deployed as Improvement Services within the CBDT. When a repair script is available for a predicted category, it is executed automatically. Upon successful execution, the framework can trigger a single job rerun via the CI provider's API[5] to verify resolution. For categories that require the AI agent, the framework routes corresponding failures to the agent, which retrieves and synthesizes repair guidance for developer review.

[5] https://docs.gitlab.com/api/jobs/#retry-a-job

This architecture enables a spectrum of automation levels, from fully autonomous repair for well-understood categories to AI-assisted diagnosis for complex ones.

Future Research Opportunities. Several research challenges remain on the path to fully autonomous self-healing pipelines:

Incremental learning for model maintenance. Currently, our classification models require periodic manual retraining as failure patterns evolve. Online and continual learning techniques (Wang *et al.*, 2024c; Gama, Žliobaitė, Bifet, Pechenizkiy & Bouchachia, 2014) could enable models to adapt to new failure patterns without forgetting previously learned ones, maintaining prediction performance as CI/CD environments change.

Digital twin modeling of CI/CD processes. The current CBDT implementation focuses on data integration, predictive modeling, and monitoring of pipeline performance metrics. Extending it to a full digital twin requires a formal representation of the CI/CD process itself. Ongoing work explores meta-modeling CI/CD pipelines using process notations such as BPMN[6] to create a structural and behavioral counterpart of the actual pipeline. Such a representation will help capture both its structure and performance evolution over time, a foundation for what-if analysis.

What-if analysis for process changes. A compelling application of digital twins is the ability to simulate process changes (such as introducing a new tool, modifying caching strategies, or updating runner configurations) and evaluate their impact on reliability and other performance metrics before deployment. Effective what-if analysis can be enabled by a foundation model (Bommasani *et al.*, 2022) for CI/CD pipelines, trained on an extremely diverse set of pipeline configurations across a wide range of project contexts, to accurately predict the effects of changes it has not directly observed. Developing such a model represents a significant research undertaking, but it could transform how organizations approach CI/CD process improvement.

[6] https://www.omg.org/spec/BPMN

CONCLUSION

Reliable software delivery requires dependable CI/CD pipelines, yet pipeline anomalies remain frequent and a significant source of waste for software organizations. This thesis presents a comprehensive investigation into the development of practical, intelligent automation solutions to address these reliability issues. Through a combination of empirical studies, efficient machine learning approaches, and practical tool and framework design, this work demonstrates that AI-powered automation offers a viable path toward sustainable CI/CD reliability.

First, we introduced SLID, an efficient detection approach for intermittent job failures based on few-shot fine-tuned sentence transformers. By requiring only 12 manually labeled log pairs per class, SLID significantly outperforms the state-of-the-art, which relies on large, often mislabeled datasets. SLID also demonstrates strong robustness across diverse project contexts, underscoring the importance of data quality over quantity. These results establish few-shot learning as a practical paradigm for log-based classification under limited labeled data.

Second, we advanced the diagnosis of intermittent job failures by establishing the first comprehensive taxonomy of 46 failure categories and introducing FlakeRanker, a tool that combines RFM analysis with clustering to identify the 14 most wasteful categories. This prioritization strategy enables organizations to focus automated repair efforts where they yield the greatest return, moving beyond traditional frequency-based approaches.

Third, building on these priority categories, we developed FlaXifyer, a few-shot multi-class classifier that predicts the failure category of a detected intermittent job failure using only 12 labeled logs per category. We also introduced LogSift, an interpretability technique that highlights the most influential log segments for each prediction, accelerating triage and diagnosis across teams. Together, FlaXifyer and LogSift complete the detection-to-diagnosis automation chain for intermittent job failures.

Fourth, we conducted an empirical study of silent failures, examining jobs that developers rerun despite reporting success. This study identified key contributing factors, particularly jobs involving testing or static code analysis that rely on or produce critical artifacts, and established a taxonomy of 11 themes of silent failures with their practical mitigation strategies. These findings lay the groundwork for future automated detection and diagnosis of silent failures.

Finally, we introduced the CBDT Framework, a novel framework inspired by digital twins that provides the foundation for orchestrating automated detection, diagnosis, and repair services in production CI/CD environments. The framework enables continuous data integration and real-time monitoring, providing the architectural backbone for systematic and holistic pipeline optimization beyond reliability.

Overall, these contributions advance the SOTA in automated detection and diagnosis of pipeline reliability issues, and demonstrate that intelligent automation, grounded in efficient machine learning, can improve pipeline reliability, reduce the burden on developers, and lower the costs associated with delivery delays and software defects.

APPENDIX I

INTERVIEW PROTOCOL

This appendix presents the interview guide used for semi-structured interviews with TELUS practitioners at the outset of this research. The interviews aimed to understand CI/CD processes in the company, identify common pipeline failures, and assess opportunities for reducing waste.

1. Introduction Script

Participants are informed that the interview aims to understand the structure and specificity of DevOps processes at TELUS, and to identify opportunities for improving CI/CD pipeline efficiency and reliability by addressing failures whose resolution involves significant challenges and waste. Participation is voluntary and would not affect employment status. Participants are informed that the interview will last approximately one hour, that responses will be kept confidential and de-identified, and that they can decline to answer any question or stop the interview at any time. Permission to audio-record is requested before proceeding.

2. Interview Themes and Questions

The interview is organized around four themes, summarized below.

2.1 Theme 1: Demographics

This section gathers participant background information for characterizing the sample.

1. What team are you part of at TELUS? What are its main missions and products?
2. What is your current role in this team?
3. How many years of experience do you have: (a) in the technology industry, (b) with CI/CD technologies, and (c) in your current position?
4. What are your day-to-day tasks in your current role regarding CI/CD pipelines?

2.2 Theme 2: CI/CD Pipeline Architecture

This section aims to understand the pipeline architecture and tooling at TELUS.

5. What is the typical workflow of a submitted change from coding to production? What are the different stages of your CI/CD pipelines?
6. To the extent of your knowledge, what are the primary tools used at the different stages of the CI/CD pipeline (e.g., Python, Pytest, GitLab CI, Docker, runners)?
7. Do you use any specific in-house server or tool for running your pipeline jobs?

2.3 Theme 3: CI/CD Pipeline Failures

This section gathers perceptions of pipeline failure frequency and characteristics.

8. How many pipelines are triggered in your team on average per day?
   Options: <50, 50–100, 100–300, 300–500, >500
9. What is the proportion of pipeline failures per day?
   Options: <5%, 5–10%, 10–30%, 30–50%, >50%
10. Do you often observe failures due to specific issues that are not necessarily code-related (e.g., network outages, out of memory errors)? (*open question*)
11. What is your estimation of the relative daily proportion of such build failures?
    Options: <5%, 5–10%, 10–30%, 30–50%, >50%

2.4 Theme 4: Hard-to-Resolve Pipeline Failures

This section focuses on failures requiring manual, long, or repetitive work to resolve, to establish a taxonomy of common failures, their symptoms, and fix patterns.

12. What are the most common (top 5) failures you face that require hard, manual, and repetitive work to resolve? For each, describe the symptoms, current diagnosis and fix procedures.
13. How would you qualify or quantify the impact of these failures (i.e., the waste)?

3. Conclusion

Participants are invited to share any additional comments not covered by the questions, and thanked for their time and the information shared.

APPENDIX II

CHARACTERISTICS OF STUDIED PROJECTS

Table-A II-1 Characteristics of studied industrial projects at TELUS. Project identifiers are anonymized for confidentiality. *FR* stands for Failure Ratio (failed jobs / total jobs).

#	Purpose	Months	Language	Commits	Jobs	FR (%)
1	Backend Data API	46	Python	805	2,612	23.70
2	Network Lifecycle Manager	46	Python	1,784	3,678	49.48
3	ID and Access Management	50	Python	450	1,102	30.76
4	Telemetry Exporter	77	Go	381	714	42.58
5	BI Server	35	JavaScript	139	154	8.44
6	BI Backend	39	Python	629	2,403	18.06
7	BI Backend	40	Python	613	2,178	15.61
8	API Gateway	48	Python	463	2,247	7.97
9	BI Web Portal	37	HTML	658	3,383	46.56
10	Network Slicing	14	Python	535	2,518	9.69
11	Network Slicing	17	Python	663	2,105	16.48
12	Network Slicing	17	Python	419	1,924	8.37
13	Network Slicing	14	Python	433	1,083	27.79
14	Cache Management	31	Python	224	471	24.42
15	Network Planning	34	Python	406	954	44.97
16	Network Management Portal	33	TypeScript	136	457	13.35
17	Template Maintenance	38	Python	228	654	30.73
18	Network Optimization	39	Python	370	1,160	27.41
19	Network Forecasting	41	Python	572	1,239	48.43
20	Cache Management	41	Python	526	1,716	23.31
21	SDN API Gateway	36	Python	88	504	5.16
22	SDN Controller	36	Python	231	887	13.30
23	SDN Controller	35	Python	287	1,071	16.25
24	SDN Controller	36	Python	192	753	9.30
25	SDN Web Portal	36	TypeScript	457	2,034	20.94
26	Polling Engine	16	Python	212	778	12.85
27	Database Management	19	Python	326	1,211	7.27

Continuation of Table II-1

#	Purpose	#Months	Language	Commits	Jobs	FR (%)
28	Query Service	22	Python	148	492	14.63
29	Backend Data API	27	Python	486	1,465	21.30
30	API Gateway	26	Python	136	552	13.41
31	Web Portal	26	TypeScript	782	1,596	36.15
32	SDN Controller	33	Python	845	3,097	12.11
33	SDN Controller	33	Python	916	3,216	12.47
34	Network Topology Engine	29	Python	176	593	18.72
35	SDN Controller	29	Python	432	1,401	19.20
36	VPN Controller	16	Python	554	1,938	18.42
37	VPN Service	36	Python	154	264	77.27
38	Data Ingestion	26	Python	169	608	19.74
39	VPN Controller	30	Python	549	2,070	19.37
40	VPN Controller	23	Python	844	3,286	8.89
41	CLM Service	23	Python	1,013	4,109	7.74
42	CLM Service	23	Python	1,087	3,269	12.14
43	BI Backend	25	Python	558	1,571	24.32
44	BI Web Portal	24	TypeScript	943	1,725	9.39
45	API Proxy	8	Shell	1,119	2,302	30.93
46	Backend Data API	13	Shell	190	546	34.98
47	Backend Data API	13	Shell	101	242	37.60
48	ID and Access Management	13	Shell	53	236	28.39
49	API Proxy	13	Shell	473	737	44.51
50	IaC Configuration	3	HCL	59	169	26.63
51	Web Portal	4	JavaScript	197	677	11.52
52	API Documentation	14	Shell	3,174	4,098	56.81
53	E2E Testing	33	Gherkin	1,272	2,515	16.10
54	Config. Management API	9	Shell	48	51	0.00
55	Config. Management Portal	13	Shell	48	53	5.66
56	Config. Management	12	Shell	69	78	2.56
57	Config. Management	34	Shell	49	50	2.00
58	Log Management	13	Shell	136	274	24.09
59	Config. Management	47	Python	387	1,290	8.76

Continuation of Table II-1

#	Purpose	#Months	Language	Commits	Jobs	FR (%)
60	Config. Generator	50	Jinja	1,158	4,139	13.46
61	SDN Controller	6	Python	141	396	20.20
62	SDN Controller	56	Python	1,397	5,607	18.73
63	Web Portal	59	TypeScript	2,012	7,325	21.80
64	Backend Data API	59	Python	1,084	3,747	16.20
65	Message Queue Service	16	Python	1,696	8,833	28.66
66	Web Portal	15	JavaScript	383	1,569	17.97
67	API Engine	13	Dockerfile	54	103	23.30
68	Web Portal	19	TypeScript	631	1,297	17.12
69	Backend API	21	Python	378	1,233	18.09
70	Backend API	20	Python	430	1,306	16.00
71	Backend API	21	Python	496	1,694	18.54
72	Backend API	11	Python	65	180	17.22
73	Backend API	29	Python	953	3,089	16.09
74	SDN Controller	30	Python	365	1,208	15.32
75	SDN Controller	30	Python	570	1,952	15.27
76	SDN Controller	33	Python	674	2,347	14.23
77	SDN Controller	30	Python	793	2,790	14.84
78	SDN Controller	34	Python	362	1,351	16.80
79	SDN Controller	32	Python	801	2,657	13.59
80	API Engine	24	Dockerfile	58	181	20.44
81	Web Portal	35	TSX	1,367	4,823	17.56
Total (81 projects, 6.5 years, 13 languages)				**45,262**	**142,387**	**21.06%**

Table-A II-2 Characteristics of studied open-source GitLab projects. FR = Failure Ratio; FFR = Flaky Failure Ratio.

#	Project	Purpose	Stars	Lang.	Commits	Jobs	FR %	FFR %
1	inkscape/inkscape	Vector Graphics Editor	3,408	C++	15,300	201,206	12.75	5.73
2	CalcProgrammer1/OpenRGB	RGB Lighting Control	2,975	C++	3,900	81,316	6.36	34.25
3	gitlab-org/gitlab-runner	CI/CD Executor	2,379	Go	5,700	422,720	6.50	17.54
4	F-Droid/Client	Android App Repository	2,294	Java	2,300	11,962	12.98	15.47
5	veloren/veloren	Multiplayer Game	2,169	Rust	5,100	54,484	21.58	21.96
6	baserow/baserow	No-Code Database	1,999	Python	14,600	299,900	4.56	12.46
7	AuroraOSS/AuroraStore	Alternative App Store	1,834	Kotlin	1,400	3,313	3.23	2.47
8	graphviz/graphviz	Graph Visualization	1,306	C	6,700	414,559	3.39	8.49
9	NickBusey/HomelabOS	Self-Hosted Services	1,198	Jinja	2,300	4,637	4.66	0.47
10	wireshark/wireshark	Network Analyzer	1,128	C	14,600	200,038	9.20	31.01
Total (10 projects, 8 languages)					**71,900**	**1,694,135**	**6.97**	**16.57**

APPENDIX III

REPLICATION OF INTERMITTENT FAILURE DETECTION ON ADDITIONAL OPEN-SOURCE PROJECTS

This appendix presents results from the replication of the few-shot learning-based detection approach (RO_1) on open-source projects. The main paper reports results for industrial projects and Veloren. This appendix extends the evaluation to four additional OSS projects: Inkscape, OpenRGB, GitLab Runner, and F-Droid Client.

1. Mislabeling Error Rates

Table III-1 presents the error rates of automated labeling compared to manual labeling across the five open-source projects. As a reminder, automated labeling relies on the rerun heuristic (i.e., a failure is labeled as intermittent if it succeeds upon rerun on the same commit), while manual labeling involves inspection of failure logs to determine the true root cause.

Table-A III-1 Mislabeling error rates of automated labeling compared to manual labeling across open-source projects

Project	Automated		Manual		Error
	Intermittent	Regular	Intermittent	Regular	Rate (E)
`inkscape`	23	356	77	302	14.25%
`openrgb`	122	235	156	201	9.52%
`gitlab-runner`	66	314	156	224	23.68%
`fdroidclient`	50	255	161	144	36.39%
`veloren`	81	285	141	225	16.39%

The error rates vary considerably across projects, ranging from 9.52% (OpenRGB) to 36.39% (F-Droid Client). Notably, Veloren (the OSS project presented in the main paper) exhibits a relatively moderate error rate of 16.39%, which is representative of the median across the five projects. F-Droid Client shows the highest mislabeling rate, due to its high number of intermittent failures that were incorrectly labeled as regular by the automated heuristic (111 out of 161 intermittent failures were missed). These findings reinforce observations from industrial projects regarding the limitations of automated labeling based on reruns. First, the heuristic systematically

underestimates the prevalence of intermittent failures by missing instances that were not rerun or did not succeed upon rerun within the observation window. Second, and more critically, these missed instances are mislabeled as regular failures, introducing noise into training data that confuses models and degrades detection performance. Third, the heuristic depends entirely on rerun practices that are neither guaranteed nor desirable. In fact, organizations actively seek to reduce wasteful reruns, yet doing so would undermine the very labeling mechanism that sustains detector accuracy. This circular dependency makes rerun-based labeling fundamentally unsuitable as a long-term solution.

2. Few-Shot Learning Detection Performance

Figure III-1 illustrates the evolution of the average F1-score as the number of labeled examples per class increases from 1 to 15. All OSS projects exhibit rapid performance gains in the early shots (1–5), followed by diminishing returns as more examples are added. This pattern confirms that the few-shot learning approach achieves practical performance with minimal labeling effort.

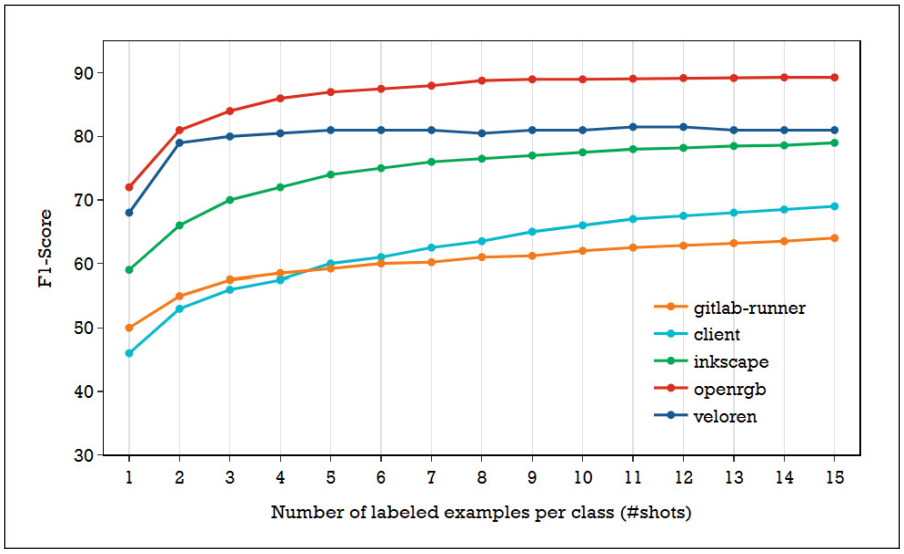


Figure-A III-1 F1-score evolution as a function of the number of labeled examples per class across five open-source projects

Table III-2 presents detailed performance metrics at $N = 12$ (the recommended configuration from the main paper) and $N = 15$ shots per class. Performance is reported as mean with standard deviation subscript across 100 Monte Carlo cross-validation iterations.

Table-A III-2 Performance (mean_{std}) of FSL models on open-source projects using $N = 12$ and $N = 15$ examples per class

Project	N **= 12**					N **= 15**				
	F1	Intermit.		Regular		F1	Intermit.		Regular	
		Pre	Rec	Pre	Rec		Pre	Rec	Pre	Rec
`inkscape`	78_{5}	46_{7}	81_{10}	94_{3}	75_{8}	79_{5}	48_{7}	83_{10}	95_{3}	76_{8}
`openrgb`	89_{3}	91_{5}	85_{6}	89_{4}	93_{5}	90_{3}	92_{5}	86_{6}	90_{4}	94_{5}
`gitlab-runner`	63_{5}	55_{6}	61_{10}	70_{5}	64_{10}	64_{5}	56_{6}	62_{10}	71_{5}	65_{10}
`fdroidclient`	67_{6}	73_{7}	66_{13}	67_{7}	70_{11}	69_{6}	74_{7}	68_{13}	68_{7}	71_{11}
`veloren`	82_{3}	85_{5}	74_{8}	84_{4}	90_{3}	90_{3}	91_{5}	83_{8}	90_{4}	94_{3}
Median	78_{4}	73_{6}	74_{9}	84_{5}	75_{7}	79_{4}	74_{6}	83_{10}	90_{4}	76_{8}
Mean	76_{4}	70_{6}	73_{9}	81_{5}	78_{7}	78_{4}	72_{6}	76_{9}	83_{5}	80_{7}

3. Discussion

3.1 Generalizability of Findings

The results demonstrate that the few-shot learning approach generalizes well to open-source projects, achieving F1-scores between 63% and 89% at $N = 12$ shots. Two projects, OpenRGB (89%) and Veloren (82%), achieve performance comparable to the best industrial projects at TELUS. Inkscape also performs well, with an F1-score of 78%, demonstrating that the approach transfers effectively across projects in different programming languages and domains (e.g., graphics software vs. telecommunications).

3.2 Challenging Cases

GitLab Runner exhibits the lowest performance among the five projects (63% F1-score at $N = 12$), despite having a moderate mislabeling error rate (23.68%). This outcome can be attributed to several characteristics of the project, identified during manual analysis:

- **Log heterogeneity.** GitLab Runner logs are exceptionally large and diverse, containing output from multiple CI executors (Docker, Kubernetes, Shell, VirtualBox) with varying formats and verbosity levels.
- **Misleading failure patterns.** Many failures in GitLab Runner contain generic error messages that appear in both intermittent and regular failures, making discrimination difficult. For instance, timeout messages may indicate either infrastructure issues (intermittent) or genuine test failures (regular) under certain scenarios.
- **Complex failure chains.** Failures often result from cascading issues across multiple components, producing logs where downstream effects obscure the root cause.

Similar patterns have also been observed in the F-Droid Client project, which shows moderate performance (67% F1-score).

3.3 Implications

These findings suggest that while the few-shot learning approach is broadly applicable, performance varies with project characteristics. Projects with well-structured logs and distinctive failure patterns (e.g., OpenRGB, Veloren) benefit most from the approach. Projects with highly heterogeneous logs or ambiguous failure signatures (e.g., GitLab Runner) may require additional domain-specific preprocessing or larger training sets to achieve comparable performance.

The median F1-score of 78% across open-source projects, combined with the 70-88% range observed at TELUS, confirms that the approach provides practical detection capability across diverse CI/CD environments with minimal labeling effort (12 examples per class).

APPENDIX IV

SUPPLEMENTARY MATERIAL AND REPLICATION OF SILENT FAILURE ANALYSIS ON OPEN-SOURCE PROJECTS

This appendix presents supplementary material for the silent failure analysis (Chapter 7), including feature engineering procedures, collinearity analysis, and the replication study on open-source projects.

1. Feature Engineering of Categorical Variables

The mixed-effects logistic regression model includes three categorical variables derived from job metadata: job type, branch type, and programming language. This section describes the feature engineering process for each variable.

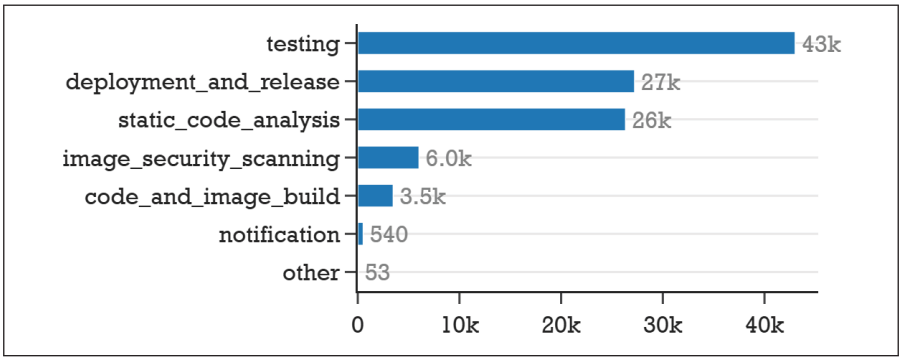


Figure-A IV-1 Distribution of job types

1.1 Job Type

Job types were derived using iterative open coding of job names. The process began with manual review of the first 100 job names to identify initial categories such as deployment and release (e.g., `infra_deploy`, `trigger_build_deploy`), testing (e.g., `job-unit_test`, `e2e-run`), and static code analysis (e.g., `code_verification`, `lint`). From this initial coding, a keyword-based script was developed to assign job types based on keyword presence in job names. For instance, `deploy`, `release` map to deployment and release; `test`, `e2e` map to

testing; and `valid`, `check`, `lint`, `quality` map to static code analysis. The script applies keyword matching in fixed priority order to handle overlapping terms (e.g., `e2e-build` is assigned to code and image build). When the type was ambiguous, the job specification in the pipeline configuration file was manually inspected. The process was repeated iteratively with batches of 100 unassigned job names until all jobs were categorized. Seven job types were identified; their distribution is shown in Figure IV-1.

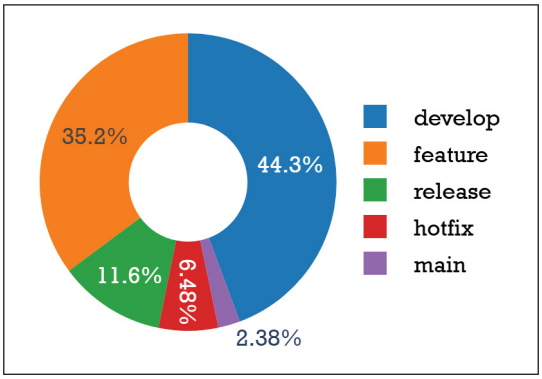


Figure-A IV-2 Distribution of branch types

1.2 Branch Type

Branch types were derived from the `ref` metadata field using the GitFlow naming convention (Atlassian, 2024) as the initial classification scheme. This convention defines five primary branch types: main, develop, feature, release, and hotfix. To refine keyword associations, batches of 100 unassigned branch names were iteratively reviewed using closed card sorting. The resulting keyword mappings are: `main`, `master` → *main*; `develop` → *develop*; `deploy`, `preprod`, `qa`, `prod` → *release*; `bug`, `fix`, `issue`, `patch` → *hotfix*; and remaining keywords (e.g., `feature`, `implement`, `add`, `update`) → *feature*. The distribution of branch types is shown in Figure IV-2.

1.3 Programming Language

For each project, GitLab provides a JSON dictionary listing programming languages and their usage ratios. The language with the highest ratio was assigned as the project's primary

programming language. For example, a project with `{Python: 94.14%, C: 28.9%, JavaScript: 0.41%}` was assigned Python as its primary language.

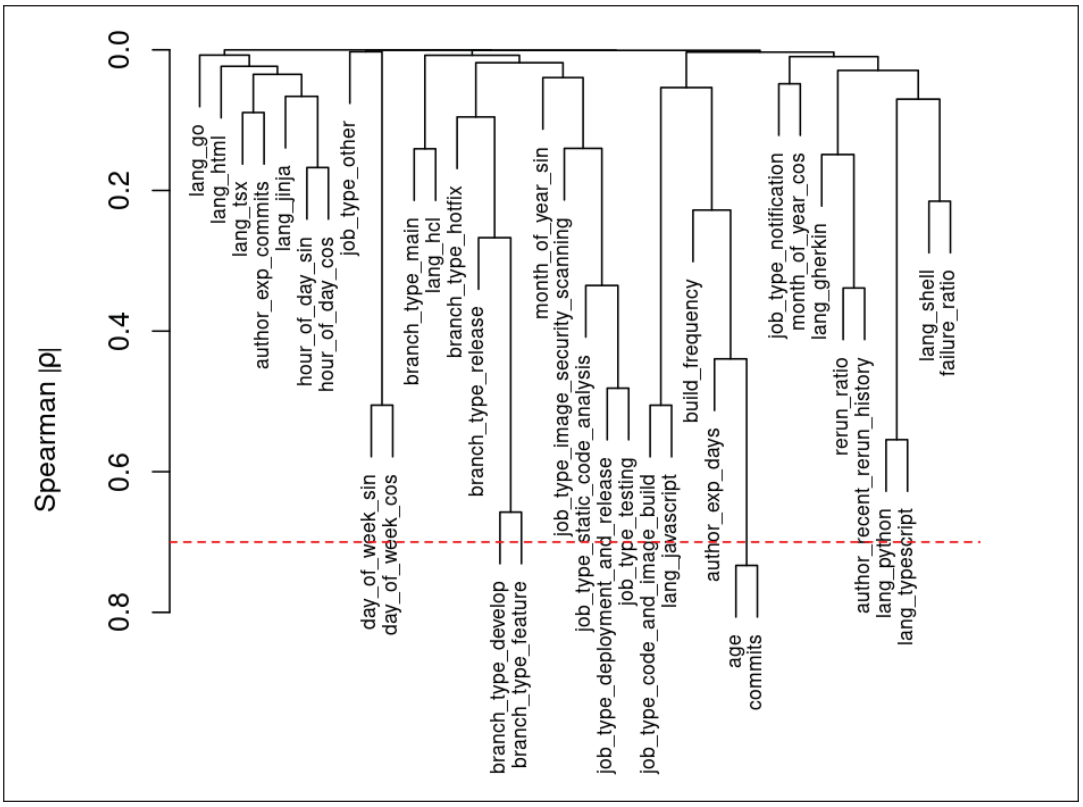


Figure-A IV-3 Hierarchical clustering of variables based on Spearman's rank correlation $|\rho|$

2. Collinearity Analysis

Before fitting the mixed-effects model, collinearity among independent variables was assessed using Spearman's rank correlation. Figure IV-3 shows the hierarchical clustering of the 36 initial numerical independent variables. Using a correlation threshold of $|\rho| = 0.7$ (indicated by the red dashed line), one pair of highly correlated variables was identified: `age` and `commits`. To mitigate this collinearity, `age` was removed because `commits` captures both project maturity and complexity. In fact, a recent project may contain more commits than older projects, indicating higher change complexity.

3. Model Evaluation Metrics

The mixed-effects logistic regression model was evaluated using five metrics assessing discrimination, calibration, and explanatory power.

ROC AUC. The Area Under the Receiver Operating Characteristic Curve (Hanley & McNeil, 1982) measures the model's ability to discriminate between positive and negative classes. Values range from 0 (worst) to 1 (perfect), with 0.5 representing random guessing.

AUC-PR. The Area Under the Precision-Recall Curve (Saito & Rehmsmeier, 2015) evaluates the trade-off between precision and recall, particularly useful for imbalanced datasets. The baseline equals the prevalence of the positive class. In the TELUS study, the baseline is therefore $11{,}808/(11{,}808+94{,}870) = 0.11$. Values significantly above this baseline indicate performance better than random.

Brier Score. The Brier Score (Brier, 1950) quantifies the mean squared error of predicted probabilities. Values range from 0 to 1, with lower scores indicating well-calibrated predictions.

Marginal R^2. The marginal R^2 (R^2_m) (Nakagawa & Schielzeth, 2013) represents the proportion of variance explained by fixed effects alone. Values range from 0 to 1, with values between 0.2 and 0.4 indicating an excellent fit (McFadden, 1978).

Conditional R^2. The conditional R^2 (R^2_c) represents the proportion of variance explained by both fixed and random effects. A large difference between R^2_c and R^2_m indicates that random effects (i.e., project-level variation) play a substantial role in explaining outcome variance.

4. Replication on Open-Source Projects

To assess the generalizability of findings beyond the industrial context, the study was replicated on popular open-source GitLab projects. This section presents the replication methodology, dataset, and results for RQ_1 (prevalence and impact) and RQ_2 (influential factors).

4.1 Studied Projects

Table II-2 lists the open-source projects included in the replication study. Due to the larger scale of these projects compared to TELUS microservices and computing resource constraints, only jobs executed within the last three months of the collected data were included. Projects without any reruns of successful jobs in this period (F-Droid/Client and NickBusey/HomelabOS) were excluded, resulting in 198,146 jobs across eight projects covering six programming languages.

Table-A IV-1 Statistics of job reruns

Metric	Industrial	Open-Source
# Projects including rerun jobs	81 (100%)	8 (80%)
# Rerun jobs	23,109 (16.23%)	15,749 (7.95%)
# Rerun sequences	13,380	10,983
# Sequence of 1 rerun	9,317 (69.63%)	8,589 (78.20%)
# Sequence of 2 reruns	2,263 (16.91%)	1,438 (13.09%)
# Sequence of 3 reruns	865 (6.46%)	519 (4.73%)
# Sequence of 4 reruns	379 (2.83%)	195 (1.78%)
# Sequence of 5 reruns	186 (1.39%)	55 (0.50%)
# Sequence of $\geq$ 6 reruns	370 (2.76%)	187 (1.70%)
Avg # of jobs per sequence	$\approx$ 2	$\approx$ 2

4.2 Prevalence and Impact of Rerunning Successful Jobs (RQ_1)

Table IV-1 compares job rerun statistics between the industrial (IND) and open-source (OSS) contexts, including the distribution of rerun sequence lengths. Figure IV-4 shows the distribution of status transitions between rerun-initiating jobs and their immediate reruns. Figure IV-5 shows the distribution of time delays between rerun-initiating jobs and their reruns, by status of the initiating job. Three key observations emerge from the replication:

4.2.1 Observation 1: Lower but substantial rerun rates in OSS

The proportion of job reruns in OSS projects (7.95%) is lower than in industrial projects (16.23%), aligning with prior work suggesting that CI reliability issues are more prevalent in

industrial settings (Durieux *et al.*, 2020; Olewicki *et al.*, 2022). Nevertheless, the 7.95% rerun rate is substantial, and the rerun practice in OSS projects aligns with that observed at TELUS, with single reruns being most frequent (78.20% of rerun sequences).

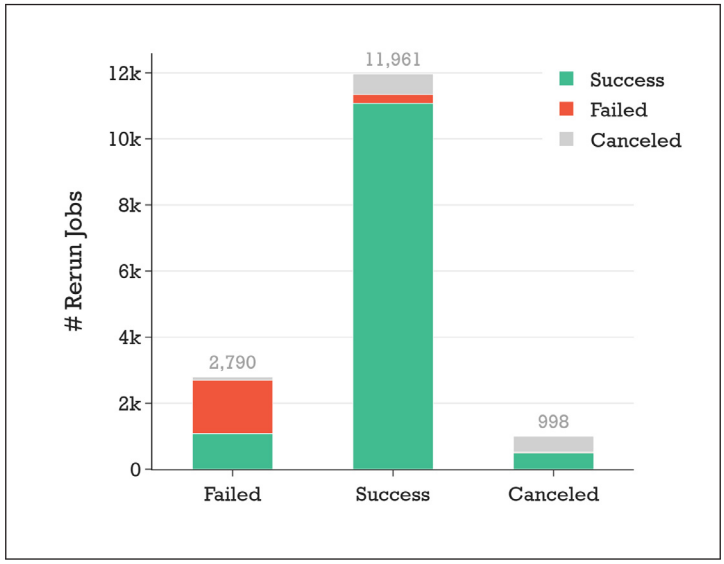


Figure-A IV-4 Evolution of job statuses between rerun-initiating jobs and immediate reruns

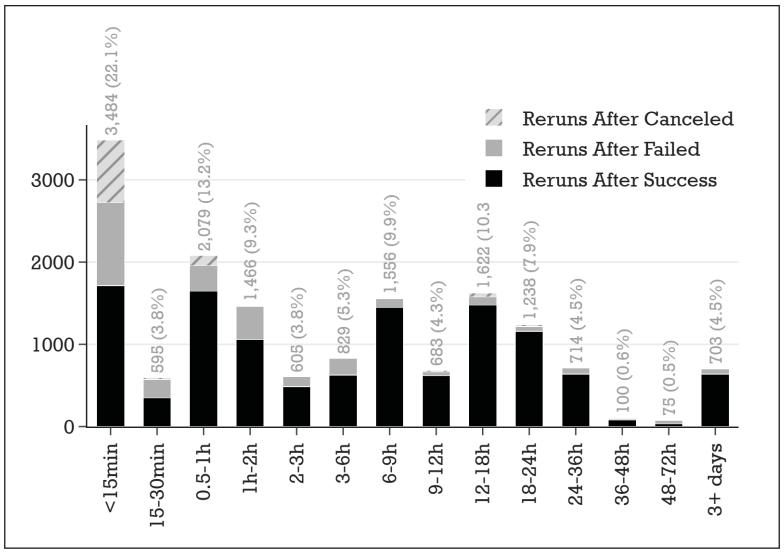


Figure-A IV-5 Distribution of time delays between rerun-initiating jobs and reruns

4.2.2 Observation 2: Predominance of successful job reruns

Similar to the industrial context, reruns of successful jobs are predominant in OSS projects, accounting for 75.95% of all job reruns, even higher than the 51% observed at TELUS. Conversely, the proportion of reruns after failed executions (17.72%) is much lower than in industrial projects (44%). This practice of rerunning successful jobs more frequently than failed jobs suggests the presence of silent failures (with success reruns turning into failures), though further investigation with developer input is recommended before drawing definitive conclusions in the OSS context.

4.2.3 Observation 3: Successful job reruns dominate long-delay reruns

As in the industrial context, reruns of failed and canceled jobs become less common after longer delays. Reruns following failure account for 28.99% of reruns within 15 minutes but only 7.76% of reruns after 6 hours. In contrast, reruns following successful jobs account for 90.94% of all reruns after 6 hours, providing further evidence that findings regarding the prevalence and cost of rerunning successful jobs generalize beyond the industrial context.

4.3 Influential Factors of Rerunning Successful Jobs (RQ_2)

4.3.1 Collinearity Analysis

Figure IV-6 shows the hierarchical clustering of variables for OSS projects using Spearman's rank correlation. Two groups of highly correlated variables ($|\rho| > 0.7$) were identified. The first group contains `commits`, `month_of_year_cos`, `age`, and `author_exp_days`; consistent with the industrial analysis, `commits` was retained. The second group contains `lang_go` and `build_frequency`; `lang_go` was retained since build frequency is a common-wisdom factor. In total, four variables were removed (`month_of_year_cos`, `age`, `author_exp_days`, `build_frequency`), retaining 27 variables.

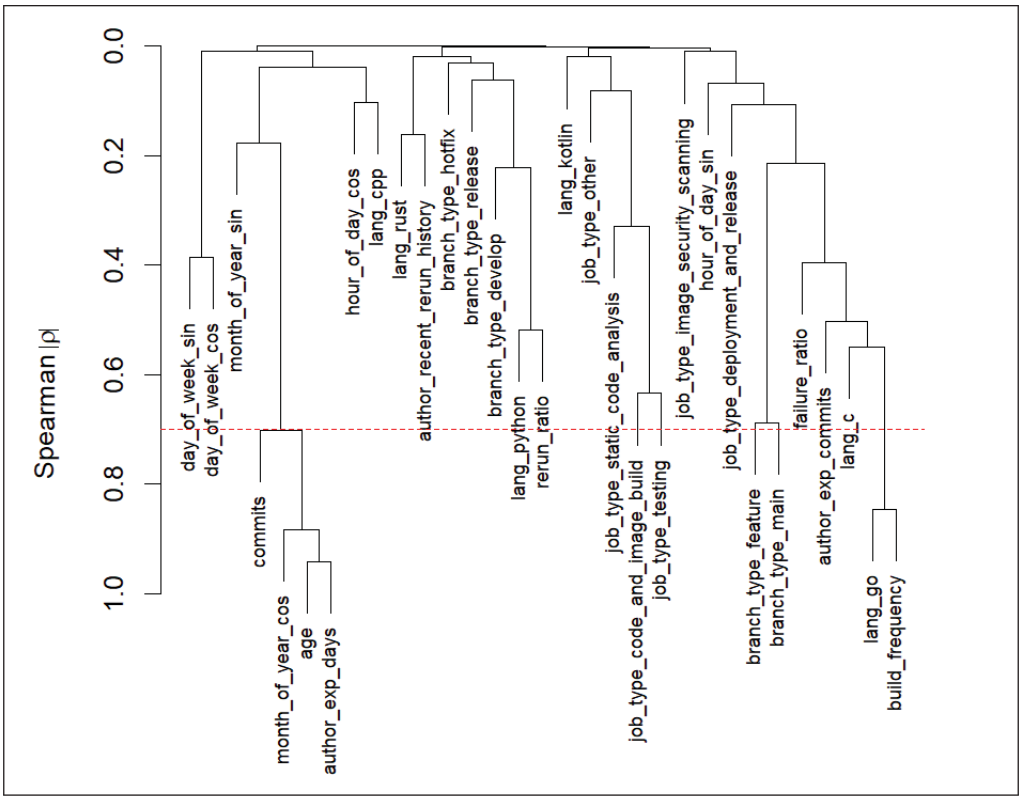


Figure-A IV-6 Hierarchical clustering of variables for OSS projects

4.3.2 Redundancy Analysis

Following the redundancy analysis procedure described in Chapter 7, three variables were identified as redundant: `job_type_deployment_and_release`, `branch_type_main`, and `lang_python`. For consistency with the industrial analysis, these were removed and treated as reference categories for job type, branch type, and language variables, respectively. After this stage, 24 variables remained.

Table-A IV-2 Model performance on OSS projects

ROC AUC	AUC-PR	Brier Score	R^2m	R^2c
0.976	0.849	0.021	0.516	0.570

Table-A IV-3 Results of the mixed-effects logistic model on open-source projects. Positive coefficients indicate higher odds of rerunning successful jobs (↗); negative coefficients indicate lower odds (↘).

F.	Factor	Coef.	χ^2	Pr($<\chi^2$)	Sign.	OR	Rel.
Job Execution Context	(Intercept)	−4.527	24.1	< 9.2e−7	***	0.01	
	job_type_code_and_image_build	0.387	9.0	0.0034	**	1.47	↗
	job_type_image_security_scan	−0.089	9.0	0.0027	**	0.91	↘
	job_type_other	0.134	7.0	0.0073	**	1.14	↗
	job_type_static_code_analysis	0.314	10.0	0.0019	**	1.37	↗
	job_type_testing	0.317	6.0	0.0134	*	1.37	↗
	branch_type_feature	0.280	105.0	< 2.2e−16	***	1.32	↗
	branch_type_hotfix	0.032	2.0	0.2027		1.03	↗
	branch_type_develop	0.364	369.0	< 2.2e−16	***	1.44	↗
	branch_type_release	0.552	1032.0	< 2.2e−16	***	1.74	↗
	month_of_year_sin	−0.033	3.0	0.0997	.	0.97	↘
	day_of_week_sin	0.004	0.0	0.8309		1.00	
	day_of_week_cos	0.057	11.0	0.0009	***	1.06	↗
	hour_of_day_sin	−0.027	2.0	0.1450		0.97	↘
	hour_of_day_cos	0.123	44.0	< 2.9e−11	***	1.13	↗
Project	lang_c	−0.425	1.0	0.2349		0.65	↘
	lang_go	−0.484	1.0	0.3070		0.62	↘
	lang_cpp	0.107	0.0	0.5758		1.11	↗
	lang_rust	0.362	6.0	0.0115	*	1.44	↗
	lang_kotlin	0.006	0.0	0.8767		1.01	↗
	rerun_ratio	−0.322	77.0	< 2.2e−16	***	0.72	↘
	failure_ratio	−0.109	12.0	0.0005	***	0.90	↘
	commits	−0.079	12.0	0.0005	***	0.92	↘
Dev.	author_exp_commits	0.349	38.0	< 8.3e−10	***	1.42	↗
	author_recent_rerun_history	1.643	43391.0	< 2.2e−16	***	5.17	↗

Significance codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1. OR = Odds Ratio; Rel. = Relationship.

4.3.3 Model Performance

Table IV-2 reports the performance of the mixed-effects logistic regression model on OSS projects. The model demonstrates strong discriminatory power (ROC AUC = 0.976) and excellent precision (AUC-PR = 0.849, significantly higher than the baseline of 0.06). The low Brier score (0.021) indicates well-calibrated predictions. The ratio of marginal R^2 to conditional R^2 (approximately 91%) indicates that fixed effects account for the majority of explained variance.

These results support the generalizability of the finding that rerunning successful jobs is driven by factors distinct from those associated with regular job success.

4.3.4 Variable Significance

Table IV-3 presents the significance and influence of each independent variable. While some variables exhibit similar trends to the industrial context (e.g., testing and static code analysis job types show higher odds of rerun), others show opposite effects (e.g., `commits`). These differences should be interpreted cautiously, as understanding variable influences in OSS projects requires contextual knowledge of project practices. For instance, highly experienced developers may contribute to open-source projects only on specific issues, resulting in few commits that may cause them to be falsely classified as inexperienced. The industrial findings benefited from collaboration with TELUS engineers who provided contextual insights; similar engagement with OSS maintainers would be needed to rigorously interpret these results.

BIBLIOGRAPHY

(2019). Some Docker Gitlab Runner jobs fail at 4:20 AM (#4578) · Issues · GitLab.org / gitlab-runner · GitLab. Retrieved on 2025-07-22 from: https://gitlab.com/gitlab-org/gitlab-runner/-/issues/4578.

(2021). Windows gitlab runner problems when using SSH RPC (#28074) · Issues · GitLab.org / gitlab-runner · GitLab. Retrieved on 2025-07-25 from: https://gitlab.com/gitlab-org/gitlab-runner/-/issues/28074.

Abdalkareem, R., Mujahid, S. & Shihab, E. (2021a). A Machine Learning Approach to Improve the Detection of CI Skip Commits. *IEEE Transactions on Software Engineering*, 47(12), 2740 – 2754. Retrieved from: http://dx.doi.org/10.1109/TSE.2020.2967380.

Abdalkareem, R., Mujahid, S., Shihab, E. & Rilling, J. (2021b). Which Commits Can Be CI Skipped? *IEEE Transactions on Software Engineering*, 47(3), 448 – 463. Retrieved from: http://dx.doi.org/10.1109/TSE.2019.2897300.

Akiba, T., Sano, S., Yanase, T., Ohta, T. & Koyama, M. (2019). Optuna: A Next-generation Hyperparameter Optimization Framework. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, (KDD '19), 2623–2631. doi: 10.1145/3292500.3330701.

Akli, A., Haben, G., Habchi, S., Papadakis, M. & Le Traon, Y. (2023). FlakyCat: Predicting Flaky Tests Categories using Few-Shot Learning. *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*, pp. 140–151. doi: 10.1109/AST58925.2023.00018.

Aleem, S., Capretz, L. F. & Ahmed, F. (2016). Game development software engineering process life cycle: a systematic review. *Journal of Software Engineering Research and Development*, 4(1), 6. doi: 10.1186/s40411-016-0032-7.

Allamanis, M., Barr, E. T., Devanbu, P. & Sutton, C. (2018). A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.*, 51(4), 81:1–81:37. doi: 10.1145/3212695.

Alshammari, A., Morris, C., Hilton, M. & Bell, J. (2021). FlakeFlagger: Predicting Flakiness Without Rerunning Tests. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1572–1584. doi: 10.1109/ICSE43902.2021.00140.

Alves Gomes, M. & Meisen, T. (2023). A review on customer segmentation methods for personalized customer targeting in e-commerce use cases. *Information Systems and e-Business Management*, 21(3), 527–570. doi: 10.1007/s10257-023-00640-4.

Angluin, D. (1988). Queries and Concept Learning. *Machine Learning*, 2(4), 319–342. doi: 10.1023/A:1022821128753.

Atlassian. (2024). Gitflow Workflow | Atlassian Git Tutorial. Retrieved on 2024-02-01 from: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.

Aïdasso, H. [Artwork Size: 1878764485 Bytes Pages: 1878764485 Bytes]. (2025a). Artifact for Towards Build Optimization Using Digital Twins. figshare. Retrieved on 2026-04-06 from: https://figshare.com/articles/software/Artifact_for_Towards_Build_Optimization_Using_Digital_Twins/27641388.

Aïdasso, H. [arXiv:2503.12312 [cs]]. (2025b). FlakeRanker: Automated Identification and Prioritization of Flaky Job Failure Categories. arXiv. Retrieved on 2026-01-23 from: http://arxiv.org/abs/2503.12312.

Aïdasso, H. (2026). Artifact for Predicting Intermittent Job Failure Categories for Diagnosis Using Few-Shot Fine-Tuned Language Models [software]. figshare. Retrieved on 2026-01-23 from: https://figshare.com/s/003070f1478ba8e87869?file=61272721.

Aïdasso, H., Bordeleau, F. & Tizghadam, A. (2025a). On the Diagnosis of Flaky Job Failures: Understanding and Prioritizing Failure Categories. *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 192–202. doi: 10.1109/ICSE-SEIP66354.2025.00023.

Aïdasso, H., Bordeleau, F. & Tizghadam, A. (2025b). Efficient Detection of Intermittent Job Failures Using Few-Shot Learning. *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 632–643. doi: 10.1109/ICSME64153.2025.00064.

Aïdasso, H., Bordeleau, F. & Tizghadam, A. [arXiv:2509.14347 [cs]]. (2025c). On the Illusion of Success: An Empirical Study of Build Reruns and Silent Failures in Industrial CI. arXiv. Retrieved on 2026-02-03 from: http://arxiv.org/abs/2509.14347.

Aïdasso, H., Bordeleau, F. & Tizghadam, A. (2025d). Replication Package for On the Illusion of Success in Industrial CI [software]. figshare. Retrieved on 2025-09-15 from: https://figshare.com/s/0a7df599167838533bdb.

Aïdasso, H., Bordeleau, F. & Tizghadam, A. (2025e). Towards Build Optimization Using Digital Twins. *Proceedings of the 21st International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 95–98. doi: 10.1145/3727582.3728684.

Aïdasso, H., Sayagh, M. & Bordeleau, F. (2025f). Build Optimization: A Systematic Literature Review. *ACM Comput. Surv.*, 58(1), 12:1–12:38. doi: 10.1145/3757912.

Aïdasso, H., Bordeleau, F. & Tizghadam, A. [arXiv:2601.22264 [cs]]. (2026). Predicting Intermittent Job Failure Categories for Diagnosis Using Few-Shot Fine-Tuned Language Models. arXiv. Retrieved on 2026-02-03 from: http://arxiv.org/abs/2601.22264.

Bansal, A., Balaji, K. & Lalani, Z. [arXiv:2503.15456 [cs]]. (2025). Temporal Encoding Strategies for Energy Time Series Prediction. arXiv. Retrieved on 2025-08-07 from: http://arxiv.org/abs/2503.15456.

Baral, T., Rahman, S., Chanumolu, B. N., Balcı, B., Tuncer, T., Shi, A. & Lam, W. (2023). Optimizing Continuous Development by Detecting and Preventing Unnecessary Content Generation. *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 901–913. doi: 10.1109/ASE56229.2023.00216.

Barrak, A., Eghan, E. E., Adams, B. & Khomh, F. (2021). Why do builds fail?—A conceptual replication study. *Journal of Systems and Software*, 177, 110939. doi: 10.1016/j.jss.2021.110939.

Beheshtian, M. J., Bavand, A. H. & Rigby, P. C. (2022). Software Batch Testing to Save Build Test Resources and to Reduce Feedback Time. *IEEE Transactions on Software Engineering*, 48(8), 2784 – 2801. Retrieved from: http://dx.doi.org/10.1109/TSE.2021.3070269.

Belcak, P., Heinrich, G., Diao, S., Fu, Y., Dong, X., Muralidharan, S., Lin, Y. C. & Molchanov, P. [arXiv:2506.02153 [cs]]. (2025). Small Language Models are the Future of Agentic AI. arXiv. Retrieved on 2026-01-03 from: http://arxiv.org/abs/2506.02153.

Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T. & Marinov, D. (2018). DeFlaker: Automatically Detecting Flaky Tests. *Proceedings - International Conference on Software Engineering*, 2018-January, 433 – 444. Retrieved from: http://dx.doi.org/10.1145/3180155.3180164.

Beller, M., Bacchelli, A., Zaidman, A. & Juergens, E. (2014). Modern code reviews in open-source projects: which problems do they fix? *Proceedings of the 11th Working Conference on Mining Software Repositories*, (MSR 2014), 202–211. doi: 10.1145/2597073.2597082.

Beller, M., Gousios, G. & Zaidman, A. (2017a). Oops, my tests broke the build: an explorative analysis of Travis CI with GitHub. *Proceedings of the 14th International Conference on Mining Software Repositories*, (MSR '17), 356–367. doi: 10.1109/MSR.2017.62.

Beller, M., Gousios, G. & Zaidman, A. (2017b). TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 447–450. doi: 10.1109/MSR.2017.24.

Benjamin, J. & Mathew, J. (2024). Enhancing continuous integration predictions: a hybrid LSTM-GRU deep learning framework with evolved DBSO algorithm. *Computing*, 107(1), 9. doi: 10.1007/s00607-024-01370-2.

Benjamin, J., Mathew, J. & Jose, R. T. (2023). Study of the contextual factors of a project affecting the build performance in Continuous Integration. *2023 Annual International Conference on Emerging Research Areas: International Conference on Intelligent Systems (AICERA/ICIS)*, pp. 1–6. doi: 10.1109/AICERA/ICIS59538.2023.10420309.

Beyer, B., Jones, C., Petoff, J. & Murphy, N. R. (2016). *Site Reliability Engineering: How Google Runs Production Systems*. "O'Reilly Media, Inc.".

Birant, D. (2011). Data Mining Using RFM Analysis. In Funatsu, K. (Ed.), *Knowledge-Oriented Applications in Data Mining*. InTech. doi: 10.5772/13683.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. New York: Springer.

Bisong, E., Tran, E. & Baysal, O. (2017). Built to Last or Built Too Fast? Evaluating Prediction Models for Build Times. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 487–490. doi: 10.1109/MSR.2017.36.

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., Arx, S. v., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., Brynjolfsson, E., Buch, S., Card, D., Castellon, R., Chatterji, N., Chen, A., Creel, K., Davis, J. Q., Demszky, D., Donahue, C., Doumbouya, M., Durmus, E., Ermon, S., Etchemendy, J., Ethayarajh, K., Fei-Fei, L., Finn, C., Gale, T., Gillespie, L., Goel, K., Goodman, N., Grossman, S., Guha, N., Hashimoto, T., Henderson, P., Hewitt, J., Ho, D. E., Hong, J., Hsu, K., Huang, J., Icard, T., Jain, S., Jurafsky, D., Kalluri, P., Karamcheti, S., Keeling, G., Khani, F., Khattab, O., Koh, P. W., Krass, M., Krishna, R., Kuditipudi, R., Kumar, A., Ladhak, F., Lee, M., Lee, T., Leskovec, J., Levent, I., Li, X. L., Li, X., Ma, T., Malik, A., Manning, C. D., Mirchandani, S., Mitchell, E., Munyikwa, Z., Nair, S., Narayan, A., Narayanan, D., Newman, B., Nie, A., Niebles, J. C., Nilforoshan, H., Nyarko, J., Ogut, G., Orr, L., Papadimitriou, I., Park, J. S., Piech, C., Portelance, E., Potts, C., Raghunathan, A., Reich, R., Ren, H., Rong, F., Roohani, Y., Ruiz, C., Ryan, J., Ré, C., Sadigh, D., Sagawa, S., Santhanam, K., Shih, A., Srinivasan, K., Tamkin, A., Taori, R., Thomas, A. W., Tramèr, F., Wang, R. E., Wang, W., Wu, B., Wu, J., Wu, Y., Xie, S. M., Yasunaga, M., You, J., Zaharia, M., Zhang, M., Zhang, T., Zhang, X., Zhang, Y., Zheng, L., Zhou, K. & Liang, P. [arXiv:2108.07258 [cs]]. (2022). On the Opportunities and Risks of Foundation Models. arXiv. Retrieved on 2026-02-03 from: http://arxiv.org/abs/2108.07258.

Brandt, C. E., Panichella, A., Zaidman, A. & Beller, M. (2020). LogChunks: A Data Set for Build Log Analysis. *Proceedings of the 17th International Conference on Mining Software Repositories*, (MSR '20), 583–587. doi: 10.1145/3379597.3387485.

Brier, G. W. (1950). Verification of Forecasts Expressed in Terms of Probability. *American Meteorological Society*. Retrieved from: https://journals.ametsoc.org/view/journals/mwre/78/1/1520-0493_1950_078_0001_vofeit_2_0_co_2.xml.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I. & Amodei, D. (2020). Language models are few-shot learners. *Proceedings of the 34th International Conference on Neural Information Processing Systems*, (NIPS '20), 1877–1901.

Brus, Y., Maipradit, R., Barr, E. T. & McIntosh, S. (2025). Rechecking Recheck Requests in Continuous Integration: An Empirical Study of OpenStack. *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1427–1439. doi: 10.1109/ASE63991.2025.00121.

Camara, B. H. P., Silva, M. A. G., Endo, A. T. & Vergilio, S. R. (2021). What is the Vocabulary of Flaky Tests? An Extended Replication. *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pp. 444–454. doi: 10.1109/ICPC52881.2021.00052.

Cao, Q., Wen, R. & McIntosh, S. (2017). Forecasting the Duration of Incremental Build Jobs. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 524–528. doi: 10.1109/ICSME.2017.34.

Cao, T., Law, M. & Fidler, S. [arXiv:1909.11722 [cs]]. (2020). A Theoretical Analysis of the Number of Shots in Few-Shot Learning. arXiv. Retrieved on 2025-05-24 from: http://arxiv.org/abs/1909.11722.

Carter, B., Mueller, J., Jain, S. & Gifford, D. [arXiv:1810.03805 [cs]]. (2019). What made you do this? Understanding black-box decisions with sufficient input subsets. arXiv. Retrieved on 2026-04-08 from: http://arxiv.org/abs/1810.03805.

Celik, A., Knaust, A., Milicevic, A. & Gligoric, M. (2016). Build system with lazy retrieval for Java projects. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, (FSE 2016), 643–654. doi: 10.1145/2950290.2950358.

Chawla, N. V., Bowyer, K. W., Hall, L. O. & Kegelmeyer, W. P. (2002). SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, 16, 321–357. doi: 10.1613/jair.953.

Chen, B., Chen, L., Zhang, C. & Peng, X. (2020). BUILDFAST: History-Aware Build Outcome Prediction for Fast Feedback and Reduced Cost in Continuous Integration. *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 42 – 53. Retrieved from: http://dx.doi.org/10.1145/3324884.3416616.

Cohen, J. (1968). Weighted kappa: nominal scale agreement with provision for scaled disagreement or partial credit. *Psychological Bulletin*, 70(4), 213–220. doi: 10.1037/h0026256.

Corbin, J. & Strauss, A. *Sage Research Methods - Basics of Qualitative Research (3rd ed.): Techniques and Procedures for Developing Grounded Theory*. Retrieved from: https://methods.sagepub.com/book/mono/basics-of-qualitative-research/toc.

Corbin, J. & Strauss, A. (2008). *Basics of Qualitative Research (3rd ed.): Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, Inc. doi: 10.4135/9781452230153.

Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. [arXiv:1810.04805 [cs]]. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv. Retrieved on 2025-04-05 from: http://arxiv.org/abs/1810.04805.

Dimitropoulos, P., Aung, Z. & Svetinovic, D. (2017). Continuous integration build breakage rationale: Travis data case study. *2017 International Conference on Infocom Technologies and Unmanned Systems (Trends and Future Directions) (ICTUS)*, pp. 639–645. doi: 10.1109/ICTUS.2017.8286087.

Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM*, 55(10), 78–87. doi: 10.1145/2347736.2347755.

Durieux, T., Le Goues, C., Hilton, M. & Abreu, R. (2020). Empirical Study of Restarted and Flaky Builds on Travis CI. *Proceedings of the 17th International Conference on Mining Software Repositories*, pp. 254–264. doi: 10.1145/3379597.3387460.

Eck, M., Palomba, F., Castelluccio, M. & Bacchelli, A. (2019). Understanding flaky tests: the developer's perspective. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (ESEC/FSE 2019), 830–840. doi: 10.1145/3338906.3338945.

El-Masri, D., Petrillo, F., Guéhéneuc, Y.-G., Hamou-Lhadj, A. & Bouziane, A. (2020). A systematic literature review on automated log abstraction techniques. *Information and Software Technology*, 122, 106276. doi: 10.1016/j.infsof.2020.106276.

Elazhary, O., Werner, C., Li, Z. S., Lowlind, D., Ernst, N. & Storey, M.-A. (2022). Uncovering the Benefits and Challenges of Continuous Integration Practices. *IEEE Transactions on Software Engineering*, 48(7), 2570–2583. doi: 10.1109/TSE.2021.3064953. arXiv:2103.04251 [cs].

Elsevier. (2022). Engineering Websites Index & Journals Database. Retrieved on 2022-12-20 from: https://www.elsevier.com/solutions/engineering-village/databases.

Esfahani, H., Fietz, J., Ke, Q., Kolomiets, A., Lan, E., Mavrinac, E., Schulte, W., Sanches, N. & Kandula, S. (2016). CloudBuild: Microsoft's distributed and caching build service. *Proceedings of the 38th International Conference on Software Engineering Companion*, (ICSE '16), 11–20. doi: 10.1145/2889160.2889222.

Faraway, J. J. (2016). *Extending the Linear Model with R: Generalized Linear, Mixed Effects and Nonparametric Regression Models, Second Edition* (ed. 2). New York: Chapman and Hall/CRC. doi: 10.1201/9781315382722.

Fatima, S., Ghaleb, T. A. & Briand, L. (2023). Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests. *IEEE Transactions on Software Engineering*, 49(4), 1912–1927. doi: 10.1109/TSE.2022.3201209.

Fatima, S., Hemmati, H. & C. Briand, L. (2024). FlakyFix: Using Large Language Models for Predicting Flaky Test Fix Categories and Test Code Repair. *IEEE Transactions on Software Engineering*, 50(12), 3146–3171. doi: 10.1109/TSE.2024.3472476. Conference Name: IEEE Transactions on Software Engineering.

Felizardo, K. R., Mendes, E., Kalinowski, M., Souza, E. F. & Vijaykumar, N. L. (2016). Using Forward Snowballing to update Systematic Reviews in Software Engineering. *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, (ESEM '16), 1–6. doi: 10.1145/2961111.2962630.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D. & Zhou, M. [arXiv:2002.08155 [cs]]. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv. Retrieved on 2025-04-05 from: http://arxiv.org/abs/2002.08155.

Finlay, J., Pears, R. & Connor, A. M. (2014). Data stream mining for predicting software build outcomes using source code metrics. *Information and Software Technology*, 56(2), 183–198. doi: 10.1016/j.infsof.2013.09.001.

Forsgren, N., Humble, J. & Kim, G. (2018). Accelerate: The Science of Lean Software and DevOps Building and Scaling High Performing Technology Organizations | Guide books | ACM Digital Library. IT Revolution Press. Retrieved on 2026-01-27 from: https://dl.acm.org/doi/book/10.5555/3235404.

Gallaba, K., Macho, C., Pinzger, M. & McIntosh, S. (2018). Noise and heterogeneity in historical build data: an empirical study of Travis CI. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, (ASE 2018), 87–97. doi: 10.1145/3238147.3238171.

Gallaba, K., Ewart, J., Junqueira, Y. & McIntosh, S. (2022a). Accelerating Continuous Integration by Caching Environments and Inferring Dependencies. *IEEE Transactions on Software Engineering*, 48(6), 2040 – 2052. Retrieved from: http://dx.doi.org/10.1109/TSE.2020.3048335.

Gallaba, K., Lamothe, M. & McIntosh, S. (2022b). Lessons from Eight Years of Operational Data from a Continuous Integration Service: An Exploratory Case Study of CircleCI. *Proceedings - International Conference on Software Engineering*, 2022-May, 1330 – 1342. Retrieved from: http://dx.doi.org/10.1145/3510003.3510211.

Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M. & Bouchachia, A. (2014). A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4), 44:1–44:37. doi: 10.1145/2523813.

Ghaleb, T., da Costa, D. & Zou, Y. (2019). An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, 24(4), 2102 – 39. Retrieved from: http://dx.doi.org/10.1007/s10664-019-09695-9. Place: Germany.

Ghaleb, T. A., Hassan, S. & Zou, Y. (2022). Studying the Interplay between the Durations and Breakages of Continuous Integration Builds. *IEEE Transactions on Software Engineering*, 1–21. doi: 10.1109/TSE.2022.3222160. Conference Name: IEEE Transactions on Software Engineering.

Ghaleb, T. A., da Costa, D. A., Zou, Y. & Hassan, A. E. (2021). Studying the Impact of Noises in Build Breakage Data. *IEEE Transactions on Software Engineering*, 47(9), 1998–2011. doi: 10.1109/TSE.2019.2941880. Conference Name: IEEE Transactions on Software Engineering.

Gil, S., Mikkelsen, P. H., Gomes, C. & Larsen, P. G. (2024). Survey on open-source digital twin frameworks–A case study approach. *Software: Practice and Experience*, 54(6), 929–960. doi: 10.1002/spe.3305. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3305.

Glaessgen, E. & Stargel, D. (2012). The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles. *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference
20th AIAA/ASME/AHS Adaptive Structures Conference
14th AIAA*. doi: 10.2514/6.2012-1818.

Golzadeh, M., Decan, A. & Mens, T. (2022). On the rise and fall of CI services in GitHub. *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 662–672. doi: 10.1109/SANER53432.2022.00084.

Grafberger, S., Groth, P. & Schelter, S. (2022). Towards data-centric what-if analysis for native machine learning pipelines. *Proceedings of the Sixth Workshop on Data Management for End-To-End Machine Learning*, (DEEM '22), 1–5. doi: 10.1145/3533028.3533303.

Grieves, M. & Vickers, J. (2017). Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems. In Kahlen, F.-J., Flumerfelt, S. & Alves, A. (Eds.), *Transdisciplinary Perspectives on Complex Systems: New Findings and Approaches* (pp. 85–113). Cham: Springer International Publishing. doi: 10.1007/978-3-319-38756-7_4.

Gyori, A., Lambeth, B., Shi, A., Legunsen, O. & Marinov, D. (2016). NonDex: a tool for detecting and debugging wrong assumptions on Java API specifications. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, (FSE 2016), 993–997. doi: 10.1145/2950290.2983932.

Hanley, J. A. & McNeil, B. J. (1982). The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology*, 143(1), 29–36. doi: 10.1148/radiology.143.1.7063747.

Harrell, F. E. (2015). *Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis*. Cham: Springer International Publishing. doi: 10.1007/978-3-319-19425-7.

Hassan, F. (2019). Tackling Build Failures in Continuous Integration. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). Proceedings*, pp. 1242 – 5. Retrieved from: http://dx.doi.org/10.1109/ASE.2019.00150.

Hassan, F. & Wang, X. (2017). Change-Aware Build Prediction Model for Stall Avoidance in Continuous Integration. *International Symposium on Empirical Software Engineering and Measurement*, 2017-November, 157 – 162. Retrieved from: http://dx.doi.org/10.1109/ESEM.2017.23.

Hassan, F. & Wang, X. (2018). HireBuild: an automatic approach to history-driven repair of build scripts. *Proceedings of the 40th International Conference on Software Engineering*, (ICSE '18), 1078–1089. doi: 10.1145/3180155.3180181.

Hassan, F., Meng, N. & Wang, X. (2023). UniLoc: Unified Fault Localization of Continuous Integration Failures. *ACM Trans. Softw. Eng. Methodol.*, 32(6), 136:1–136:31. doi: 10.1145/3593799.

Henkel, J., Bird, C., Lahiri, S. K. & Reps, T. (2020). Learning from, understanding, and supporting devops artifacts for docker. *Proceedings - International Conference on Software Engineering*, pp. 38 – 49. Retrieved from: http://dx.doi.org/10.1145/3377811.3380406.

Herzig, K., Greiler, M., Czerwonka, J. & Murphy, B. (2015). The Art of Testing Less without Sacrificing Quality. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 1, 483–493. doi: 10.1109/ICSE.2015.66.

Hevner, A. R., March, S. T., Park, J. & Ram, S. (2004). Design Science in Information Systems Research. *Management Information Systems Quarterly*, 28, 75–106. Retrieved from: https://www.in.th-nuernberg.de/professors/Holl/Personal/Hevner_DesignScience_ISRes.pdf.

Hills, M. (2020). Introducing DevOps Techniques in a Software Construction Class. *2020 IEEE 32nd Conference on Software Engineering Education and Training (CSEE&T)*, pp. 5 pp. –. Retrieved from: http://dx.doi.org/10.1109/CSEET49119.2020.9206183.

Hilton, M., Tunnell, T., Huang, K., Marinov, D. & Dig, D. (2016). Usage, costs, and benefits of continuous integration in open-source projects. *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 426–437.

Hilton, M., Nelson, N., Tunnell, T., Marinov, D. & Dig, D. (2017). Trade-offs in continuous integration: assurance, security, and flexibility. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, (ESEC/FSE 2017), 197–207. doi: 10.1145/3106237.3106270.

Hong, Y., Tantithamthavorn, C., Pasuksmit, J., Thongtanunam, P., Friedman, A., Zhao, X. & Krasikov, A. (2024). Practitioners' Challenges and Perceptions of CI Build Failure Predictions at Atlassian. *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, (FSE 2024), 370–381. doi: 10.1145/3663529.3663856.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J. & Wang, H. (2024). Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.*, 33(8), 220:1–220:79. doi: 10.1145/3695988.

Islam, M. R. & Zibran, M. F. (2017). Insights into Continuous Integration Build Failures. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 467–470. doi: 10.1109/MSR.2017.30.

Jiang, H., Li, X., Yang, Z. & Xuan, J. (2017). What Causes My Test Alarm? Automatic Cause Analysis for Test Alarms in System and Integration Testing. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 712–723. doi: 10.1109/ICSE.2017.71.

Jin, X. (2021). Reducing cost in continuous integration with a collection of build selection approaches. *ESEC/FSE 2021: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1650 – 4. Retrieved from: http://dx.doi.org/10.1145/3468264.3473103.

Jin, X. & Servant, F. (2020). A cost-efficient approach to building in continuous integration. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 13 – 25. Retrieved from: http://dx.doi.org/10.1145/3377811.3380437.

Jin, X. & Servant, F. (2021a). CIBench: A Dataset and Collection of Techniques for Build and Test Selection and Prioritization in Continuous Integration. *Proceedings - International Conference on Software Engineering*, pp. 166 – 167. Retrieved from: http://dx.doi.org/10.1109/ICSE-Companion52605.2021.00070.

Jin, X. & Servant, F. (2021b). What helped, and what did not? An evaluation of the strategies to improve continuous integration. *Proceedings - International Conference on Software Engineering*, pp. 213 – 225. Retrieved from: http://dx.doi.org/10.1109/ICSE43902.2021.00031.

Jin, X. & Servant, F. (2022). Which builds are really safe to skip? Maximizing failure observation for build selection in continuous integration. *Journal of Systems and Software*, 188, 111292. doi: 10.1016/j.jss.2022.111292.

Jin, X. & Servant, F. (2023). HybridCISave : A Combined Build and Test Selection Approach in Continuous Integration. *ACM Transactions on Software Engineering and Methodology*, 32(4), 1–39. doi: 10.1145/3576038.

Kamath, D. M., Fernandes, E., Adams, B. & Hassan, A. E. (2024). On combining commit grouping and build skip prediction to reduce redundant continuous integration activity. *Empirical Software Engineering*, 29(6), 145. doi: 10.1007/s10664-024-10477-1.

Kawalerowicz, M. & Madeyski, L. (2023). Continuous build outcome prediction: an experimental evaluation and acceptance modelling. *Applied Intelligence*, 53(8), 8673–8692. doi: 10.1007/s10489-023-04523-6.

Kerzazi, N., Khomh, F. & Adams, B. (2014). Why Do Automated Builds Break? An Empirical Study. *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, (ICSME '14), 41–50. doi: 10.1109/ICSME.2014.26.

Khandkar, S. H. (2009). Open coding. *University of Calgary*, 23. Retrieved from: https://cspages.ucalgary.ca/~saul/wiki/uploads/CPSC681/open-coding.pdf.

Khatami, A., Willekens, C. & Zaidman, A. (2024). Catching Smells in the Act: A GitHub Actions Workflow Investigation | IEEE Conference Publication | IEEE Xplore. Retrieved on 2025-01-07 from: https://ieeexplore.ieee.org/abstract/document/10795325?casa_token=CcuT9brDk30AAAAA:TAg92khqkMVFluxrDbCwnuucyPYb93MYFSTIkAaFTjEIvRx5EKnoqmlOuGa7PpJjnTg3JJ9YZyhQ.

Kim, G., Humble, J., Debois, P. & Willis, J. (2016). *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution.

Kitchenham, B. & Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. Retrieved from: https://scholar.google.com/scholar_lookup?title=Guidelines%20for%20performing%20systematic%20literature%20review%20in%20software%20engineering%2C%20Technical%20report%20EBSE-2007-001%2C%20UK&author=B.A.%20Kitchenham&publication_year=2007.

Knuth, D. E. (1998). *The Art of Computer Programming: Sorting and Searching, Volume 3*. Addison-Wesley Professional.

Koch, G., Zemel, R. & Salakhutdinov, R. (2015). Siamese Neural Networks for One-shot Image Recognition. *Proceedings of the 32nd International Conference on Machine Learning*. Retrieved from: https://www.cs.cmu.edu/~rsalakhu/papers/oneshot1.pdf.

Kola-Olawuyi, A., Weeraddana, N. R. & Nagappan, M. (2024). The Impact of Code Ownership of DevOps Artefacts on the Outcome of DevOps CI Builds. *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pp. 543–555. Retrieved from: https://ieeexplore.ieee.org/abstract/document/10555822/authors#authors.

Kumar, A. (2021). *Logistic Regression | MGHIHP HE-802*. Retrieved from: https://bookdown.org/anshul302/HE802-MGHIHP-Spring2020/LogReg1.html.

Lam, W., Godefroid, P., Nath, S., Santhiar, A. & Thummalapenta, S. (2019a). Root causing flaky tests in a large-scale industrial setting. *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, (ISSTA 2019), 101–111. doi: 10.1145/3293882.3330570.

Lam, W., Oei, R., Shi, A., Marinov, D. & Xie, T. (2019b). iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pp. 312–322. doi: 10.1109/ICST.2019.00038.

Lampel, J., Just, S., Apel, S. & Zeller, A. (2021). When life gives you oranges: detecting and diagnosing intermittent job failures at Mozilla. *ESEC/FSE 2021: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1381 – 92. Retrieved from: http://dx.doi.org/10.1145/3468264.3473931.

Lebeuf, C., Voyloshnikova, E., Herzig, K. & Storey, M.-A. (2018). Understanding, Debugging, and Optimizing Distributed Software Builds: A Design Study. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 496–507. doi: 10.1109/ICSME.2018.00060.

Lee, J., Li, M. & Hsu, K.-H. (2024). Applying Transformer Models for Automatic Build Errors Classification of Java-Based Open Source Projects. *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, (ICSE-Companion '24), 282–283. doi: 10.1145/3639478.3643068.

Lehman, M. M. (1979). On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1, 213–221. doi: 10.1016/0164-1212(79)90022-0.

Liang, J., Elbaum, S. & Rothermel, G. (2018). Redefining prioritization: Continuous prioritization for continuous integration. *Proceedings - International Conference on Software Engineering*, pp. 688 – 698. Retrieved from: http://dx.doi.org/10.1145/3180155.3180213.

Lou, Y., Chen, J., Zhang, L., Hao, D. & Zhang, L. (2019). History-driven build failure fixing: how far are we? *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, (ISSTA 2019), 43–54. doi: 10.1145/3293882.3330578.

Lou, Y., Chen, Z., Cao, Y., Hao, D. & Zhang, L. (2020). Understanding build issue resolution in practice: symptoms and fix patterns. *ESEC/FSE 2020: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 617 – 28. Retrieved from: http://dx.doi.org/10.1145/3368089.3409760.

Luo, Q., Hariri, F., Eloussi, L. & Marinov, D. (2014). An empirical analysis of flaky tests. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, (FSE 2014), 643–653. doi: 10.1145/2635868.2635920.

Luo, Y., Zhao, Y., Ma, W. & Chen, L. (2017). What are the Factors Impacting Build Breakage? *2017 14th Web Information Systems and Applications Conference (WISA)*, pp. 139–142. doi: 10.1109/WISA.2017.17.

Macho, C., McIntosh, S. & Pinzger, M. (2016). Predicting Build Co-changes with Source Code Change and Commit Categories. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 1, 541–551. doi: 10.1109/SANER.2016.22.

Macho, C., McIntosh, S. & Pinzger, M. (2018). Automatically repairing dependency-related build breakage. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 106–117. doi: 10.1109/SANER.2018.8330201.

Maes-Bermejo, M., Gallego, M., Gortázar, F., Robles, G. & Gonzalez-Barahona, J. M. (2022). Revisiting the building of past snapshots — a replication and reproduction study. *Empirical Software Engineering*, 27(3), 65. doi: 10.1007/s10664-022-10117-6.

Maipradit, R., Wang, D., Thongtanunam, P., Kula, R. G., Kamei, Y. & McIntosh, S. (2024). Repeated Builds During Code Review: An Empirical Study of the OpenStack Community. *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, (ASE '23), 153–165. doi: 10.1109/ASE56229.2023.00030.

Martin Fowler. (2006). Continuous Integration. Retrieved on 2022-12-20 from: https://martinfowler.com/articles/continuousIntegration.html.

Matthews, B. W. (1975). Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2), 442–451. doi: 10.1016/0005-2795(75)90109-9.

McFadden, D. (1978). Quantitative Methods for Analyzing Travel Behaviour of Individuals: Some Recent Developments. In *Behavioural Travel Modeling* (pp. 279–318). Groom Helm London. Retrieved from: https://eml.berkeley.edu/reprints/mcfadden/quant_methods.pdf.

McIntosh, S., Adams, B., Nguyen, T. H., Kamei, Y. & Hassan, A. E. (2011). An empirical study of build maintenance effort. *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 141–150. doi: 10.1145/1985793.1985813.

Mcintosh, S., Adams, B., Nagappan, M. & Hassan, A. E. (2014). Mining Co-change Information to Understand When Build Changes Are Necessary. *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 241–250. doi: 10.1109/ICSME.2014.46.

Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R. & Micco, J. (2017). Taming Google-scale continuous testing. *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pp. 233–242. doi: 10.1109/ICSE-SEIP.2017.16.

Mesbah, A., Rice, A., Johnston, E., Glorioso, N. & Aftandilian, E. (2019). DeepDelta: Learning to repair compilation errors. *ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 925 – 936. Retrieved from: http://dx.doi.org/10.1145/3338906.3340455.

Mitchell, T. M. (1997). *Machine Learning* (ed. 1). USA: McGraw-Hill, Inc.

Moriconi, F., Troncy, R., Francillon, A. & Zouaoui, J. (2022). Automated identification of flaky builds using knowledge graphs. Retrieved from: https://www.eurecom.fr/en/publication/7046.

Nakagawa, S. & Schielzeth, H. (2013). A general and simple method for obtaining R2 from generalized linear mixed-effects models. *Methods in Ecology and Evolution*, 4(2), 133–142. doi: 10.1111/j.2041-210x.2012.00261.x. _eprint: https://besjournals.onlinelibrary.wiley.com/doi/pdf/10.1111/j.2041-210x.2012.00261.x.

Ni, A. & Li, M. (2017). Cost-effective build outcome prediction using cascaded classifiers. *Proceedings of the 14th International Conference on Mining Software Repositories*, (MSR '17), 455–458. doi: 10.1109/MSR.2017.26.

Ni, A. & Li, M. (2018). ACONA: Active online model adaptation for predicting continuous integration build failures. *Proceedings - International Conference on Software Engineering*, pp. 366 – 367. Retrieved from: http://dx.doi.org/10.1145/3183440.3195012.

Olewicki, D., Nayrolles, M. & Adams, B. (2022). Towards language-independent Brown Build Detection. *Proceedings - International Conference on Software Engineering*, 2022-May, 2177 – 2188. Retrieved from: http://dx.doi.org/10.1145/3510003.3510122.

Pareja, A., Nayak, N. S., Wang, H., Killamsetty, K., Sudalairaj, S., Zhao, W., Han, S., Bhandwaldar, A., Xu, G., Xu, K., Han, L., Inglis, L. & Srivastava, A. [arXiv:2412.13337 [cs]]. (2024). Unveiling the Secret Recipe: A Guide For Supervised Fine-Tuning Small LLMs. arXiv. Retrieved on 2025-05-23 from: http://arxiv.org/abs/2412.13337.

Parry, O., Kapfhammer, G. M., Hilton, M. & McMinn, P. (2020). Flake It 'Till You Make It: Using Automated Repair to Induce and Fix Latent Test Flakiness. *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, (ICSEW'20), 11–12. doi: 10.1145/3387940.3392177.

Parry, O., Kapfhammer, G. M., Hilton, M. & McMinn, P. (2022). Evaluating Features for Machine Learning Detection of Order- and Non-Order-Dependent Flaky Tests. *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 93–104. doi: 10.1109/ICST53961.2022.00021.

Peduzzi, P., Concato, J., Kemper, E., Holford, T. R. & Feinstein, A. R. (1996). A simulation study of the number of events per variable in logistic regression analysis. *Journal of Clinical Epidemiology*, 49(12), 1373–1379. doi: 10.1016/S0895-4356(96)00236-3.

Pinto, G., Reboucas, M. & Castor, F. (2017). Inadequate testing, time pressure, and (over) confidence: a tale of continuous integration users. *2017 IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp. 74 – 7. Retrieved from: http://dx.doi.org/10.1109/CHASE.2017.13.

Pinto, G., Miranda, B., Dissanayake, S., d'Amorim, M., Treude, C. & Bertolino, A. (2020). What is the Vocabulary of Flaky Tests? *Proceedings of the 17th International Conference on Mining Software Repositories*, (MSR '20), 492–502. doi: 10.1145/3379597.3387482.

Pontillo, V., Palomba, F. & Ferrucci, F. (2021). Toward static test flakiness prediction: a feasibility study. *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution*, (MaLTESQuE 2021), 19–24. doi: 10.1145/3472674.3473981.

Rabbani, N., Harvey, M. S., Saquif, S., Gallaba, K. & McIntosh, S. (2018). Revisiting "programmers' build errors" in the visual studio context: A replication study using IDE interaction traces. *Proceedings - International Conference on Software Engineering*, pp. 98 – 101. Retrieved from: http://dx.doi.org/10.1145/3196398.3196469.

Rahman, S. & Shi, A. (2024). FlakeSync: Automatically Repairing Async Flaky Tests. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, (ICSE '24), 1–12. doi: 10.1145/3597503.3639115.

Ramos, J. E. (2003). Using TF-IDF to Determine Word Relevance in Document Queries. Retrieved from: https://www.semanticscholar.org/paper/Using-TF-IDF-to-Determine-Word-Relevance-in-Queries-Ramos/b3bf6373ff41a115197cb5b30e57830c16130c2c.

Randrianaina, G. A., Khelladi, D. E., Zendra, O. & Acher, M. (2022). Towards incremental build of software configurations. *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, (ICSE-NIER '22), 101–105. doi: 10.1145/3510455.3512792.

Rausch, T., Hummer, W., Leitner, P. & Schulte, S. (2017). An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 345–355. doi: 10.1109/MSR.2017.54.

Ravichandran, A., Bhotika, R. & Soatto, S. (2019). Few-Shot Learning With Embedded Class Models and Shot-Free Meta Training. *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 331–339. doi: 10.1109/ICCV.2019.00042.

Reimers, N. & Gurevych, I. [arXiv:1908.10084 [cs]]. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. arXiv. Retrieved on 2025-04-05 from: http://arxiv.org/abs/1908.10084.

Ribeiro, M. T., Singh, S. & Guestrin, C. (2016). "Why Should I Trust You?": Explaining the Predictions of Any Classifier. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (KDD '16), 1135–1144. doi: 10.1145/2939672.2939778.

Runeson, P. & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2), 131–164. doi: 10.1007/s10664-008-9102-8.

Saidani, I., Ouni, A., Chouchen, M. & Mkaouer, M. (2021a). BF-detector: an automated tool for CI build failure detection. *ESEC/FSE 2021: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1530 – 4. Retrieved from: http://dx.doi.org/10.1145/3468264.3473115.

Saidani, I., Ouni, A., Chouchen, M. & Mkaouer, M. W. (2020). Predicting continuous integration build failures using evolutionary search. *Information and Software Technology*, 128, 106392. doi: 10.1016/j.infsof.2020.106392.

Saidani, I., Ouni, A. & Mkaouer, W. (2021b). Detecting Skipped Commits in Continuous Integration Using Multi-objective Evolutionary Search. *IEEE Transactions on Software Engineering*. Retrieved from: http://dx.doi.org/10.1109/TSE.2021.3129165.

Saidani, I., Ouni, A. & Mkaouer, M. W. (2022). Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering*, 29(1). Retrieved from: http://dx.doi.org/10.1007/s10515-021-00319-5.

Saito, T. & Rehmsmeier, M. (2015). The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets. *PloS One*, 10(3), e0118432. doi: 10.1371/journal.pone.0118432.

Santolucito, M., Zhang, J., Zhai, E., Cito, J. & Piskac, R. (2022). Learning CI Configuration Correctness for Early Build Feedback. *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 1006–1017. doi: 10.1109/SANER53432.2022.00118.

Senapathi, M., Buchan, J. & Osman, H. (2018). DevOps Capabilities, Practices, and Challenges: Insights from a Case Study. *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, (EASE '18), 57–67. doi: 10.1145/3210459.3210465.

Seo, H., Sadowski, C., Elbaum, S., Aftandilian, E. & Bowdidge, R. (2014). Programmers' build errors: A case study (at google). *Proceedings - International Conference on Software Engineering*, 0(1), 724 – 734. Retrieved from: http://dx.doi.org/10.1145/2568225.2568255.

Shahin, M., Ali Babar, M. & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, 5, 3909–3943. doi: 10.1109/ACCESS.2017.2685629.

Shan, G. (2022). Monte Carlo cross-validation for a study with binary outcome and limited sample size. *BMC Medical Informatics and Decision Making*, 22(1), 270. doi: 10.1186/s12911-022-02016-z.

Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Vats, I., Moazen, H. & Sarro, F. (2024). A survey on machine learning techniques applied to source code. *Journal of Systems and Software*, 209, 111934. doi: 10.1016/j.jss.2023.111934.

Shimmi, S. & Rahimi, M. (2024). On Association of Code Change Types and CI Build Failures in Software Repositories. *European Journal of Information Technologies and Computer Science*, 4(2), 1–15. doi: 10.24018/compute.2024.4.2.124.

Silva, D., Teixeira, L. & d'Amorim, M. (2020). Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker. *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 301–311. doi: 10.1109/ICSME46990.2020.00037.

Silva, G., Bezerra, C., Uchôa, A. & Machado, I. (2023). What Factors Affect the Build Failures Correction Time? A Multi-Project Study. *Proceedings of the 17th Brazilian Symposium on Software Components, Architectures, and Reuse*, (SBCARS '23), 41–50. doi: 10.1145/3622748.3622753.

Simon, R. (2007). Resampling Strategies for Model Assessment and Selection. In Dubitzky, W., Granzow, M. & Berrar, D. (Eds.), *Fundamentals of Data Mining in Genomics and Proteomics* (pp. 173–186). Boston, MA: Springer US. doi: 10.1007/978-0-387-47509-7_8.

Spencer, D. (2009). *Card Sorting: Designing Usable Categories*. Rosenfeld Media.

Stol, K.-J. & Fitzgerald, B. (2018). The ABC of Software Engineering Research. *ACM Trans. Softw. Eng. Methodol.*, 27(3), 11:1–11:51. doi: 10.1145/3241743.

Strauss, A. & Corbin, J. M. (1998). *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications.

Sulír, M., Bačíková, M., Madeja, M., Chodarev, S. & Juhár, J. (2020). Large-Scale Dataset of Local Java Software Build Results. *Data*, 5(3), 86. doi: 10.3390/data5030086. Number: 3.

Sun, G., Habchi, S. & McIntosh, S. (2024). RavenBuild: Context, Relevance, and Dependency Aware Build Outcome Prediction. *RavenBuild: Context, Relevance, and Dependency Aware Build Outcome Prediction*, 1(FSE), 45:996–45:1018. doi: 10.1145/3643771.

Tufano, M., Sajnani, H. & Herzig, K. (2019). Towards Predicting the Impact of Software Changes on Building Activities. *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 49–52. doi: 10.1109/ICSE-NIER.2019.00021.

Tunstall, L., Reimers, N., Jo, U. E. S., Bates, L., Korat, D., Wasserblat, M. & Pereg, O. [arXiv:2209.11055]. (2022). Efficient Few-Shot Learning Without Prompts. arXiv. Retrieved on 2024-11-15 from: http://arxiv.org/abs/2209.11055.

Turhan, B. (2012). On the dataset shift problem in software engineering prediction models. *Empirical Software Engineering*, 17(1), 62–74. doi: 10.1007/s10664-011-9182-8.

Vassallo, C. (2019). Enabling Continuous Improvement of a Continuous Integration Process. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). Proceedings*, pp. 1246 – 9. Retrieved from: http://dx.doi.org/10.1109/ASE.2019.00151.

Vassallo, C., Proksch, S., Zemp, T. & Gall, H. (2020a). Every build you break: developer-oriented assistance for build failure resolution. *Empirical Software Engineering*, 25(3), 2218 – 57. Retrieved from: http://dx.doi.org/10.1007/s10664-019-09765-y. Place: Germany.

Vassallo, C., Schermann, G., Zampetti, F., Romano, D., Leitner, P., Zaidman, A., Di Penta, M. & Panichella, S. (2017). A Tale of CI Build Failures: An Open Source and a Financial Organization Perspective. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 183–193. doi: 10.1109/ICSME.2017.67.

Vassallo, C., Proksch, S., Zemp, T. & Gall, H. C. (2018). Un-break my build: Assisting developers with build repair hints. *Proceedings - International Conference on Software Engineering*, pp. 41 – 51. Retrieved from: http://dx.doi.org/10.1145/3196321.3196350.

Vassallo, C., Proksch, S., Jancso, A., Gall, H. C. & Di Penta, M. (2020b). Configuration smells in continuous delivery pipelines: a linter and a six-month study on GitLab. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (ESEC/FSE 2020), 327–337. doi: 10.1145/3368089.3409709.

Wang, G., Sun, Z., Chen, Y., Zhao, Y., Liang, Q. & Hao, D. (2024a). Commit Artifact Preserving Build Prediction. *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, (ISSTA 2024), 1236–1248. doi: 10.1145/3650212.3680356.

Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., Zhao, W. X., Wei, Z. & Wen, J. (2024b). A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), 186345. doi: 10.1007/s11704-024-40231-1.

Wang, L., Zhang, X., Su, H. & Zhu, J. (2024c). A Comprehensive Survey of Continual Learning: Theory, Method and Application. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(8), 5362–5383. doi: 10.1109/TPAMI.2024.3367329.

Weeraddana, N., Alfadel, M. & McIntosh, S. (2024a). Characterizing Timeout Builds in Continuous Integration. *IEEE Transactions on Software Engineering*, 50(6), 1450–1463. doi: 10.1109/TSE.2024.3387840. Conference Name: IEEE Transactions on Software Engineering.

Weeraddana, N. R., Alfadel, M. & McIntosh, S. (2024b). Dependency-Induced Waste in Continuous Integration: An Empirical Study of Unused Dependencies in the npm Ecosystem. *Proc. ACM Softw. Eng.*, 1(FSE), 116:2632–116:2655. doi: 10.1145/3660823.

Widder, D. G., Hilton, M., Kastner, C. & Vasilescu, B. (2018). I'm leaving you, Travis: A continuous integration breakup story. *Proceedings - International Conference on Software Engineering*, pp. 165 – 169. Retrieved from: http://dx.doi.org/10.1145/3196398.3196422.

Wieringa, R. J. (2014). *Design Science Methodology for Information Systems and Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-662-43839-8.

Wohlin, C., Mendes, E., Felizardo, K. R. & Kalinowski, M. (2020). Guidelines for the search strategy to update systematic literature reviews in software engineering. *Information and Software Technology*, 127, 106366. doi: 10.1016/j.infsof.2020.106366.

Wolf, T., Schroter, A., Damian, D. & Nguyen, T. (2009). Predicting build failures using social network analysis on developer communication. *Proceedings of the 31st International Conference on Software Engineering*, (ICSE '09), 1–11. doi: 10.1109/ICSE.2009.5070503.

Wróbel, M. R., Szymukowicz, J. & Weichbroth, P. (2023). Using Continuous Integration Techniques in Open Source Projects—An Exploratory Study. *IEEE Access*, 11, 113848–113863. doi: 10.1109/ACCESS.2023.3324536. Conference Name: IEEE Access.

Wu, Y., Zhang, Y., Chang, J., Ding, B., Wang, T. & Wang, H. (2020a). Using Configuration Semantic Features and Machine Learning Algorithms to Predict Build Result in Cloud-Based Container Environment. *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 248–255. doi: 10.1109/ICPADS51040.2020.00042.

Wu, Y., Zhang, Y., Wang, T. & Wang, H. (2020b). An Empirical Study of Build Failures in the Docker Context. *Proceedings of the 17th International Conference on Mining Software Repositories*, (MSR '20), 76–80. doi: 10.1145/3379597.3387483.

Xia, J. & Li, Y. (2017). Could We Predict the Result of a Continuous Integration Build? An Empirical Study. *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 311–315. doi: 10.1109/QRS-C.2017.59.

Xia, J., Li, Y. & Wang, C. (2017). An Empirical Study on the Cross-Project Predictability of Continuous Integration Outcomes. *2017 14th Web Information Systems and Applications Conference (WISA)*, pp. 234–239. doi: 10.1109/WISA.2017.53.

Xia, X., Lo, D., McIntosh, S., Shihab, E. & Hassan, A. E. (2015). Cross-project build co-change prediction. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 311–320. doi: 10.1109/SANER.2015.7081841.

Xiao, S., Liu, Z., Zhang, P., Muennighoff, N., Lian, D. & Nie, J.-Y. [arXiv:2309.07597 [cs]]. (2024). C-Pack: Packed Resources For General Chinese Embeddings. arXiv. Retrieved on 2026-01-16 from: http://arxiv.org/abs/2309.07597.

Yin, M., Kashiwa, Y., Gallaba, K., Alfadel, M., Kamei, Y. & McIntosh, S. (2024). Developer-Applied Accelerations in Continuous Integration: A Detection Approach and Catalog of Patterns. *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, (ASE '24), 1655–1666. doi: 10.1145/3691620.3695533.

Zhang, C., Chen, B., Chen, L., Peng, X. & Zhao, W. (2019). A large-scale empirical study of compiler errors in continuous integration. *ESEC/FSE 2019: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 176 – 87. Retrieved from: http://dx.doi.org/10.1145/3338906.3338917.

Zhang, C., Chen, B., Hu, J., Peng, X. & Zhao, W. (2022). BuildSonic: Detecting and Repairing Performance-Related Configuration Smells for Continuous Integration Builds. 13.

Zhang, Q., Fang, C., Xie, Y., Zhang, Y., Yang, Y., Sun, W., Yu, S. & Chen, Z. [arXiv:2312.15223 [cs]]. (2024). A Survey on Large Language Models for Software Engineering. arXiv. Retrieved on 2026-01-03 from: http://arxiv.org/abs/2312.15223.

Zolfagharinia, M., Adams, B. & Gueheneuc, Y.-G. (2019). A study of build inflation in 30 million CPAN builds on 13 Perl versions and 10 operating systems. *Empirical Software Engineering*, 24(6), 3933 – 71. Retrieved from: http://dx.doi.org/10.1007/s10664-019-09709-6. Place: Germany.

Zolfagharinia, M., Adams, B. & Guéhénuc, Y.-G. (2017). Do Not Trust Build Results at Face Value - An Empirical Study of 30 Million CPAN Builds. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 312–322. doi: 10.1109/MSR.2017.7.