Characterizing Recheck Builds in Continuous Integration

by

Yelizaveta Brus

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Yelizaveta Brus 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Continuous Integration (CI) is a critical component of modern software development, automating the process of downloading, compiling, and testing patch sets. Unsurprisingly, it periodically fails due to non-deterministic (a.k.a., "flaky") behavior. Since a patch set may not be the cause of a flaky failure, developers may issue a "recheck" command to request the CI to re-execute a build request on that patch set. While necessary in some cases, prior work shows that rechecks are often issued carelessly, leading to substantial waste. In the OpenStack community alone, an estimated 187.4 compute years have been wasted on rechecks that resulted in repeated failures. As software development scales, optimizing CI efficiency by reducing wasteful rechecks is essential for improving resource utilization, reducing operational costs, and enhancing developer productivity.

To mitigate unnecessary rechecks, I fit and analyze statistical models that discriminate between recheck requests where a failing outcome will (a) change to passing (i.e., successful rechecks) or (b) continue to fail (i.e., failed rechecks). My empirical study is based on 314,947 recheck requests collected from OpenStack over a 10-year period. I extract and analyze features related to bot behavior, job outcomes, user activity, patch characteristics, and the timing of rechecks. Using logistic regression with restricted cubic splines, I model nonlinear relationships and evaluate predictive performance using AUROC, AUPRC, and Brier score. My model achieves an AUROC of 0.736, outperforming baseline approaches by 23.6 percentage points (or 47.2%), while maintaining strong calibration with a Brier score of 0.191. The model also produces an AUPRC of 0.604, exceeding the baseline performance by 73%. If the requests that my model identifies as failed rechecks were skipped, 86.49% of wasted recheck requests could have been avoided, saving substantial CI resources.

Our analysis reveals that the historical success rates of jobs, bots, and users are the strongest predictors of recheck outcomes. Feature importance analysis shows that the "jobs success ratio" and "bots success ratio" together account for 50% of the explanatory power, indicating that certain jobs and bots are more prone to triggering unnecessary rechecks. In contrast, static patch characteristics, such as the number of modified lines or affected files, contribute minimally to predictive performance. I also investigate the impact of different time windows for feature computation, finding that model performance remains stable regardless of the amount of historical data used. AUROC varies by only 0.59% between a one-day and an all-time window, indicating that both short-term and long-term data can be effectively used to predict recheck outcomes.

Guided by the analysis, I suggest practical steps to improve CI efficiency. Since authors have little control over the past behaviour characteristics of their patch sets, the feedback

of the model may be disheartening for individual contributors who wish to avoid issuing useless rechecks. Instead of applying model insights at the patch set level, my findings indicate that misbehaving bots could be throttled to limit excessive rechecks, unreliable jobs could have their voting power revoked, and automated feedback mechanisms could be introduced to discourage wasteful recheck requests. Additionally, organizations could integrate model-driven insights into CI dashboards, allowing teams to monitor inefficiencies and take corrective action in real time. By focusing on process-level improvements rather than individual developer actions, CI workflows can be optimized to reduce resource waste, accelerate build turnaround times, and enhance the overall developer experience.

Acknowledgements

First and foremost, I am deeply grateful to my supervisor, Shane McIntosh, for his guidance, patience, and unwavering support throughout my research journey. His insights and encouragement have been invaluable.

I extend my gratitude to my thesis readers, Joanne Atlee and Pengyu Nie, for their time, thoughtful feedback, and constructive suggestions that helped shape this work.

A heartfelt thank you to my lab folk, especially Noble for always being supportive, Mattie, the first person I worked under and learned from, Albert for sitting across from me and sharing daily conversations, Raymond for teaching me how to use tmux, and Yiwen for not only teaching me how to drive but also offering constant emotional support. I am also grateful to Ice, my postdoc mentor, whose advice and expertise were always invaluable. To my SWAG labmates Joy, Shaquille, and Favour for taking me to the cinema and making life outside the lab more enjoyable, and Nimmi and Gen for their academic support, I deeply appreciate the encouragement, discussions, and shared moments that made this experience so much richer.

As a member of the University of Waterloo, I acknowledge that this work took place on the traditional territory of the Neutral, Anishinaabe and Haudenosaunee peoples.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements v

List of Figures ix

List of Tables x

1 Introduction 1

2 Core Recheck Concepts 5

3 Study Design 7

3.1 OpenStack Community 7

3.2 (DC) Data Curation 8

3.2.1 (DC-1) Collect change sets data. 9

3.2.2 (DC-2) Label Authors as Bot or User. 9

3.2.3 (DC-3) Identify Rechecks. 10

3.2.4 (DC-4) Identify Recheck Outcomes. 10

3.2.5 (DC-5) Select Rechecks for Prior Failures. 10

3.2.6 (DC-6) Construct Features. 10
3.3 (MF) Model Fitting 11
3.3.1 (MF-1) Perform Collinearity Analysis. 11
3.3.2 (MF-2) Perform Redundancy Analysis. 12
3.3.3 (MF-3) Estimate Budget for Degrees of Freedom (DoF). 12
3.3.4 (MF-4) Allocate Degrees of Freedom. 13
3.3.5 (MF-5) Fit Regression Model. 13
3.4 (MA) Model Analysis 14
3.4.1 (MA-1) Estimate Model Performance. 14
3.4.2 (MA-2) Interpret the Model. 14

4 Overall performance **17**
4.1 (RQ1) How effectively can our model differentiate successful and failed rechecks? 17
4.2 (RQ2) What are the most important characteristics of builds that should be repeated? 20

5 Analysis **26**

6 Threats to Validity **29**
6.1 Construct Validity. 29
6.2 Internal Validity. 29
6.3 External Validity. 30

7 Related Work **31**
7.1 CI Operation. 31
7.2 Build outcome prediction. 32
7.3 Flaky test detection. 32
7.4 CI re-execution. 33

8 Concluding Remarks **34**

References **36**

List of Figures

3.1 Study Design 8

3.2 Hierarchical clustering of features based on Spearman's rank correlation. The line shows $\rho = 0.7$. 12

3.3 Dotplot showing the Spearman multiple ρ^2 of each feature and the likelihood of successful recheck. 12

4.1 In this figure, the threshold balances correctly identifying failing rechecks (precision) against identifying them all (recall). At the threshold 0.2, for instance, 88% of the rechecks skipped are actually failing, and 33% of all failing rechecks are skipped. 19

4.2 Number of rechecks per hour. 22

4.3 Jobs success ratio vs. the probability of successful recheck. 23

4.4 Bots success ratio vs. the probability of successful recheck. 23

4.5 A nomogram without features with a small impact on points. 24

5.1 Feature ranking based on Scott-Knott ESD test using Wald χ^2. 28

List of Tables

3.1 Description and rationale of the selected features. 16

4.1 Importance of families and their individual features based on Wald χ^2. . . 25

5.1 Model performance per different window sizes. 27

Chapter 1

Introduction

Continuous Integration (CI) [9] systems execute routine checks (e.g., compilation, testing) when developers propose change sets. CI systems provide developers with early feedback on code changes, helping them to quickly identify and fix issues before integration into the main codebase. Prior work has associated the adoption of CI with the acceleration of the development process [21], improvements to code quality [49], and the adoption of best practices for automated testing [51].

Despite their benefits, CI systems are imperfect, periodically generating an unreliable pass/fail signal [11]. When a CI build fails, reports are sent to developers who may attempt to diagnose and fix the issue. Unreliable build failures may waste limited developer attention on the diagnosis and attempted fixes of code problems that do not exist. Such unreliable build failures may occur due to infrastructure failures [34], service outages [11], or non-deterministic (a.k.a., "flaky") build behaviour [27, 6, 26].

The developer survey conducted by Maipradit et al. [29] highlights that developers frequently encounter challenges debugging errors caused by flaky builds. When developers believe a failure is not due to their code but rather an external factor, they may request for the build to be re-executed without modifying the patch set. Even without using a tool to analyze whether a build should be re-executed, 20 of the 24 respondents (i.e., regular contributors and community members of influence) still attempt to debug failed builds locally regardless of re-executing, which may lead to debugging non-existing errors. Durieux et al. [5] found that a non-negligible proportion of studied failures (27,006 out of 583,415) had been restarted on the Travis CI platform. Maipradit et al. [29] found that 55% of code reviews within the OpenStack community had rechecked failing builds. Although recheck requests are often a necessary step in ensuring that code is correctly

validated, they can be abused, leading to a substantial waste of resources if developers request rechecks before diagnosing the cause of the failure. According to Maipradit et al. [29], rechecks in the OpenStack community alone have resulted in an estimated 187.4 years of wasted compute time. Durieux et al. [5] found that in 54.42% of cases, developers restarted a build within an hour, suggesting that these requests may be filed prematurely.

While previous studies have primarily focused on quantifying rechecks and the waste that they generate, they offer limited insight into strategies for mitigating wasteful rechecks. Minimizing unnecessary rechecks would not only reduce resource consumption and computational costs platforms like Travis CI and GitHub Actions, but would also benefit project maintainers by lowering operational expenses along with accelerating the development process.

In my study, we fill this gap by exploring the characteristics that influence recheck outcomes by fitting and analyzing statistical models. While AI-driven tools can assist developers in determining whether to re-execute a failed build, they may introduce false positives. Although false positives remain a concern in automated decision-making, my research into developer forums, industry blogs, and other grey literature sources suggests that developers generally accept them as an inherent limitation of AI tools. Many advocate for a hybrid approach that combines automation with human oversight to improve accuracy.

For instance, Anura,[1] a cybersecurity company that provides a highly accurate ad fraud solution, acknowledges the limitations of AI in fraud detection, such as false positives and lack of context [1]. Rather than discarding AI-based solutions, they advocate for combining AI analytics with human judgment to enhance accuracy. Mahesh, an Associate Director of Quality Assurance (QA) at Qburst,[2] argues [28] that false positives in AI-driven testing remain a concern and emphasizes the need for human oversight, where QA professionals review AI-generated outputs to ensure reliability. Responding to this, Shoemaker, Head of Quality Assurance at TestRigor,[3] supports this approach, stating that while AI automates many testing processes, stating that "human intuition is still needed to identify edge cases or unusual behavior" [42]. Similarly, Kalva, an Advisory Board Member at the Cloud Security Alliance,[4] notes that while AI can automate security tasks, false positives require manual review to prevent alert fatigue and ensure that critical threats are not overlooked [37].

We recognize that false positives pose challenges in predictive automation; however,

[1] https://www.anura.io/
[2] https://www.qburst.com/en-ca/
[3] https://testrigor.com/
[4] https://cloudsecurityalliance.org/

my study does not aim to develop a fully autonomous decision-making system, but rather conducts an explanatory analysis. Unlike fully automated approaches, my study focuses on uncovering the underlying patterns that drive a recheck outcome. By understanding these patterns, we provide insights that support decision-making and offer actionable steps for process improvement, such as identifying key factors to prioritize during a recheck request. For my analysis, we examine historical CI data, focusing on the behaviour of bots, jobs, and users to identify patterns that differentiate successful rechecks from failed ones.

We conduct an empirical study of 314,947 recheck requests from the OpenStack community. We structure my study by addressing the following two research questions:

RQ1: How effectively can my model differentiate successful and failed rechecks?
My model achieves an Area Under the Receiver Operator Characteristic (AUROC) curve of 0.736, surpassing the baseline of random guessing (AUROC of 0.5) by 23.6 percentage points or 47.2%, demonstrating its strong discriminatory power. The model also produces a Brier score of 0.191, indicating that its risk estimates are well-calibrated compared to a random guessing model (Brier score of 0.25[5]), outperforming the baseline by 5.6 percentage points or 23.5%. In terms of precision-recall tradeoff, the model achieves an Area Under the Precision Recall Curve (AUPRC) of 0.604, exceeding the baseline performance (AUPRC of 0.349) by 25.5 percentage points or 73%. The optimism penalties for performance metrics are all near zero, indicating that the model is stable and unlikely to be overfitted. My model correctly identifies 177,399 out of 205,122 recheck failures, which if they were skipped, would avoid 86.49% of all failed rechecks and save a substantial amount of build resources.

RQ2: What are the most important characteristics of builds that should be repeated?
Feature families that characterize bot history, job history, patch information, user history, and timing of a recheck request all contribute at least one feature that contributes a significant amount of explanatory power to the model. The "job history" and "bot history" families contribute the largest amount of the explanatory power, alone accounting for 60.19%. More specifically, the "jobs success ratio" and "bots success ratio" features account for 50% of the explanatory power, highlighting that past behaviour dominates when determining whether a recheck will pass. This underscores the importance of focusing on improving the behaviour of jobs and bots to reduce unnecessary rechecks and optimize resource use.

Inspired by the impact of past behaviour on recheck outcomes, we explore whether

[5]Note that lower Brier scores indicate better performance.

recent data trends are being overshadowed by the long history of past data. We analyze how sensitive past behaviour features are to the *window size*, i.e., the time period considered when calculating these features. We experiment with window sizes of one day up to one year, and find that model fitness is not sensitive to window size, with the difference in AUROC between the smallest and largest window sizes being only 0.59%. Brier scores slightly improve with larger windows, while AUPRC decreases, indicating better precision-recall trade-offs for smaller windows. The optimism values remain consistently near zero across all window sizes, suggesting a low optimism bias. Additionally, feature importance remains consistent for highly important features, such as "job success ratio" and "bots success ratio" across all window sizes. The insensitivity of both model fitness and feature importance to changes in window size suggests that either recent data or larger quantities of data can be effectively used to generate actionable recommendations.

My findings may be disheartening for individual contributors, since they are powerless to change the characteristics of their patch sets that influence the likelihood of a recheck to pass. Instead of integrating model feedback at the patch set level, we advocate using its findings to guide collective action, such as focusing process improvement efforts on bots, jobs, and users that misbehave. For example, misbehaving bots can be throttled to limit their access to resources (e.g., by reducing their execution frequency), misbehaving jobs can have their voting power revoked (e.g., by lowering their status from mandatory to optional), and users who recheck efficiently should be rewarded (e.g., by offering recheck efficiency badges on social coding platforms).

Chapter 2

Core Recheck Concepts

The Continuous Integration (CI) process starts when a developer submits to a *change set*, i.e., a collection that includes all proposed changes to the codebase of a project. A change set may contain several *patch sets*, i.e., atomic code changes. Each patch set may elicit general discussion or inline review *comments* from other developers. In addition to the message content itself, each comment submitted to a code review platform like Gerrit includes metadata, describing when the comment was recorded and who submitted the comment.

Once a patch set is submitted, it is common practice for bots to initiate a *build*, i.e., a series of routine steps that may compile, link, assemble, and test code to ensure that the new changes do not introduce regression. This process is managed by *CI bots*, which may interact with external systems like Jenkins and Zuul, or may implement custom logic specified by developers. Each bot may perform one or more *jobs*, each representing a concrete task (e.g., executing a suite of unit or integration tests, running static code analysis, or performing security checks). Each job produces an *outcome*, which describes the result of running a job as either successful or failed. The outcome of a CI bot is successful only if all its jobs are successful (failed otherwise). Similarly, the outcome of a build is successful only if all its voting CI bots are successful (failed otherwise). If the build of a patch set is successful, it is considered validated, and presuming reviewers are also satisfied, the patch set can be queued for integration.

Build failures can occur due to issues unrelated to the code, such as flaky tests that produce inconsistent results, or environmental issues, such as network outages. In these situations, a user can *recheck* the build by issuing a "recheck" request, which repeats the build for the same patch set.

In this study, I focus on rechecks of build failures, as developers who recheck successful builds are likely doing so intentionally, prioritizing other factors over concerns about waste or flakiness. Rechecking a build failures can have two possible outcomes: a "failing" recheck, where the build fails again, or a "successful" recheck, where the build eventually succeeds. A "failing" recheck does not provide new information to the developer but consumes additional CI resources. I consider such rechecks wasteful and recommend skipping them to save resources and improve efficiency.

Chapter 3

Study Design

In this chapter, I describe our procedures for curating the dataset (DC), fitting the model (MF), and performing model analysis (MA). Figure 3.1 shows an overview of our study design. Below I describe each procedure.

3.1 OpenStack Community

I select the OpenStack community for our study because it supports recheck functionality, and provides detailed guidelines about its use.[1] Unlike other CI services, OpenStack emphasizes the responsible use of the "recheck" command with its guidelines, emphasizing that "CI test resources are very scarce (and becoming more so), so please be extremely sparing when asking the system to re-run tests". In contrast, even though Travis CI also offers the same functionality using the "restart" button, it does not provide such advice about minimizing use of rechecks (likely because the financial incentives are reversed, i.e., the service provider benefits from the use of CI resources).

Even in the stringent OpenStack environment, Maipradit et al. [29] found that ill-advised rechecks produced up to 187.4 compute years of waste. It would be reasonable to expect similar if not worse rates of waste being produced in CI services without guidelines. As recheck data is used for model fitting, collecting high-quality data is essential. A model trained using an overabundance of wasteful rechecks would likely capture basic patterns that are already covered by the community guidelines of more stringent communities.

[1] https://docs.openstack.org/project-team-guide/testing.html

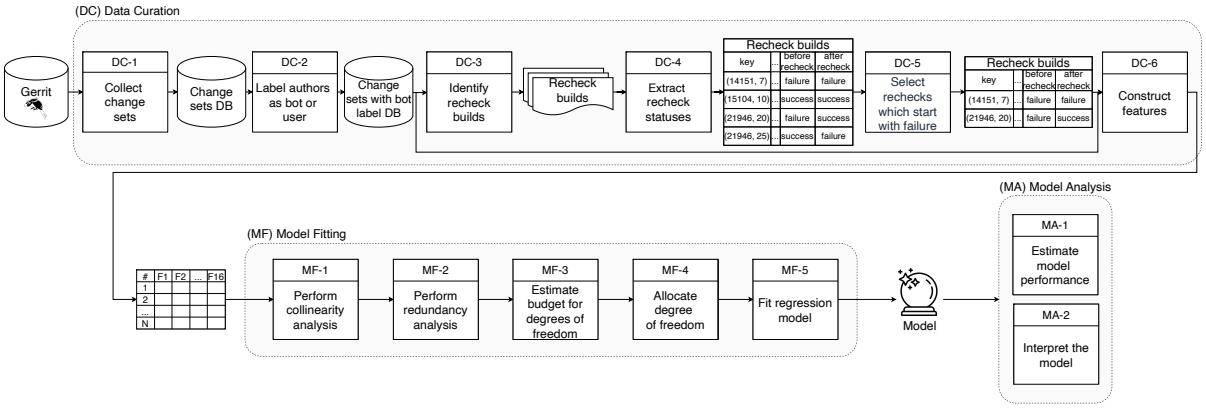


Figure 3.1: Study Design

It is for these reasons that I choose to focus our study on the OpenStack community. OpenStack is an open-source software project for cloud computing infrastructure, ranking among the top three most active open-source projects globally.[2] As an open-source project, OpenStack provides data that is openly accessible. Its large community ensures enough data for analysis.

Our focus on OpenStack rules out other interesting recheck use cases, for instance our current approach focuses on reducing CI costs by identifying rechecks that are likely to fail, helping save resources; however, different workflows may have different priorities. If the goal is to reduce developer distraction, the model could be applied earlier in the workflow to automatically recheck builds without needing developer input. The approach could also work alongside other strategies, i.e., flaky test detection to further optimize CI workflows. By adjusting how the model is used, it is possible to balance saving resources with improving developer experience.

3.2 (DC) Data Curation

In this stage, I collect change sets (i.e., a cohesive set of changes to project files) from projects that are developed by the OpenStack community (DC-1). Then, I categorize authors as bots or a user (DC-2) to identify recheck builds (DC-3) and extract recheck outcomes (DC-4). Finally, I filter the data to focus on rechecks initiated after a failing

[2] https://www.openstack.org/

build (DC-5) and compute features that I suspect may correlate with the likelihood of a recheck build passing (DC-6). I describe each step below.

3.2.1 (DC-1) Collect change sets data.

Prior work of Maipradit et al. [29] studied repeated builds in the OpenStack community. Inspired by Maipradit et al. [29], I collect change sets and user information from OpenStack community projects using the RESTful API of the Gerrit code review tool.[3] To avoid analyzing unfinished or abandoned work, I select the closed change sets that were merged into the "master" branch of each project. Additionally, I collect metadata describing the owner, reviewers, patch, and repository name. The previous dataset from Maipradit et al. [29], included 66,932 change sets from four selected projects. In contrast, our dataset contains 665,137 change sets, 2,254,332 patch sets (i.e., revisions of change sets), and 15,112,495 comments that span 2,285 projects that occurred between May 1, 2014, and April 30, 2024.

3.2.2 (DC-2) Label Authors as Bot or User.

To check whether the author is a bot or a user, I detect authors that have bot characteristics. Among those authors, I manually evaluate their comments. I consider authors as potential bots if:

- Any of the four fields related to author information (i.e., "name", "email", "username", or "display_name") contain either the "ci" or "bot" keywords, e.g., IBM Storage CI, OpenStack Proposal Bot.
- The email field is empty, e.g., Jenkins, Zuul.

As a result, I narrowed the list for manual verification to 7.29% (1,082) of all authors (14,838). During manual verification, I inspect the comments of each bot candidate, classifying them as a user if they contain human text (i.e., include information that is not related to a build outcome) such as code review comments. Otherwise, I consider them as bot. In total, I detect 328 (2.21%) bot accounts and 14,510 (97.79%) user accounts.

[3] https://www.gerritcodereview.com/

3.2.3 (DC-3) Identify Rechecks.

A recheck is an action initiated to repeat the CI process on a patch set due to a previous build failing since the build outcome can change without updating the patch set. I define a build as a combination of CI bots executed on a specific patch set within a single scope. For instance, if a recheck is triggered, it initiates a new build for the same patch set, distinguishing it from the previous one.

To identify rechecks, I search for user-initiated comments that contain the term "recheck". This query produces a set of 487,131 comments. I consider the comment to be issuing a recheck request if it receives a response from a bot. I find that 412,518 of the 487,131 comments issue recheck requests.

3.2.4 (DC-4) Identify Recheck Outcomes.

To identify the status of a recheck request, I analyze the bot-initiated comments that appear following the recheck comment in the same patch set. The outcome of a recheck request is considered successful only if the results of all re-invoked bots are successful; otherwise, it is considered failed.

I use a regular expression to identify the bot result. If the bot result is not available, I summarize the outcome by its job results using regular expressions. Note that I exclude non-voting jobs when determining outcomes because such jobs do not influence the final result of the bot. Therefore, if at least one voting job fails, then I consider that the bot has failed as well.

3.2.5 (DC-5) Select Rechecks for Prior Failures.

I select recheck requests issued after failed builds, resulting in a total of 353,700 rechecks out of 412,518 rechecks. This final dataset includes 238,293 (67.37%) failed rechecks and 115,407 (32.63%) successful rechecks.

3.2.6 (DC-6) Construct Features.

Our features are inspired by previous studies [50] of build outcome and defect prediction. As a result, I formulate 16 features, which are explained in detail in Table 3.1 along

with their feature families.[4] The features are grouped into different families based on their characteristics. These families include bot history, job history, patch content and metadata, the timing of recheck requests, and features that characterize the user who requested the recheck. For the "when" family I use UTC timestamps. I compute the feature values using the database of change sets.

I normalize features belonging to the "patch info" family, including "insertions", "deletions", and "file numbers", as well as "experience" features from the "user" family. This normalization is performed using the L2 norm, also known as the Euclidean norm [3]. It is particularly suitable for our context because it penalizes large values more heavily, reducing the influence of outliers and ensuring that features with larger numeric ranges do not disproportionately affect the analysis.

3.3 (MF) Model Fitting

Before fitting the model, I aim to ensure that the dataset is robust and that I have mitigated issues that could weaken the predictive models. To do so, I perform collinearity analysis (MF-1), redundancy analysis (MF-2), and estimate a budget for degrees of freedom (MF-3). After addressing these issues, I appropriately allocate degrees of freedom (MF-4) and fit the model to the data (MF-5). The steps are described below.

3.3.1 (MF-1) Perform Collinearity Analysis.

Collinear features can distort each other's importance, and interfere with each other when fitting the model [30, 31]. To address this, I apply Spearman's rank correlation (ρ) [44] due to its ability to detect nonlinear correlation compared to other types of correlation measures (e.g., Pearson). Similar to previous studies, I set the threshold as $\rho = 0.7$ [12, 13, 46, 50], i.e., for any pair of features with $\rho > 0.7$, I select only one feature to include in our model fit.

Figure 3.2 provides an overview of the feature correlations hierarchically. From the pairs with $\rho > 0.7$, I select "lines inserted" and "bots success ratio", therefore "number of unique files touched" and "bots flip success ratio" are removed.

[4] I only consider the flip ratio for the bot, when I introduce the flip ratio for job 15.42% (54,534) of the dataset result in N/A.

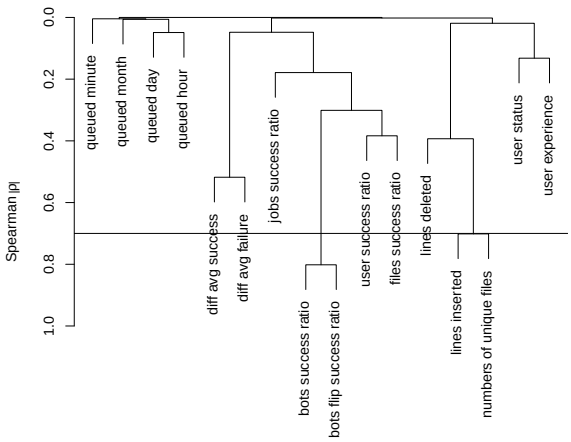


Figure 3.2: Hierarchical clustering of features based on Spearman's rank correlation. The line shows $\rho = 0.7$.

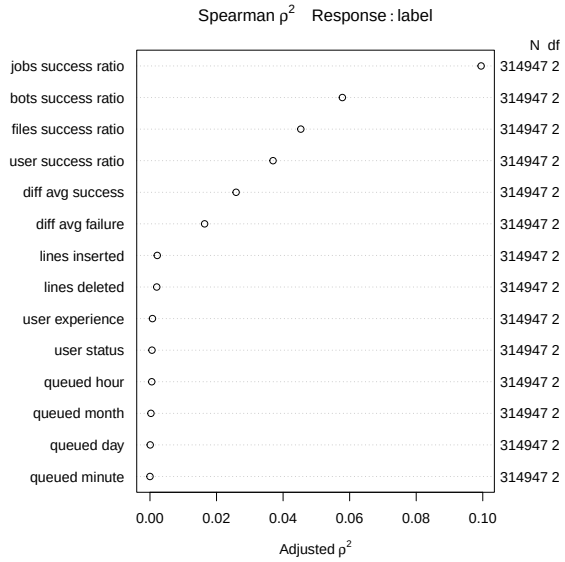


Figure 3.3: Dotplot showing the Spearman multiple ρ^2 of each feature and the likelihood of successful recheck.

3.3.2 (MF-2) Perform Redundancy Analysis.

While initial correlation analysis helps to identify and remove directly correlated features, redundancy analysis is still necessary to capture and address more complex multicollinear relationships among the remaining features. Multicollinearity arises when a feature can be accurately predicted by other features, leading to redundancy and adding noise to model interpretation. To address this, I conduct a redundancy analysis on the remaining 14 features. The analysis fits a set of 14 models where each explains one feature using the 13 other features. A feature is considered redundant if the model fit to explain its values exceeds an R^2 of 0.9 as recommended by Hanley et al. [15]. In our analysis, none of the 14 features exceed the threshold.

3.3.3 (MF-3) Estimate Budget for Degrees of Freedom (DoF).

To capture non-monotonic or nonlinear connections, I allocate additional DoFs to features [8]. A feature with a single DoF can only represent linear or monotonic relationships with

the likelihood of a successful recheck. Allocating additional DoFs allows the model to account for more complex relationships between features and the build outcome; however, allocating too many DoFs can lead to overfitting, where the model becomes overly specific to the training data [16].

To balance between model complexity and overfitting risk, a DoF budget can be established [17, 18]. The purpose of a DoF budget is to limit the total number of DoFs that can be safely spent during model fitting. For logistic regression models, one approach to estimate the DoF budget is by considering the number of records in the minority class. For logistic regression models, this budget can be estimated as shown in Equation (3.1), where n is the number of records in the minority class [17].

$$\text{DoF budget} = \frac{n}{15} \tag{3.1}$$

3.3.4 (MF-4) Allocate Degrees of Freedom.

Figure 3.3 highlights the relationship between features and the likelihood of a successful recheck based on Spearman's multiple ρ^2. I choose features that have higher ρ^2 values than others for DoF allocation. In our case, I select the top six features (e.g, "user success ratio', "build success ratio", "job success ratio", "files success ratio", "diff avg success" and "diff avg failure") to assign a three degrees of freedom for these variables.

3.3.5 (MF-5) Fit Regression Model.

I note that I use traditional statistical analysis and not machine learning to fit this model as I prioritize interpretability and explainability to derive insights from features and their influence. I fit our regression model by selecting the relevant explanatory features and strategically allocating the degrees of freedom. I then fit the model to our data using restricted cubic splines [16] for the features with additional degrees of freedom.

Restricted cubic splines are used to fit relations between variables that are non-linear in nature. They smooth the connection between curves that good at different areas of the range of variable settings. They also keep the tails of the separate curves straight, which helps to avoid overcomplicating the model.

3.4 (MA) Model Analysis

This stage explains how I analyze the model performance (MA-1) and interpret the model (MA-2). I describe each step below.

3.4.1 (MA-1) Estimate Model Performance.

I assess the performance of our logistic regression model according to its: discriminatory power, reliability of risk predictions, and precision-recall trade-off.

I measure *discriminatory power*, i.e., the capacity of the model to distinguish between different classes using the Area Under the Receiver Operating Characteristic Curve (AUROC). An AUROC of 0.5 would be achieved by a random guessing model, whereas an AUROC of 1.0 indicates perfect discrimination, and an AUROC of 0 indicates the worst discrimination, therefore the higher the AUROC the better [15].

To evaluate precision and recall, I use the Area Under the Precision-Recall Curve (AUPRC). The AUPRC is particularly useful for imbalanced datasets, as it focuses on correctly identifying positive cases [41]. The AUPRC is between 0 and 1, with higher values indicating better performance. I compare our AUPRC to a baseline determined by the positive class prevalence. The baseline is calculated using Equation (3.2), where tp and tn are the numbers of true positives and negatives, respectively [41]. For a balanced class distribution, the baseline is 0.5.

$$\text{AUPRC baseline} = \frac{tp}{tp + tn} \tag{3.2}$$

To evaluate the reliability of the predicted probabilities of the model, I use Brier score. Brier score measures the mean squared difference between the predicted probabilities and the actual outcomes. A lower Brier score indicates better calibration, meaning the predicted probabilities closely match the observed outcomes. A Brier score of 1 indicates the worst calibration, whereas 0 indicates the perfect calibration—the lower Brier score the better.

3.4.2 (MA-2) Interpret the Model.

Similar to prior work [50, 31], to interpret the model, I study the features using Wald χ^2 maximum likelihood tests (a.k.a., "chunk"), and plot response curves.

Wald χ^2 maximum likelihood tests help us understand the overall contribution of a feature by comparing the performance of the model with and without it. A higher Wald χ^2 value indicates a greater contribution of the feature to the performance of the model. I also estimate the statistical significance of the features using the p-value.

Additionally, I plot response curves for the most significant features. These curves illustrate how the probability of a successful recheck changes as the value of the feature varies, with all other features being held constant at their "typical" values (median for numeric features and mode for categorical features). The plots also show 95% confidence intervals for the probabilities, calculated from 1,000 bootstrap iterations.

Table 3.1: Description and rationale of the selected features.

Family	Feature	Description	Rationale
Bot history	• bots success ratio	The ratio of successful bot outcomes to the total number of bot calls. For multiple CI bots in one build, we take the minimum.	We hypothesize that a higher ratio increases the likelihood of bot success.
	• flip ratio	The ratio of flips from failure to success to the total flips (failure to success and failure to failure). For multiple CI bots in one build, we take the minimum.	We assume a higher ratio increases the likelihood of the bot flipping to success.
	• difference from average rechecks to success	The difference between the average number of rechecks leading to success and the current number. For multiple CI bots in one build, we take the maximum.	If the current number of rechecks is higher than average, it is more likely to succeed.
	• difference from average rechecks to failure	The difference between the average number of rechecks leading to failure and the current number. For multiple CI bots in one build, we take the minimum.	If the current number of rechecks is higher than average, it is less likely to succeed.
Job history	• jobs success ratio	The ratio of successful job outcomes to the total number of job calls. For multiple jobs in one build, we take the minimum.	We hypothesize that a higher ratio increases the likelihood of current job success.
Patch info	• insertions	The total number of inserted lines of code.	The size of the change corresponding to the build may have some relation to the outcome of the build [35, 39].
	• deletions	The total number of deleted lines of code.	
	• file numbers	The number of unique files touched.	
	• files success ratio	The ratio of successful build outcomes for the file to all builds including the file. For multiple files, we take the minimum.	We assume a higher ratio increases the likelihood of the current build containing the file to succeed.
When	• recheck month	The month when the recheck was issued.	Recheck requests made at specific times may have a higher or lower likelihood of resulting in success [50].
	• recheck day	The day of the week when the recheck was issued.	
	• recheck hour	The hour when the recheck was issued.	
	• recheck minute	The minute when the recheck was issued.	
User	• users success ratio	The ratio of successful outputs after the user called for a build recheck to the total number of rechecks called by the user.	We assume a higher ratio increases the likelihood of success.
	• user status	The position of user within the team (owner, reviewer, or none).	We hypothesize that a higher status increases the likelihood of successful rechecks [31].
	• experience	The number of messages produced by the user, including discussions, code reviews, comments, etc.	We assume that the more experience user have a higher chance of a successful build.

Chapter 4

Overall performance

In this chapter, following an initial assessment of the fitness of our model, I characterize recheck builds that result in a change of outcome by examining the significance of the features of our model.

4.1 (RQ1) How effectively can our model differentiate successful and failed rechecks?

Before fitting the model, I construct features. During this process, I encounter some information for the first time, such as a patch introducing a new file, a user submitting their first recheck, or the appearance of a new bot or job. In such cases, I assign a special value, N/A, to represent the absence of historical data. As our model cannot complete calculations without crashing when N/A values are present, I explore replacing them with 0 (i.e., represent minimum), 1 (i.e., represent maximum), or removing them entirely. The model is fitted for each strategy using the full dataset with the corresponding N/A handling. All three strategies result in similar performance metrics (i.e., AUROC, Brier score, AUPRC) with at most a 7.3 percentage points difference.[1] As the results are not substantially different, I select removing observations with N/A values. In our case, an N/A value indicates that a feature has not yet been observed, so removing it is more appropriate than replacing it, as replacing it would create data that does not exist.

[1]The results for each strategy are available in our online appendix at https://doi.org/10.5281/zenodo.13755309.

After dropping N/A's, 314,947 observations remain, with 109,825 belonging to the minority class (i.e., successful rechecks). Based on Equation (3.1), I can allocate up to $\frac{109{,}825}{15} = 7,321$ degrees of freedom without raising concerns about overfitting. Using Equation (3.2), I estimate the baseline to which the AUPRC of our models should be compared to be $\frac{109{,}825}{314{,}947} = 0.349$.

To investigate the performance of our model, I use AUROC, AUPRC, and Brier score along with assessing the stability of the model fit using bootstrap-calculated optimism [7]. First, I draw a bootstrap sample of length 314,947 from our original dataset with replacement. Next, I refit the logistic regression model to this sample, using the same degrees of freedom. I then evaluate the AUROC, Brier score, and AUPRC of the bootstrap-trained model both on the bootstrap sample and the original dataset. The difference in these performance metrics provides an estimate of the optimism of the model fitness, i.e., the degree to which these performance scores are overestimated. I repeat this process 1,000 times and calculate the mean optimism values. Smaller optimism values indicate a model fit that is more robust to natural fluctuations in the shape of the dataset.

I also evaluate traditional machine learning methods (i.e., Logistic Regression, SVM, XGBoost, and Random Forest) using time-based 10-fold cross-validation. I do not use the single-repetition holdout evaluation method because prior studies [47] have demonstrated that they are the most prone to bias (i.e., producing estimates that differ from a ground truth performance score) and suffer from high variance (i.e., producing estimates that are highly reliant on the point at which data is split). Time-based 10-fold cross-validation is an approach in which the data is sequentially split into 10 parts, with earlier data being used for training and later data for testing. Compared to 1000 bootstrap iterations for logistic regression, results remain within 1 percentage point for AUROC and Brier Score, while AUPRC shows a maximum drop of 3.2 percentage points. Given the negligible differences, I focus on bootstrap results in the thesis and provide time-based cross-validation results in the online appendix.[1] Additionally, I compare the same machine learning methods against each other, observing at most a 2 percentage point variation across all metrics. Compared to the bootstrap-based optimism approach, all results are within 1 percentage point in terms of AUROC and Brier Score, while for AUPRC, I observe a maximum drop of 4.2 percentage points when adopting another model evaluation technique. Since machine learning methods do not substantially improve statistical regression results, I focus on results from statistical regression in the thesis. A detailed comparison is provided in Table 2 in the online appendix.[1]

Observation 1: Our model outperforms naïve baselines in terms of discriminatory power and calibration. Our model achieves an AUROC of 0.736, outperforming random guessing (AUROC of 0.5) by 23.6 percentage points or 47.2%. This suggests that

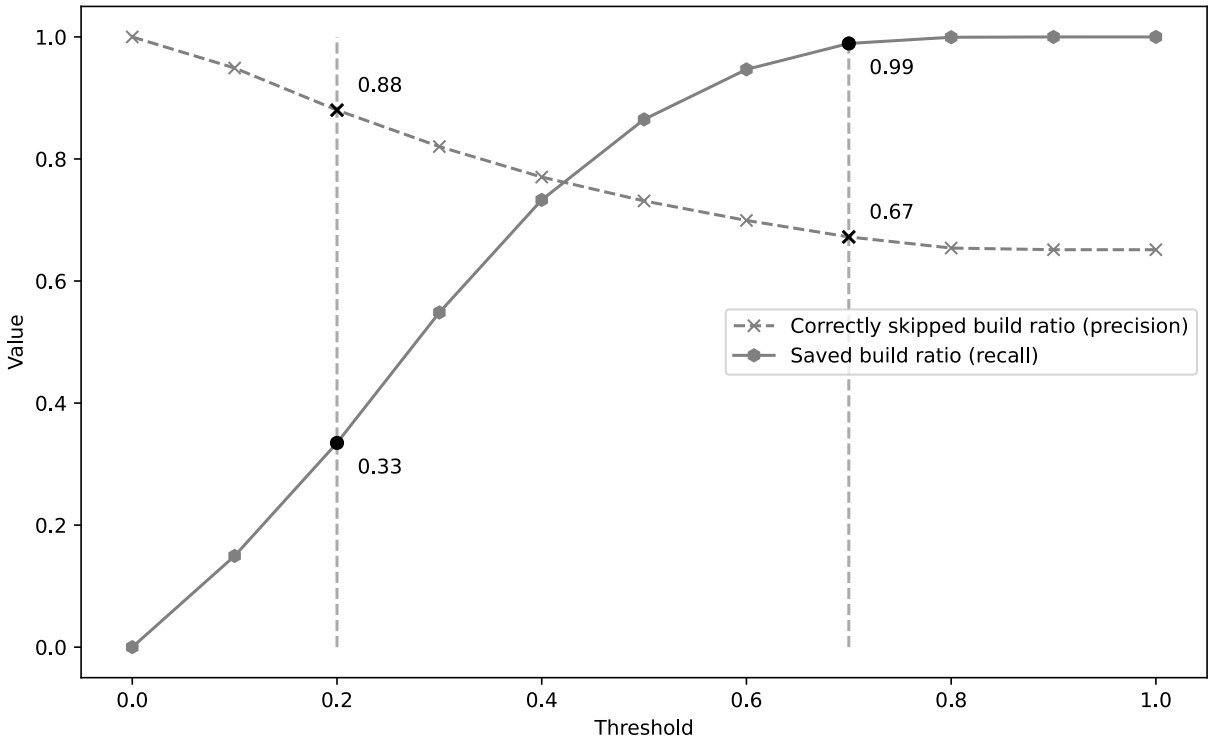


Figure 4.1: In this figure, the threshold balances correctly identifying failing rechecks (precision) against identifying them all (recall). At the threshold 0.2, for instance, 88% of the rechecks skipped are actually failing, and 33% of all failing rechecks are skipped.

our model can differentiate between successful and failed rechecks. Furthermore, our model achieves a Brier score of 0.191. Since a random guessing model would achieve a Brier score of 0.25, our model also outperforms the baseline in terms of the calibration of risk estimates by 5.6 percentage points or 23.5%. Additionally, our model achieves an AUPRC of 0.604, outperforming the baseline determined by the prevalence of the positive class (AUPRC of 0.349) by 25.5 percentage points or 73%. This shows the effectiveness of our model in identifying positive instances while minimizing false positives, which is crucial given the imbalanced nature of our dataset where positive observations (i.e., successful rechecks) are less frequent.

Observation 2: The fit of our model is stable and its explanatory power is robust. The mean optimism value for the AUROC measure is 0.00008, indicating that the AUROC values derived from bootstrap samples are nearly identical to those calculated from

the original dataset [17]. The mean optimism value for Brier score measure is -0.00002, and for the AUPRC is 0.00014. These small optimism scores suggest a low likelihood of the model overfitting the training data.

Observation 3: The model could skip 86.49% of failed rechecks, saving substantial CI resources. Figure 4.1 shows the threshold effects on the balance between savings through skipping failed rechecks (recall) and minimizing lost successes by ensuring that actual failures are skipped (precision). At the default threshold of 0.5, which provides a balanced trade-off between precision and recall, our model could drop 86.49% of all failed rechecks (i.e., 177,399 recheck failures out of 205,122 recheck failures), saving substantial build resources. For cases where the priority is to identify successful rechecks (i.e., precision), a threshold of 0.2 can be selected with a recall of 0.33 and a precision of 0.88. In contrast, if the goal is to save resources (i.e., recall) a threshold of 0.7 is a better choice with a recall of 0.99 and a precision of 0.67.

Key Insight

A predictor can be built that would save computing time.

4.2 (RQ2) What are the most important characteristics of builds that should be repeated?

I analyze the importance of individual features and feature families using Wald χ^2 maximum likelihood tests. Additionally, I analyze response curves for the two features that account for 50% of the explanatory power of our model. The response curve for all features are available in our online appendix.[1]

Table 4.1 presents the Wald χ^2 values for each family of features and individual features in our model. The "Overall" column shows the explanatory power of all degrees of freedom that I allocate to a family or feature, while the "Nonlinear" column shows the explanatory power that is provided by relaxing linearity assumptions between the feature/family and the likelihood of a recheck being successful. A dash (-) is shown in the "Nonlinear" column if no additional degrees of freedom are allocated, i.e., the feature fit is linear.

The Wald χ^2 value for a family of features may not be equal to the sum of the features that comprise its Wald χ^2 values. This is because feature importance is estimated by removing each feature or family and assessing the impact on the model. When a feature

is removed, its explanatory power may, in part, be subsumed by features, resulting in a family χ^2 value that differs from the sum of that of its individual features.

Along with Wald χ^2 value, I use a nomogram to visualize and analyze the results of our model. Each horizontal scale corresponds to a feature and the topmost scale represents the "points" attributed to specific values of these features. The bottom-most scale shows the final predicted probability of an outcome based on the total points accumulated from the values of the features. I drop features with a small impact on points to optimize vertical space, though a complete version is available in our replication package.[1]

Observation 4. All families contribute at least one significant feature to the model fit. Table 4.1 shows all families are significant and contribute meaningfully to the overall model, which supports our hypotheses about each family (see Table 3.1). Overall, "job history" and "bot history" families together account for 60.19% ($\frac{16{,}247.20+7{,}866.04}{40{,}060.51}$) of the explanatory power of the model.

Table 4.1 shows that while each family significantly contributes to the model fit, individual features within these families do not always play a significant role. Specifically, features from the "patch info" family such as "insertions" and "deletions", and from the "when" family such as "queued month", "queued day", and "queued minute", are not significant. Collectively, these features account for 0.01% of the explanatory power of the model.

Despite the overall insignificance of many features within the "when" family, the significance of the "queued hour" feature ensures that the family contributes significantly to the model fit. Figure 4.2 shows that rechecks mostly occur in the afternoon (i.e., 1 PM to 3 PM) with a peak at 2 PM (34,717). Rechecks occur least often at night (i.e., 10 PM to 5 AM) with 4 AM being the minimum (15,399).

Observation 5. Past behaviour is the dominant explanatory factor. Table 4.1 shows that the most important contributor to the explanatory power of the model is past behaviour, rather than characteristics of the patch set, the timing of the recheck, or the role of the user. This finding aligns with prior research on defect prediction [36], which showed that the number of faults in the initial release of source files is an early and strong indicator of future defect rates and reliability. Similarly, in our study, the "files success ratio" feature is one of the key explanatory factors, underscoring the importance of past behaviour in predicting recheck outcomes. The principle "faulty once, faulty forever" applies not only to files but also to jobs, bots, and users, highlighting that the changes to the patch set alone are not enough and require collective action.

Observation 6. Features related to bots and jobs are the strongest in explaining the likelihood of a successful recheck. Features closely linked to the build

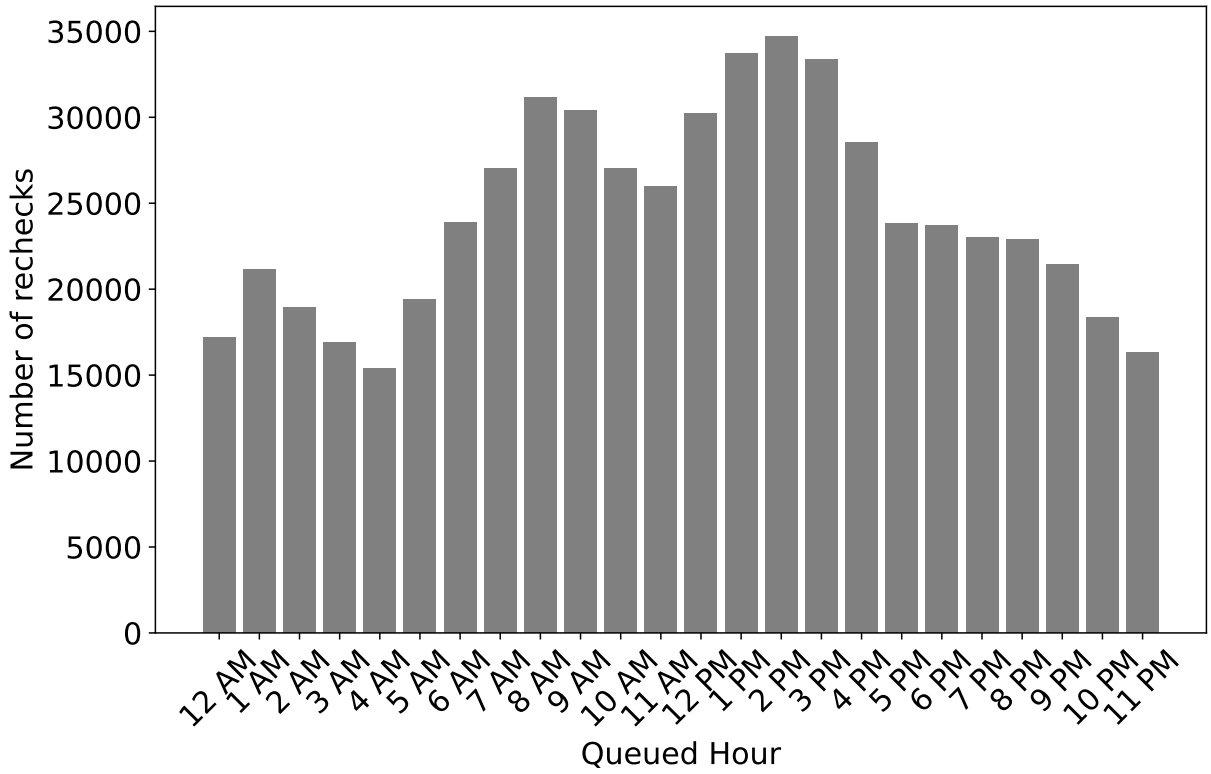


Figure 4.2: Number of rechecks per hour.

process, specifically the "jobs success ratio" and "bots success ratio", account for 50% of the explanatory power of our model. Notably, the "jobs success ratio" contributes significantly to this, accounting for 40.56% of the total explanatory power. I further analyze these two features using response curves.

Figure 4.3 presents the response curve, showing a strong relationship between the "jobs success ratio" and the probability of a successful recheck, with narrow confidence intervals. Notably, the response curve shows an exponential increase, particularly after the "jobs success ratio" exceeds 0.6. This occurs because of how a sample of our dataset supports the trend, e.g., in the dataset, the ratio of failed rechecks to successful ones is 4.36 when the "jobs success ratio" is below 0.6, and the ratio drops to 1.34 when the "jobs success ratio" exceeds 0.6. This shows how key aspects of past behaviour (i.e., past successes or failures) strongly influence recheck outcomes.

The "bots success ratio" follows a similar pattern, highlighting the importance of past

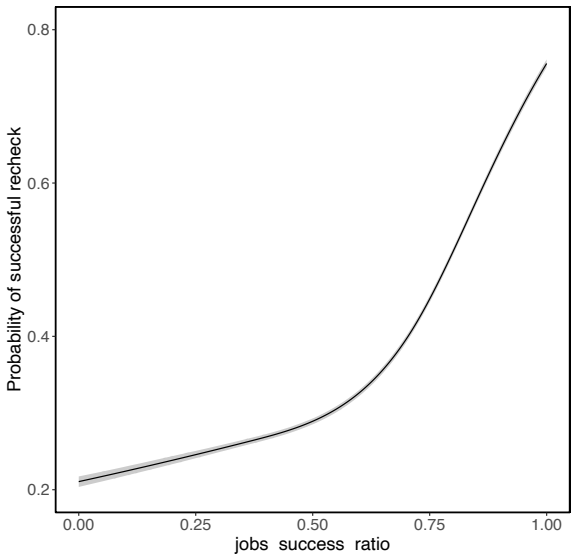


Figure 4.3: Jobs success ratio vs. the probability of successful recheck.

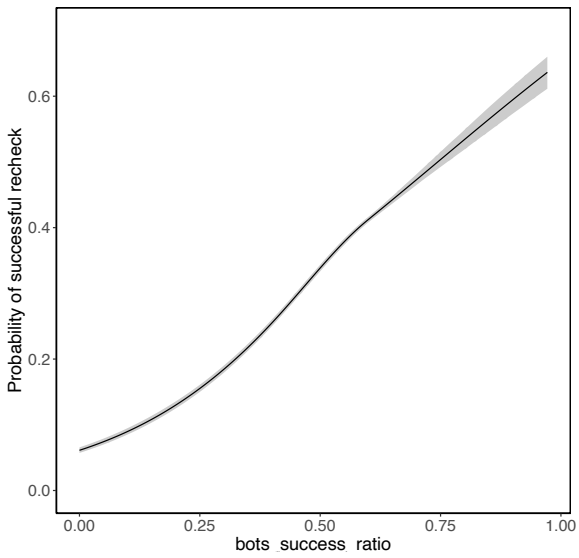


Figure 4.4: Bots success ratio vs. the probability of successful recheck.

behaviour. Figure 4.4 shows a strong relationship between the "bots success ratio" and the probability of a successful recheck, though the confidence intervals begin to widen once the ratio exceeds 0.62. This broadening occurs because a sample of our dataset is likely to have very few data points with a "bots success ratio" higher than 0.62, similar to the entire dataset, where only 1.60% of data points exceed a "bots success ratio" of 0.62.

To understand the impact of the "bots success ratio" on recheck outcomes, I compared rechecks involving one bot to those with multiple bots. Of the total 314,947 rechecks, 208,925 involve a single-bot which is twice as many as those with multiple bots (106,652). Rechecks with multiple bots are 1.5 times more likely to fail, with a prediction accuracy of 0.65 for single-bot rechecks and 0.82 for those with multiple bots. This higher accuracy suggests that multiple-bot rechecks are more predictable, likely due to their higher failure rate. The data underscores how the number of bots involved and their past performance play a crucial role in determining recheck outcomes.

Observation 7. The "diff avg failure" feature value becomes a deciding factor when values are set to extreme. Figure 4.5 shows that even though the "diff avg failure" feature is ranked third by importance, it becomes a deciding factor when values are set to extreme. As the "diff avg failure" feature value increases, the nomogram shows a nonlinear relationship, with a larger gap between values above 0. The difference in

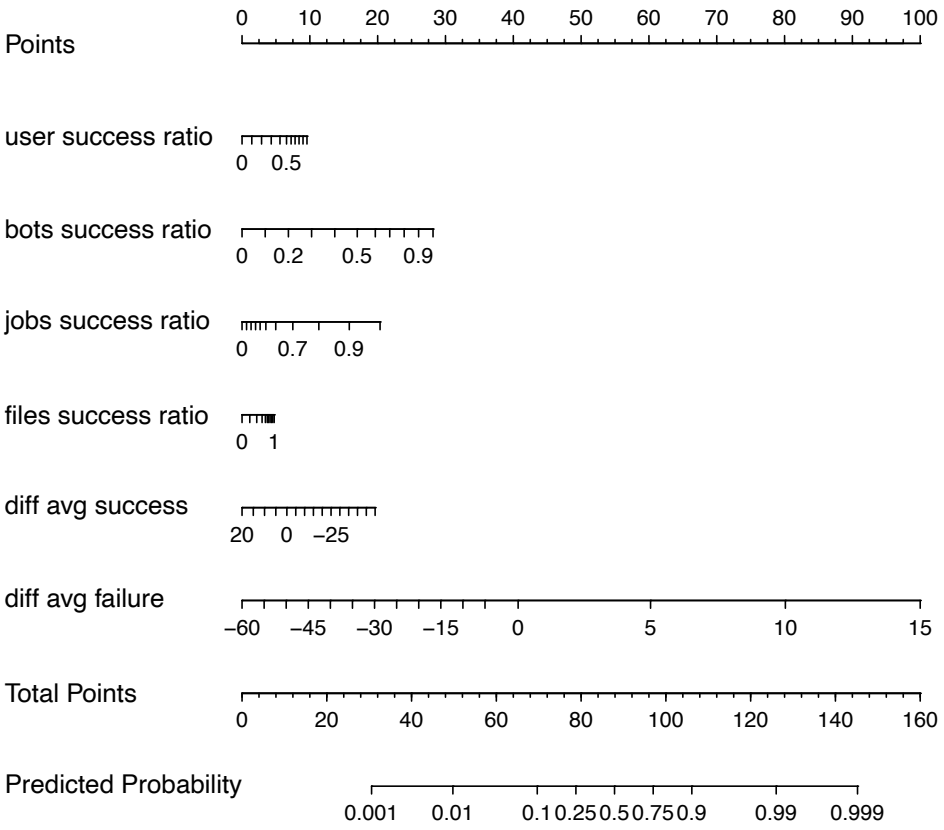


Figure 4.5: A nomogram without features with a small impact on points.

predicted probability between 0.1 and 0.9 is 36 points (i.e., from 72 to 108), which makes up 22.5% of the total points. This shows that small changes in feature values can result in substantial differences in the predicted outcome.

Key Insight

Our model shows that past behaviour dominates recheck outcomes. Thus, effort spent on reducing needless rechecks should focus on identifying and resolving past recheck failures, whether issued by job, bot, or users. For example, misbehaving bots can be throttled to limit their access to resources and misbehaving jobs can have their voting power revoked.

Table 4.1: Importance of families and their individual features based on Wald χ^2.

Family		**Overall (Family)**	**Nonlinear (Family)**	**Feature**		**Overall**	**Nonlinear**
Job history	D.F	2	1	jobs success ratio^{pb}	D.F	2	1
	χ^2	$16,247.20^{***}$	$2,631.09^{***}$		χ^2	$16,247.20^{***}$	$2,631.09^{***}$
Bot history	D.F	6	3	bots success ratio^{pb}	D.F	2	1
	χ^2	$7,866.04^{***}$	$1,008.33^{***}$		χ^2	$3,789.65^{***}$	78.17^{***}
				diff avg failure^{pb}	D.F	2	1
					χ^2	$2,089.11^{***}$	698.57^{***}
				diff avg success^{pb}	D.F	2	1
					χ^2	52.22^{***}	0.68 $\circ$
User	D.F	4	1	user success ratio^{pb}	D.F	2	1
	χ^2	$1,747.41^{***}$	81.84^{***}		χ^2	$1,592.01^{***}$	81.84^{***}
				status	D.F	1	-
					χ^2	91.23^{***}	-
				experience	D.F	1	-
					χ^2	15.40^{**}	-
Patch info	D.F	4	1	files success ratio^{pb}	D.F	2	1
	χ^2	$1,327.20^{***}$	157.57^{***}		χ^2	$1,320.56^{***}$	157.57^{***}
				insertions	D.F	1	-
					χ^2	0.38 $\circ$	-
				deletions	D.F	1	-
					χ^2	0.12 $\circ$	-
When	D.F	4	-	queued hour	D.F	1	-
	χ^2	54.09^{***}	-		χ^2	50.55^{***}	-
				queued month	D.F	1	-
					χ^2	2.09 $\circ$	-
				queued day	D.F	1	-
					χ^2	1.81 $\circ$	-
				queued minute	D.F	1	-
					χ^2	0.07 $\circ$	-
Entire model	D.F	20	6	-		-	-
	χ^2	40,060.51 ***	3,808.05 ***				

$\circ$ $p \geq 0.05$; $^{*}p < 0.05$; $^{**}p < 0.01$; $^{***}p < 0.001$; ^{pb} - past behaviour

Chapter 5

Analysis

In Chapter 4, I observe that past behaviour features are highly important in determining recheck outcomes. These past behaviours have been measured using 10 years of data. Since more recent trends may be outweighed by the bulk of historical data, in this chapter, I set out to study how sensitive past behaviour features are to the *window size*, i.e., the amount of time considered when computing the measures. I refit our recheck outcome model using window sizes of one day up to one year. During feature extraction, I calculate feature values based on data available within a given window. For example, if the window size is set to one day, I use data from the day before a potential recheck to calculate features. To ensure that the features of each observation have been recomputed using the full span of the window size of data, I exclude the first year of observations from model fitting. Note that the first year of data is still used to compute feature values.

Following the approach in RQ1, I exclude observations with N/A values before fitting the model. As the window size decreases, the number of observations also declines, since past behaviour features (e.g., "job success ratio") rely on historical data for their calculation.

In addition to studying the fitness of the refit models, I rank features based on their Wald χ^2 importance scores using the non-parametric Scott-Knott Effect Size Difference (ESD) test [48], as it does not assume normality or homogeneity of distributions. For each window size, I collect the distribution of Wald χ^2 values for each feature from 1,000 bootstrap iterations. Unlike traditional ranking methods, which strictly order features by individual values, the Scott-Knott ESD test groups features with similar explanatory power.

Observation 8: Our model fitness is not sensitive to window size. Table 5.1

Table 5.1: Model performance per different window sizes.

Window size	AUROC	AUROC Opt.	Brier	Brier Opt.	AUPRC	AUPRC Opt.	Obs.	% success recheck
1 day	0.74243	0.00016	0.19442	-0.00005	0.64046	0.00032	145,379	37.6
1 week	0.73806	0.00011	0.19330	-0.00004	0.62359	0.00020	225,124	36.4
2 weeks	0.73818	0.00011	0.19232	-0.00004	0.62242	0.00017	241,444	36.0
1 month	0.74077	0.00005	0.19050	-0.00001	0.61904	0.00014	253,916	35.5
3 months	0.74187	0.00001	0.18929	-0.00001	0.61346	0.00006	263,477	35.0
6 months	0.74147	0.00007	0.18872	-0.00002	0.60825	0.00016	268,653	34.6
1 year	0.74062	0.00001	0.18832	-0.00001	0.60482	0.00004	272,789	34.3
all time	0.73893	0.00002	0.18826	0.00000	0.60128	0.00012	274,797	34.1

shows the model performance across the studied window sizes. With the highest AUROC at 0.74243 for a window size of one day and the lowest at 0.73806 for a window size of two weeks, the 0.59% difference indicates minimal change, suggesting that window size has little impact on the discriminatory power of the model. A similarly small difference in Brier scores across window sizes suggests that the probability estimates of the model are also only negligibly impacted. In contrast, the AUPRC values decrease as window size increases, with the highest value for a window size of one day (0.64046), indicating better precision-recall trade-offs for smaller window sizes. I suspect that AUPRC increases with smaller window sizes are driven by a higher ratio of successful rechecks. Since I remove observations with N/A values, the number of observations decreases with smaller window sizes, and as most observations with N/A values result in recheck failures, the proportion of successful rechecks increases. Optimism values for AUROC, Brier score, and AUPRC range from -0.00005 to 0.00032, indicating a small and stable optimism bias across all window sizes.

Observation 9: Feature importance is not sensitive to window size. Figure 5.1 shows that the "jobs success ratio", "bots success ratio", "diff avg failure", "user success ratio", "files success ratio" are consistently top-ranked features across window sizes. This suggests that tendencies appear rather quickly, and users of this modelling approach need little historical data to produce an actionable model. The only feature to displace a feature from the top is "bots flip success ratio", which is collinear with "bots success ratio" in all fits other than the one-day fit, suggesting that the information that it provides is highly similar to the features in our top-ranked list. I also analyze response curves for the most important feature (i.e., "jobs success ratio") across window sizes and observe no substantial

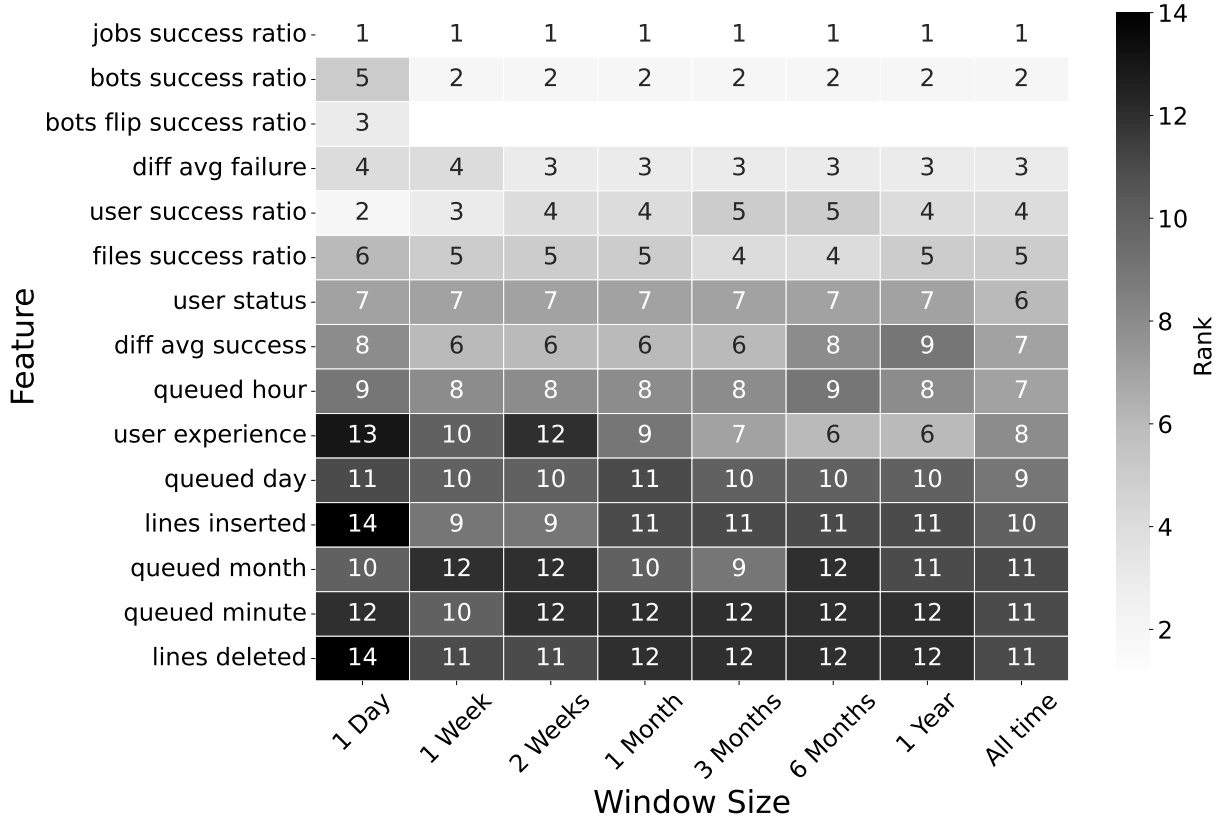


Figure 5.1: Feature ranking based on Scott-Knott ESD test using Wald χ^2.

changes in the shape of curves. It suggests that either recency or quantity of the data can be used for identifying recheck build outcome. Due to limited space, I provide response curves for the "jobs success ratio" for each window size in the online appendix.[1]

Key Insight

Our model fits stabilize after past behaviour features have a window size of one week, suggesting that little data is needed to produce actionable results.

Chapter 6

Threats to Validity

I breakdown the threats into construct, internal, and external validity.

6.1 Construct Validity.

I may not fully capture the real-world performance and oversimplify the complex interactions between CI bots, jobs, and rechecks. For example, I assume that for builds involving multiple bots the "bots success ratio" is the minimum success ratio across those bots. I justify this by noting that if one bot fails, the recheck outcome will also result in failure.

6.2 Internal Validity.

I identify the recheck requests based on the author of the comment being a human user, and not a bot. I may miss recheck requests if I mistake a human for a bot. To mitigate this, I manually label authors as humans or bots and then inspect the results.

I draw conclusions based on the current set of features; however, this set is not exhaustive. For instance, adding features that capture information about network or service status could impact the fit of our model, potentially changing the importance of features affected by these factors. OpenStack does not publicly provide data on network or service status, which prevents us from exploring these aspects.

6.3 External Validity.

As our study focuses on the OpenStack community, it may not generalize to open-source communities or other technologies other than Gerrit. I select the OpenStack community as it supports a recheck functionality and offers stringent guidelines about its use. Nonetheless, replication to other environments may prove useful.

Chapter 7

Related Work

In this chapter, I position our work with respect to the literature on CI operation, build outcome prediction, and CI re-execution.

7.1 CI Operation.

Prior work has proposed areas of improvement for CI service providers that can improve user experience. Gallaba et al. [11] conducted an empirical study of 22.2 million CircleCI builds, discovering that at least 371,647 builds failed due to service availability issues, while 31,238 builds failed as a result of misconfigurations. They highlighted that addressing misconfigurations and improving service availability present significant opportunities for enhancing the robustness of CI services. Weeraddana et al. [50] studied timeout builds in CI systems, observing their strong tendency to occur in clusters. Bouzenia and Pradel [2] conducted an empirical study on GitHub Actions workflows, revealing that CI/CD costs averaged $504 per year for paid-tier repositories, with testing and building consuming 91.2% of resources. Although existing optimizations like caching had a positive impact, adoption remained scarce, underscoring the need for better documentation and tools to promote resource-efficient workflows.

Recent studies on CI operations highlight that build failures and timeout builds can cause problems for CI operations, which in turn result in a worse user experience. Another aspect that should be considered is the waste or cost of these builds, which impacts CI operations. Building on this idea, this thesis quantifies that if our model are adopted, 86.49% of the waste could have been avoided, improving service quality.

7.2 Build outcome prediction.

Several methods have been proposed to predict build outcomes, with the goal of skipping unnecessary builds. Jin and Servant [22, 23] proposed SmartBuildSkip and PreciseBuildSkip, which skips builds predicted as successful and executes ones which are predicted as failure. Kwan et al. [25] studied the effect of socio-technical congruence on build outcomes. They found that congruence could sometimes decrease success rates, and successful builds often exhibited larger socio-technical gaps than failed ones. Saidani et al. [40] applied an evolutionary search algorithm to predict build failures, focusing on optimizing build processes by predicting which builds were likely to succeed or fail based on historical data, code metrics, and system characteristics. Pan and Pradel [35] proposed a model to predict test suite failures on a continuous basis, emphasizing the relationship between test outcomes and overall build reliability. Sun et al. [45] introduced the RavenBuild model, which integrates context, relevance, and dependency-aware features to predict CI build outcomes.

Previous studies found that historical data played a significant role in predicting build outcomes [33, 4, 40]. Ni and Li [33] discovered that historical statistics on the committer and project were the best predictors of build success. Similarly, Chen et al. [4] developed BuildFast, a history-aware approach that emphasized the importance of historical data in predicting build outcomes. Rausch et al. [38] observed that the strongest influencing factor of the outcome of a build was overall stability in the recent build history, with more than 50% (up to 80%) of failed builds following a previous failure. Hassan and Zhang [19] also used historical data for build certification prediction. By combining project attributes, their model could predict build failure with 69% accuracy and success with 95% accuracy.

Similarly to the previous study, our study also highlights the important role historical data play in understanding build outcomes. Although their study focuses on proposing a predictive model, our study focuses on a statistical model aimed at understanding which characterising features play an important role in differentiating between successful and failed rechecks.

7.3 Flaky test detection.

A common approach to determine if tests are flaky is rerunning them and seeing if the outcome changes, but it can waste CI resources. To address this challenge, researchers have developed methods that use previous test execution histories to detect flaky tests

without relying on reruns. Herzig et al. [20] used historical test execution data, combining features such as test identifiers with pass/fail outcomes, to identify patterns of flakiness through association rule learning. Kowalczyk et al. [24] focused on temporal variations in historical test results, developing a statistical model to measure flakiness by analyzing how test outcomes fluctuate over time. Gruber et al. [14] relied on historical data related to code evolution and test history, including metrics such as the number of files changed in the latest pull request and the frequency of test outcome flips.

In our study, I also use historical data in order to minimize wasted recheck requests and as a result, minimize the waste of CI resources. In contrast, I do not rely solely on flaky test detection, but rather on understanding if recheck would result in success. Additionally, a direct comparison between our approach and previous work would require substantial further analysis, as our predictions are made for the recheck outcome at the build level (i.e., a series of routine steps that may compile, link, assemble, and test code to ensure that the new changes do not introduce regression) whereas prior studies focus on detecting flakiness at the test level.

7.4 CI re-execution.

Durieux et al. [5] conducted an empirical investigation of flaky builds on Travis CI. Their study revealed that out of 56,522 restarted builds, only 46.77% changed their outcome after rerunning failed builds. The study also found that 54.42% of builds were restarted within an hour of the initial build execution, suggesting that developers frequently interrupt their workflow to wait for CI results and address issues. Similarly, Maipradit et al. [29] studied the computational waste caused be repeated builds in the OpenStack community. They found that 55% of code reviews contained at least one recheck request, resulting in substantial computational waste with the cumulative time amounting to 187.4 years. Their findings suggest that recheck requests are often initiated without sufficient investigation into the root causes of build failures.

While recent studies on rerun CI builds focus primarily on their impact, they offer limited insights into how the waste that they produce can be mitigated. Building on these studies, this thesis identifies features that characterize recheck outcomes. Our overarching goal is to provide recommendations about how best to minimize unnecessary recheck requests, such that resource are used efficiency and CI processes are optimized.

Chapter 8

Concluding Remarks

In this thesis, I characterize rechecks in CI. I analyze a dataset of 314,947 recheck from over a 10 year period. I use statistical models to access the impact of bots, jobs, patches, user and timestamp characteristics on recheck outcomes. Below, I recommend directions for future work:

Understanding the impact of human factors on recheck requests. During recheck requests analysis, I find comments that reveal insights into human thoughts and emotions during the recheck process. For example, the recheck requests, "recheck day 2, the monster still hunts me, but I can't give up, I know I need to handle this"[1] and comment, "Shoot, I rechecked against the wrong bug. It should have been bug 1292105".[2] Sentiment analysis of such comments could offer deeper insights into the human factor influencing recheck outcomes. Previous studies on commit messages and self-admitted technical debt (SATD) code comments support this idea. Souza and Silva [43] found a correlation between negative sentiment in commit messages and build outcomes, while Fucci et al. [10] discovered that certain types of SATD (e.g., functional and on-hold SATD) exhibit more negative sentiment compared to others, suggesting a link between human emotion and software development. I hypothesize that a similar relationship may exist between recheck outcomes and the emotions or thoughts expressed in the text preceding recheck requests.

Another potential influencing factor is user expertise, which may lead to overconfidence bias [32]. Experienced users may rely on their own judgment, making recheck decisions based solely on experience without considering relevant data or alternative options. This

[1] https://review.opendev.org/c/openstack/tripleo-ci/+/620063
[2] https://review.opendev.org/c/openstack/tripleo-incubator/+/92749

overconfidence can result in unnecessary rechecks. Additionally, in Agile teams, developers may face increased stress and pressure toward the end of a sprint, leading to wasteful rechecks in the hope of speeding up the build process. Exploring these human factors, e.g., emotions, expertise, and time pressure, can offer new insights into how they impact recheck outcomes, potentially reducing wasteful rechecks.

Analyzing developers perception on automated recheck prediction. Given the predictive power of our model, it is important to evaluate how best to integrate the predictive model into the CI process. One approach could be fully automating the decision, allowing the model to determine whether a failed build should be rechecked without requiring a request from the developer. Alternatively, the model could serve as a decision-support tool, providing suggestions with a rationale when a developer attempts a recheck. Since developers may not always consider the cost of rechecks, another option is to give team leads or managers the authority to approve rechecks, allowing them to weigh the potential costs before proceeding. Each approach offers a different balance between automation and human oversight.

References

[1] Anura. Ai's challenges in fraud detection: Dealing with false positives., 2024. Last accessed on 2025-02-24.

[2] Islem Bouzenia and Michael Pradel. Resource usage and optimization opportunities in workflows of github actions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[3] Kelsy Cabello-Solorzano, Isabela Ortigosa de Araujo, Marco Peña, Luís Correia, and Antonio J. Tallón-Ballesteros. The impact of data normalization on the accuracy of machine learning algorithms: A comparative analysis. In *International Conference on Soft Computing Models in Industrial and Environmental Applications*, pages 344–353. Springer, 2023.

[4] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. Buildfast: History-aware build outcome prediction for fast feedback and reduced cost in continuous integration. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 42–53, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Thomas Durieux, Claire Le Goues, Michael Hilton, and Rui Abreu. Empirical study of restarted and flaky builds on travis ci. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 254–264, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. Understanding flaky tests: the developer's perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 830–840, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Bradley Efron. How biased is the apparent error rate of a prediction rule? *Journal of the American Statistical Association*, 81(394):461–470, 1986.

[8] Joseph G Eisenhauer. *Degrees of Freedom in Statistical Inference.* Springer Berlin Heidelberg, 2011.

[9] Martin Fowler and Matthew Foemmel. Continuous integration, 2006.

[10] Gianmarco Fucci, Nathan Cassee, Fiorella Zampetti, Nicole Novielli, Alexander Serebrenik, and Massimiliano Di Penta. Waiting around or job half-done? sentiment in self-admitted technical debt. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 403–414, 2021.

[11] Keheliya Gallaba, Maxime Lamothe, and Shane McIntosh. Lessons from eight years of operational data from a continuous integration service: an exploratory case study of circleci. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1330–1342, New York, NY, USA, 2022. Association for Computing Machinery.

[12] Taher Ahmed Ghaleb, Daniel Alencar Da Costa, and Ying Zou. An empirical study of the long duration of continuous integration builds. *Empirical Softw. Engg.*, 24(4):2102–2139, aug 2019.

[13] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 345–355, New York, NY, USA, 2014. Association for Computing Machinery.

[14] Martin Gruber, Michael Heine, Norbert Oster, Michael Philippsen, and Gordon Fraser. Practical Flaky Test Prediction using Common Code Evolution and Test History Data . In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 210–221, Los Alamitos, CA, USA, April 2023. IEEE Computer Society.

[15] James A Hanley and Barbara J McNeil. The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology*, 143, 1982.

[16] Frank E Harrell et al. *Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis*, volume 608. 2001.

[17] Frank E Harrell Jr, Kerry L Lee, Robert M Califf, David B Pryor, and Robert A Rosati. Regression modelling strategies for improved prognostic prediction. *Statistics in medicine*, 3, 1984.

[18] Frank E Harrell Jr, Kerry L Lee, David B Matchar, and Thomas A Reichert. Regression models for prognostic prediction: advantages, problems, and suggested solutions. *Cancer treatment reports*, 69, 1985.

[19] Ahmed E. Hassan and Ken Zhang. Using decision trees to predict the certification result of a build. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 189–198, 2006.

[20] Kim Herzig and Nachiappan Nagappan. Empirically detecting false test alarms using association rules. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 39–48, 2015.

[21] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 426–437, New York, NY, USA, 2016. Association for Computing Machinery.

[22] Xianhao Jin and Francisco Servant. A cost-efficient approach to building in continuous integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 13–25, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Xianhao Jin and Francisco Servant. Which builds are really safe to skip? maximizing failure observation for build selection in continuous integration. *J. Syst. Softw.*, 188(C), jun 2022.

[24] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. Modeling and ranking flaky tests at apple. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '20, page 110–119, New York, NY, USA, 2020. Association for Computing Machinery.

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

[26] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. A study on the lifecycle of flaky tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1471–1482, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 643–653, New York, NY, USA, 2014. Association for Computing Machinery.

[28] Hari S Mahesh. Top challenges in ai-driven quality assurance., 2024. LinkedIn post, last accessed on 2025-02-24.

[29] R. Maipradit, D. Wang, P. Thongtanunam, R. Kula, Y. Kamei, and S. McIntosh. Repeated builds during code review: An empirical study of the openstack community. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 153–165, Los Alamitos, CA, USA, sep 2023. IEEE Computer Society.

[30] Shane McIntosh and Yasutaka Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 560, New York, NY, USA, 2018. Association for Computing Machinery.

[31] Shane Mcintosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Softw. Engg.*, 21(5):2146–2189, oct 2016.

[32] Rahul Mohanani, Iflaah Salman, Burak Turhan, Pilar Rodríguez, and Paul Ralph. Cognitive biases in software engineering: A systematic mapping study. *IEEE Transactions on Software Engineering*, 46(12):1318–1339, 2020.

[33] Ansong Ni and Ming Li. Cost-effective build outcome prediction using cascaded classifiers. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 455–458, 2017.

[34] Doriane Olewicki, Mathieu Nayrolles, and Bram Adams. Towards language-independent brown build detection. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2177–2188, New York, NY, USA, 2022. Association for Computing Machinery.

[35] Cong Pan and Michael Pradel. Continuous test suite failure prediction. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 553–565, New York, NY, USA, 2021. Association for Computing Machinery.

[36] Maurizio Pighin and Anna Marzona. An empirical analysis of fault persistence through software releases. In *Proceedings of the 2003 International Symposium on Empirical Software Engineering*, ISESE '03, page 206, USA, 2003. IEEE Computer Society.

[37] Rahul Kalva. The evolution of devsecops with ai., 2024. Last accessed on 2025-02-24.

[38] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, page 345–355. IEEE Press, 2017.

[39] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. On the prediction of continuous integration build failures using search-based software engineering. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, GECCO '20, page 313–314, New York, NY, USA, 2020. Association for Computing Machinery.

[40] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engg.*, 29(1), may 2022.

[41] Takaya Saito and Marc Rehmsmeier. The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets. *PloS one*, 10, 2015.

[42] Tiffany Shoemaker. Top challenges in ai-driven quality assurance., 2024. LinkedIn post comment, last accessed on 2025-02-24.

[43] Rodrigo Souza and Bruno Silva. Sentiment analysis of travis ci builds. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, page 459–462. IEEE Press, 2017.

[44] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904.

[45] Gengyi Sun, Sarra Habchi, and Shane McIntosh. Ravenbuild: Context, relevance, and dependency aware build outcome prediction. *Proc. ACM Softw. Eng.*, 1(FSE), jul 2024.

[46] Xin Tan, Minghui Zhou, and Zeyu Sun. A first look at good first issues on github. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering*

Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, page 398–409, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2017.

[48] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 45(7):683–711, 2019.

[49] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 805–816, New York, NY, USA, 2015. Association for Computing Machinery.

[50] Nimmi Weeraddana, Mahmoud Alfadel, and Shane McIntosh. Characterizing timeout builds in continuous integration. *IEEE Transactions on Software Engineering*, 50(6):1450–1463, 2024.

[51] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: A large-scale empirical study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 60–71, 2017.