Rechecking Recheck Requests in Continuous Integration: An Empirical Study of OpenStack

Yelizaveta Brus*, Rungroj Maipradit*§, Earl T. Barr‡, Shane McIntosh*
*Software REBELs, University of Waterloo, Canada; ‡University College London, UK; §Corresponding author
E-mail: *{ybrus, rungroj.maipradit, shane.mcintosh}@uwaterloo.ca; ‡e.barr@ucl.ac.uk

***Abstract*—Continuous Integration (CI) is a process for automatically checking patch sets for errors. CI periodically fails due to non-deterministic (a.k.a., "flaky") behaviour. Since a patch set may not be the cause of a flaky failure, developers can issue a "recheck" command to request retesting a patch set. Developers waste time considering whether or not to issue a recheck after a CI failure. Prior work also shows that rechecks are issued liberally, wasting up to 187.4 compute years when CI continues to fail. To save developer time and avoid wasteful rechecks, we fit and analyze statistical models that discriminate between *successful* and *failing* rechecks, i.e., those rechecks that will change a failing CI run into a successful one and those that will fail again. Through an empirical study of 314,947 recheck requests from OpenStack, we find that our model can differentiate successful and failed rechecks well, outperforming baseline approaches by 23.6 percentage points in terms of AUROC (0.736).**

Analysis of our model suggests that, in terms of explanatory power, past behaviour of jobs, bots, and users dominate static characteristics of patch sets. Applying our model to automatically request rechecks for those predicted to succeed would have saved roughly 247 years of elapsed developer time for OpenStack. Applying our model to skip recheck requests when they are predicted to fail would avoid 86.49% of wasted rechecks, saving roughly 262 years of compute time.

I. Introduction

Continuous Integration (CI) [1] systems automatically execute routine checks (e.g., compilation, testing) when developers submit change sets for integration. CI systems provide developers with early feedback on code changes, helping to identify mistakes before changes are merged into the main codebase. Prior work has associated CI adoption with accelerated development [2], improved code quality [3], and the adoption of best practices for automated testing [4].

Despite their benefits, CI systems are imperfect, periodically generating an unreliable pass/fail signal [5]. CI runs fail due to infrastructure failures [6], service outages [5], or non-deterministic (a.k.a., "flaky") build behaviour [7]–[9]. When a CI run fails, reports are sent to developers who may attempt to diagnose and fix the issue.

Developers waste time debugging flaky CI results, especially when they attempt to diagnose and fix correct code. When their diagnosis suggests that a failure is unrelated to the code, developers may request the re-execution of a CI run without modifying the patch set. Durieux et al. [10] found that 27,006 of 583,415 Travis CI failures were restarted, while Maipradit et al. [11] found that 55% of OpenStack code reviews included at least one recheck request.

Issuing a recheck request may not resolve the problem. Indeed, 20 of 24 OpenStack contributors surveyed by Maipradit et al. [11] reported that they still debug manually after filing recheck requests. When the root cause is flakiness, this follow-up debugging effort can further waste developer time.

To counteract the waste of developer time, rechecks may be requested without thoroughly scrutinizing CI failures; however, this can generate a substantial waste of compute resources. For example, Maipradit et al. [11] estimated 187.4 years of compute time were wasted by injudicious recheck requests. Durieux et al. [10] further note that 54.42% of restarts occur within an hour, suggesting that many retries may be requested before failures have been fully diagnosed.

In this paper, we characterize recheck outcomes by fitting and analyzing statistical models. Our study uncovers patterns that drive recheck outcomes, offering insights to support decision-making and highlighting factors to prioritize during recheck requests. For our analysis, we examine historical CI data to identify patterns that differentiate successful rechecks from failed ones, focusing on the behaviour of *bots* that react to development events by performing CI, *jobs* that represent concrete tasks (like executing a test or running an analysis tool) that a bot performs, and *users*, who initiate CI requests. We conduct an empirical study of 314,947 recheck requests from the OpenStack community. We structure our study by addressing the following two Research Questions (RQs):

RQ1. How effectively can our model differentiate successful and failed rechecks?

Our model achieves an Area Under the Receiver Operator Characteristic (AUROC) curve of 0.736, surpassing the baseline of random guessing by 23.6 percentage points, demonstrating its strong discriminatory power. The model also achieves a Brier score of 0.191, indicating that its risk estimates are well-calibrated compared to a random guessing model, outperforming the baseline by 5.9 percentage points. In terms of the precision-recall tradeoff, the model achieves an Area Under the Precision Recall Curve (AUPRC) of 0.604, outperforming a random guessing baseline by 25.5 percentage points. The optimism penalties for performance metrics are all near zero, indicating that the model is stable and unlikely to be overfitted. Our 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 job success ratio and bot 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 misbehaving jobs and bots to reduce unnecessary rechecks and optimize resource use.

Inspired by the impact of past behaviour on recheck outcomes, we explore whether 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 remains stable, with AUROC varying only 0.59%. Brier scores improve slightly with larger windows, while AUPRC decreases, suggesting better precision-recall trade-offs for smaller windows. The optimism values remain near zero, indicating low optimism bias. Feature importance for key features like job success ratio and bot success ratio remains consistent. The insensitivity of both model fitness and feature importance suggests that both recent and longer-term data are suitable for generating actionable recommendations.

Our findings may dishearten individual contributors, since they are powerless to change the characteristics of their patch 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. 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).

Our key finding is that classifiers like ours can optimize rechecks. Applying our model to automatically request rechecks for those predicted to succeed would have saved roughly 247 years of elapsed developer time for OpenStack. Applying our model to skip recheck requests when they are predicted to fail would avoid 86.49% of wasted rechecks, saving roughly 262 years of compute time.

II. Core Recheck Concepts

The 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 of its jobs are successful (failed otherwise). Similarly, the outcome of a build is successful if all of 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, we 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 failure can have two possible outcomes. A "failing" recheck is one where the build fails again. Conversely, a "successful" recheck is one where the build eventually succeeds. A "failing" recheck does not provide new information to the developer but consumes additional CI resources. We consider such rechecks wasteful and recommend skipping them to save resources and improve efficiency.

III. Study Design

In this section, we provide our reasons for studying the OpenStack community, and describe our procedures for curating the dataset (DC), fitting the model (MF), and performing model analysis (MA). Figure 1 shows an overview of our study design. Below, we describe each procedure.

OpenStack Community

We select the OpenStack community for our study because it supports recheck functionality, and provides detailed guidelines about its use.[1] Unlike other projects, 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

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

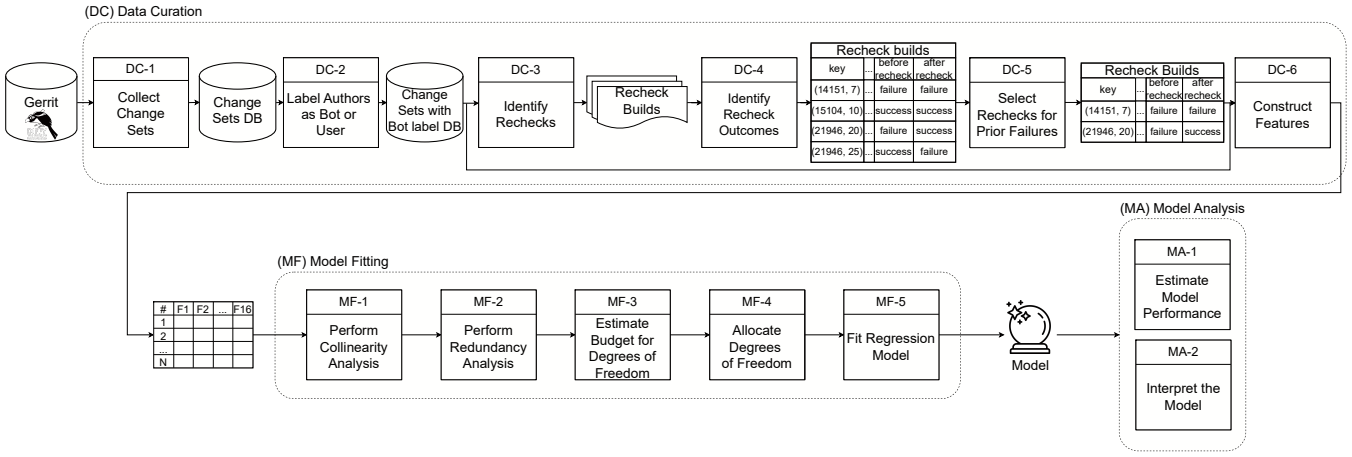

Fig. 1: An overview of our Data Curation (DC), Model Fitting (MF), and Model Analysis (MA) procedures.

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. [11] 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.

It is for these reasons that we 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 that we can collect sufficient data for our analysis.

(DC) Data Curation

In this stage, we collect change sets from projects that are developed by the OpenStack community (DC-1). Then, we categorize authors as bots or users (DC-2) to identify recheck builds (DC-3) and extract recheck outcomes (DC-4). Finally, we filter the data to focus on rechecks initiated after a failing build (DC-5) and compute features that we suspect may correlate with the likelihood of a recheck build passing (DC-6). We describe each step below. We also make our dataset and the replication package available online.[3]

(DC-1) Collect Change Sets. Our prior work [11] studied repeated builds in the OpenStack community. Following a similar procedure [11], we collect change sets and user information from projects developed by the OpenStack community using the RESTful API of the Gerrit code review tool.[4] To avoid analyzing unfinished or abandoned work, we select the closed change sets that were merged into the "master" branch of each project. We also collect metadata describing the owner, reviewers, patch, and repository name. The previous version of our dataset [11] included 66,932 change sets from four selected projects. The version of the dataset that we now study 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. We study the historical activity that was recorded between May 1, 2014 and April 30, 2024.

(DC-2) Label Authors as Bot or User. To check whether the author is a bot or a user, we detect authors that have bot characteristics. Among those authors, we manually evaluate their comments. We 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, we narrowed the list for manual verification to 7.29% (1,082) of all authors (14,838). During manual verification, the first author inspects 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, we classify them as a bot. The outcome of the assessment of the first author can also be uncertain, in which case, the second author inspects the result. In total, we detect 328 (2.21%) bot accounts and 14,510 (97.79%) user accounts.

(DC-3) Identify Rechecks. To identify rechecks, we search for user-initiated comments that contain the term "recheck". This query produces a set of 487,131 comments. We consider the comment to be issuing a recheck request if it receives a response from a bot. We find that 412,518 of the 487,131 comments issue recheck requests.

(DC-4) Identify Recheck Outcomes. To identify the status of a recheck request, we 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

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

[3]https://doi.org/10.5281/zenodo.13755309

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

TABLE I: Definitions for and rationale of the selected features that we use to model the likelihood of recheck success.

Family	Feature	Description	Rationale
Bot history	• bot 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	• job 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 • deletions • file numbers	The total number of inserted lines of code. The total number of deleted lines of code. The number of unique files touched.	The size of the change corresponding to the build may have some relation to the outcome of the build [12], [13].
	• 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 • recheck day • recheck hour • recheck minute	The month when the recheck was issued. The day of the week when the recheck was issued. The hour when the recheck was issued. The minute when the recheck was issued.	Recheck requests made at specific times may have a higher or lower likelihood of resulting in success [14].
User	• user 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 hypothesize that a higher ratio increases the likelihood of success.
	• user status	The position of the user within the team (owner, reviewer, or none).	We assume that a higher status raises the chance of successful rechecks [15].
	• experience	The number of messages produced by the user, including discussions, code reviews, comments, etc.	We assume that experienced users have a higher chance of a successful build.

if the results of all re-invoked bots are successful; otherwise, it is considered failed.

We use a regular expression to identify the bot result. If the bot result is not available, we summarize the outcome of its job results using regular expressions. Note that jobs may be *non-voting*. The outcomes of non-voting jobs do not affect the outcome of a bot. Hence, we exclude non-voting jobs when determining bot outcomes, only indicating that a bot has failed if at least one of its *voting* jobs fails.

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

(DC-6) Construct Features. Our features are inspired by previous studies [14] of build outcome and defect prediction. As a result, we formulate 16 features, which are explained in detail in Table I along with their feature families.[5] 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 we use UTC timestamps. We compute the feature values using the database of change sets.

We 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 [16]. 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.

(MF) Model Fitting

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

(MF-1) Perform Collinearity Analysis. Collinear features can distort each other's importance, and interfere with each other when fitting the model [15], [17]. To address this, we apply Spearman's rank correlation (ρ) [18] due to its ability to detect nonlinear correlation compared to other types of correlation measures (e.g., Pearson). Similar to previous studies, we set

[5]We only consider the flip ratio for bots because 15.42% (54,534) of the values for the flip ratio of jobs cannot be computed due to division by zero.

the threshold as $\rho = 0.7$ [14], [19]–[21], i.e., for any pair of features with $\rho > 0.7$, we select only one feature to include in our model fit.

The hierarchical overview of the correlations among the features is shown in Figure MF1.1 in our online appendix.[3] From the pairs with $\rho > 0.7$, we select "lines inserted" and "bot success ratio", therefore "number of unique files touched" and "bot flip success ratio" are removed.

(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, we 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. [22]. In our analysis, none of the 14 features exceed the threshold.

(MF-3) Estimate Budget for Degrees of Freedom. To capture non-monotonic or nonlinear associations, we allocate additional degrees of freedom to features [23]. A feature with a single Degree of Freedom (DoF) can only represent linear monotonic relationships with the likelihood of a successful recheck. Allocating additional degrees of freedom allows the model to account for more complex relationships between features and the build outcome; however, allocating too many degrees of freedom can lead to overfitting, where the model becomes overly specific to the training data [24].

To balance the trade-off between model complexity and the risk of overfitting, a DoF budget can be established [25], [26]. The purpose of a DoF budget is to limit the total number of degrees of freedom 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 using a common rule of thumb [25], given in Equation (1), which allocates 15 samples per degree of freedom to mitigate overfitting:

$$budget_{DoF} = \frac{n}{15} \tag{1}$$

where n is the number of records in the minority class [25].

(MF-4) Allocate Degrees of Freedom. The relationship between features and the likelihood of a successful recheck based on Spearman's multiple ρ^2 is shown in Figure MF4.1 in our online appendix.[3] We select features that have higher ρ^2 values than others when allocating additional degrees of freedom. In our case, we conservatively assign three degrees of freedom to the six features with the highest ρ^2 values (i.e., "user success ratio', "build success ratio", "job success ratio", "file success ratio", "diff avg success" and "diff avg failure").

(MF-5) Fit Regression Model. We use traditional statistical analysis rather than machine learning to fit the models that we analyze in this paper because we prioritize interpretability and explainability. Indeed, our primary aim is to derive insights from features and their influence. Moreover, our online appendix includes a comparison with four traditional machine learning classifiers, suggesting that they achieve similar fitness scores.[3]

We fit our regression model by selecting the relevant explanatory features and strategically allocating the degrees of freedom. We then fit the model to our data using restricted cubic splines [24] 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 relax the linearity assumption between features and outcome, allowing for the relation to evolve in complex ways spanning the range of variable values. Restricted cubic splines retain straight tails, which tends to improve fitness at lower and upper extremes where a purely cubic curve would tend to curl away from observed values.

(MA) Model Analysis

In this stage, we analyze the model performance (MA-1) and interpret the model (MA-2). We describe each step below.

(MA-1) Estimate Model Performance. We assess the performance of our logistic regression model according to its discriminatory power, the reliability of its risk predictions, and its capacity to balance the precision-recall trade-off.

We 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 [22].

To evaluate precision and recall, we 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 [27]. The AUPRC is between 0 and 1, with higher values indicating better performance. We compare our AUPRC to a baseline determined by the positive class prevalence. The baseline is calculated using Equation (2):

$$AUPRC_{baseline} = \frac{tp}{tp + tn} \tag{2}$$

where tp and tn are the numbers of true positives and negatives, respectively [27]. For a balanced class distribution, $AUPRC_{baseline} = 0.5$.

To evaluate the reliability of the predicted probabilities of the model, we use the Brier score, which 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.

(MA-2) Interpret the Model. Similar to our prior work [14], [15], to interpret the models that we fit, we study its features using Wald χ^2 maximum likelihood tests (a.k.a., "chunk"), and

plot the response curve of each feature with respect to the likelihood of a recheck being successful.

Wald χ^2 maximum likelihood tests help us to 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. We also test the statistical significance of the contribution of features using the p-value.

Additionally, we 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.

IV. Overall Performance

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

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

Before fitting the model, we compute features. During this process, we encounter some entries for the first time, such as a patch introducing a new file, a user submitting their first recheck request, or the appearance of a new bot or job. In such cases, we 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, we explore replacing them with 0 (i.e., the minimum replacement strategy), 1 (i.e., the maximum replacement strategy), or removing them entirely. The model is fitted for each strategy. We observe that all three strategies achieve similar AUROC (maximum difference of 2.1 percentage points) and Brier score values (all values within one percentage point).[6] The maximum replacement strategy has an AUPRC that is 5.9 and 7.3 percentage points lower than the minimum replacement and remove N/A strategies, respectively. Hence, we continue using the strategy that removes observations with N/A values. We reason that removal is a more appropriate choice because an N/A value indicates that a feature has not yet been observed, and replacement would create data that does not exist.

After dropping observations with N/A values, 314,947 observations remain, with 109,825 belonging to the minority class (i.e., successful rechecks). Based on Equation (1), we can allocate up to $\frac{109{,}825}{15} = 7,321$ degrees of freedom without raising concerns about overfitting. Using Equation (2), we estimate the baseline to which the AUPRC of our models should be compared to be $\frac{109{,}825}{314{,}947} = 0.349$.

To assess our model, we use AUROC, AUPRC, and Brier score along with assessing the stability of the model fit using *bootstrap-calculated optimism* [28]—a robust model validation technique akin to k-fold cross-validation. First, we draw a bootstrap sample of length 314,947 from our original dataset with replacement. Next, we refit the logistic regression model to this sample, using the same degrees of freedom. We 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. We repeat this process 1,000 times and calculate the mean optimism values. Smaller optimism values shows a model fit is more robust to natural fluctuations in the shape of the dataset.

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

TABLE II: Performance metrics for traditional ML models.

ML models	AUROC	Brier Score	AUPRC
Statistical model	0.736	0.191	0.604
Logistic regression	0.733	0.192	0.572
Random Forest	0.731	0.190	0.583
XGBoost	0.725	0.193	0.581
SVM	0.729	0.194	0.562

In direct response to peer review feedback, we expanded our evaluation to include specific machine learning models (i.e., Logistic Regression, SVM, XGBoost, and Random Forest) as advised. Table II shows the evaluation of traditional machine learning methods using time-based 10-fold cross-validation, avoiding single-repetition holdout due to its bias and variance issues [29]. Time-based 10-fold cross-validation sequentially splits data into ten parts, training on earlier data and testing on unseen later data. Compared to our bootstrap-calculated optimism scores for logistic regression, AUROC and Brier Score differ by less than one percentage point, while AUPRC drops by 3.2 points at most. We also find that the machine learning methods all achieve performance values that vary by two percentage points at most. As different learning approaches do not substantially outperform statistical regression, we focus on the regression results in the paper.

To estimate developer impact, we calculate the elapsed developer time between receiving a build result and initiating a recheck, acknowledging that this likely overestimates actual decision time, and interpret it as an upper bound on time savings. We also examine the trade-off between compute and elapsed developer time savings. Compute time is measured by summing job durations from bot build messages. Developer time is saved when the model correctly predicts a successful recheck, preventing unnecessary manual rechecks. Compute time is saved when failing rechecks are accurately predicted, avoiding wasted job executions.

To balance automation with prediction certainty, we apply a hybrid thresholding strategy. In our evaluation, we set the lower certainty threshold to 0.2 and the upper certainty threshold to 0.7 as an example. Predictions with probabilities below 0.2 are automatically treated as failures, while those above 0.7 are treated as successes. Predictions within the intermediate range [0.2, 0.7] are left for manual developer intervention. This design enables the system to automate decisions when the

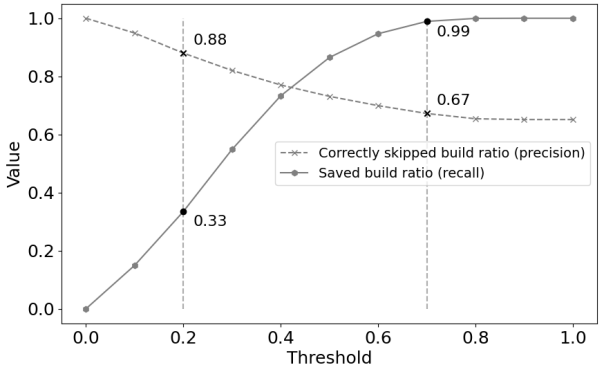

Fig. 2: In this figure, the threshold balances correctly identifying failing rechecks (precision) against identifying them all (recall). At the threshold 0.2, for instance, 88% of the rechecks that we skip failed, and we skip 33% of all failed rechecks.

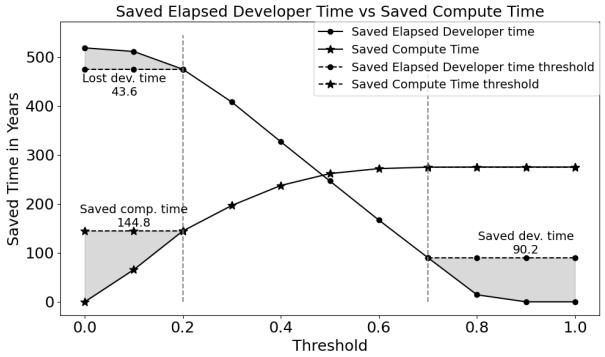

Fig. 3: In this figure, the threshold balances saved elapsed developer time with saved compute time. The shaded areas show how much time can be saved by automatically handling the most certain cases, while letting developers decide in situations where the model is less sure.

model has high confidence while preserving human oversight for uncertain cases. The selected thresholds serve as an example; they can be adjusted to reflect different organizational trade-offs between developer time savings and compute resource utilization. Figure 3 plots saved elapsed developer and compute time as model thresholds vary.

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 our model can differentiate between successful and failed rechecks. Furthermore, our model achieves a Brier score of 0.191. Since random guessing would achieve a Brier score of 0.25, our model achieves 5.9 percentage points or 23.6% better calibration of risk estimates. Our model also 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 highlights 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) outnumber the negative ones.

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 [25]. 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 2 shows the threshold effects on the balance between saving 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 of the 205,122 recheck failures), saving substantial build resources. For cases where the priority is to minimize developer distractions (i.e., by prioritizing precision in identifying successful rechecks), 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.

Observation 4: The model could save roughly 247 years of elapsed developer time and roughly 262 years of compute time. Figure 3 shows how varying the prediction threshold affects the trade-off between saved elapsed developer and compute time. Using the default threshold of 0.5, the model can save roughly 247 years of elapsed developer time and roughly 262 years of compute time. We also apply a threshold strategy that automates decisions at both low and high extremes, i.e., when the model is highly confident. When the lower threshold is set to 0.2, the model saves roughly 145 years of compute time. When the upper threshold is set to 0.7, the model saves roughly 90 years of developer time at the cost of only 0.3 years of compute.

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

We analyze the importance of individual features and feature families using Wald χ^2 maximum likelihood tests. We also analyze response curves for the two features that account for 50% of the explanatory power of our model. The response curves for all features are available in our online appendix.[3]

Table III 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 we 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

TABLE III: 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	job 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	bot 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	file 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

shown in the "Nonlinear" column if no additional degrees of freedom are allocated, i.e., the feature fit is linear.

To assess the stability of feature ranking, we repeat Wald χ^2 maximum likelihood tests 20 times, each time using 90% of the data selected at random. Table IV presents the range of Wald χ^2 values for each individual feature in our model.

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 other features, resulting in a family χ^2 value that differs from the sum of that of its features.

Along with Wald χ^2 values, we 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. We drop features with a small impact on points to optimize vertical space, though a complete version is available in our replication package.[3]

Observation 5. All families contribute at least one significant feature to the model fit. Table III shows all families are significant and contribute meaningfully to the overall fit of the model, which supports our hypotheses about each family (see Table I). Overall, "job history" and "bot history"

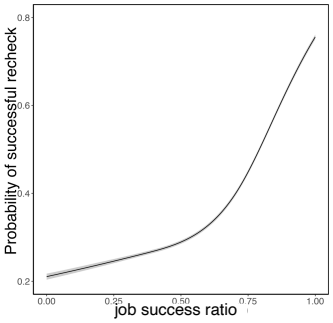

Fig. 4: Job success ratio vs. successful recheck prob.

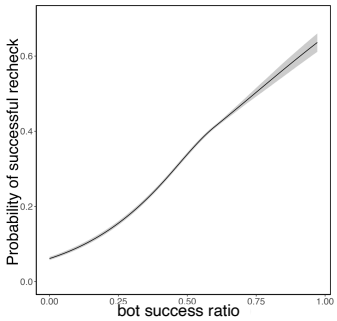

Fig. 5: Bot success ratio vs. successful recheck prob.

families together account for 60.19% ($\frac{16{,}247.20+7{,}866.04}{40{,}060.51}$) of the explanatory power of the model.

Table III also 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.

Observation 6. Past behaviour is the dominant explanatory factor. Table III 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 [30], 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 "file 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. Feature importance results are stable across bootstrap iterations. Table IV shows that the top six features ("job success ratio", "bot success ratio", "diff avg failure", "user success ratio", "file success ratio", and "user status") consistently remained in the top positions across all runs in the same order. Minor shuffling occurred only among low-ranked features, which contribute little to the overall explanatory power. These low variances and the unchanged set of top features confirm that our conclusions about the dominant factors driving recheck success are robust to natural fluctuations in the dataset.

Observation 7. Features related to bots and jobs are the strongest in explaining the likelihood of a successful recheck. Features closely linked to the build process, specifically the "job success ratio" and "bot success ratio", account for 50% of the explanatory power of our model. Notably, the "job success ratio" contributes significantly to this, accounting for 40.56%

Variable	Initial χ^2	$\sigma(\chi^2)$	Initial Rank	Min Rank	Max Rank	Rank Range
job success ratio	16,247.20	64.73	1	1	1	0
bot success ratio	3,789.65	30.11	2	2	2	0
diff avg failure	2,089.11	20.40	3	3	3	0
user success ratio	1,592.01	23.95	4	4	4	0
file success ratio	1,320.56	24.80	5	5	5	0
user status	91.23	4.72	6	6	6	0
diff avg success	52.22	5.03	7	7	8	1
queued hour	50.55	3.25	8	7	8	1
user experience	15.40	2.63	9	9	9	0
queued month	2.09	0.91	10	10	11	1
queued day	1.81	0.96	11	10	13	3
lines inserted	0.38	0.26	12	11	14	3
lines deleted	0.12	0.20	13	12	14	2
queued minute	0.07	0.33	14	11	14	3

TABLE IV: Variable ranking results with initial χ^2, standard deviation, and rank ranges.

of the total explanatory power. We further analyze these two features using response curves.

Figure 4 presents the response curve, showing a strong relationship between the "job success ratio" and the probability of a successful recheck, with narrow confidence intervals. Notably, the response curve shows an exponential increase, particularly after the "job 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 "job success ratio" is below 0.6, and the ratio drops to 1.34 when the "job success ratio" exceeds 0.6. This shows how past behaviour (i.e., past successes or failures) strongly influences recheck outcomes.

The "bot success ratio" follows a similar pattern, highlighting the importance of past behaviour. Figure 5 shows a strong relationship between the "bot 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 few data points with a "bot success ratio" higher than 0.62, similar to the entire dataset, where only 1.6% of data points exceed a "bot success ratio" of 0.62.

Observation 8. The "diff avg failure" feature becomes a deciding factor when values are at its extremes. Figure 6 shows that even though the "diff avg failure" feature is ranked third by importance, it becomes a deciding factor when values are at its extremes. As the "diff avg failure" feature value increases, the nomogram shows a nonlinear relationship, with a larger gap between values above zero. The difference in 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.

V. Analysis

In Section IV, we observe that past behaviour features are highly important in determining recheck outcomes. These past behaviours have been measured using ten years of data. Since more recent trends may be outweighed by the bulk of historical data, in this section, we set out to study how sensitive past

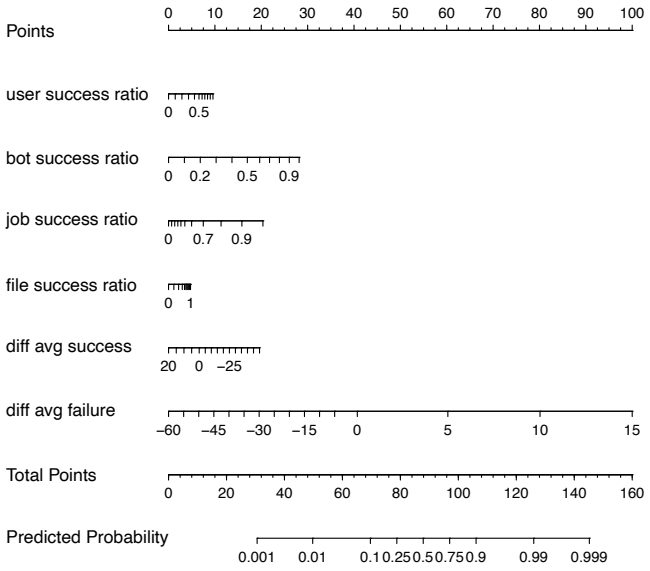

Fig. 6: A nomogram without features with a small impact.

behaviour features are to the *window size*, i.e., the amount of time considered when computing the measures. We refit our recheck outcome model using window sizes of one day up to one year. During feature extraction, we calculate feature values based on data available within a given window. For example, if the window size is set to one day, we use data from the day before a recheck request to calculate features. To ensure that the features of each observation have been recomputed using the full span of the window size of data, we 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, we 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, we rank features based on their Wald χ^2 importance scores using the non-parametric Scott-Knott Effect Size Difference (ESD) test [31], as it does not assume normality or homogeneity of distributions. For each window size, we 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 9: Our model fitness is not sensitive to window size. The model performance across the studied window sizes is shown in Table 5.1 in the online appendix.[3] AUROC varies minimally across window sizes (e.g., 0.74243 for one day vs. 0.73806 for two weeks), suggesting that window size has little impact on the discriminatory power of the model. Brier scores similarly have a small difference, showing negligible variation. In contrast, the AUPRC decreases with larger window sizes, with the highest at 0.64046 for one

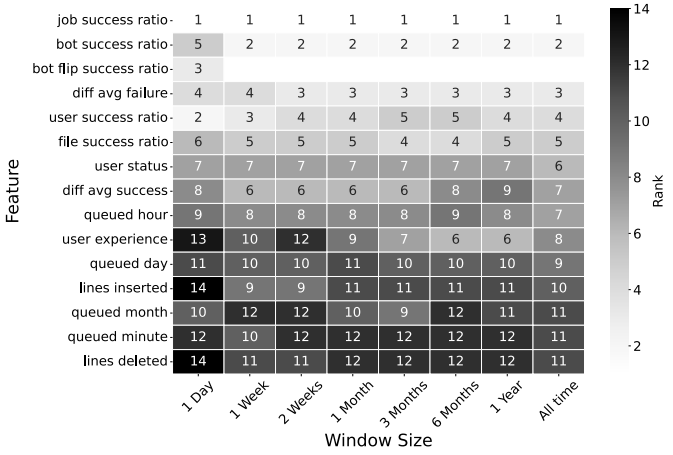

Fig. 7: Statistically distinct feature ranks based on the Scott-Knott ESD test applied to the Wald χ^2 estimates of explanatory power. Top ranked feature families (e.g., success ratios) quickly climb to and hold top ranks consistently across window sizes.

day, suggesting better precision-recall trade-offs with smaller windows. We suspect that AUPRC increases with smaller window sizes are driven by a higher ratio of successful rechecks, as smaller window sizes exclude more N/A failures, raising the success ratio. Optimism values for AUROC, Brier score, and AUPRC range from -0.00005 to 0.00032, indicating stable fits across the studied window sizes.

Observation 10: Feature importance is not sensitive to window size. Figure 7 shows that the "job success ratio", "bot success ratio", "diff avg failure", "user success ratio", "file 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 "bot 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.

We also analyze response curves for the most important feature (i.e., "job success ratio") across window sizes and observe no substantial 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, we provide response curves for the "job success ratio" for each window size in the online appendix.[3]

VI. Threats to Validity

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

Internal Validity. We identify the recheck requests based on the author of the comment being a human user, and not a bot. We may miss recheck requests if we mistake a human for a bot. To mitigate this, the first author manually labels authors as humans or bots, and the second author inspects its results. We 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.

External Validity. As our study focuses on the OpenStack community, it may not generalize to open-source communities or technologies other than Gerrit. We 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.

VII. Related Work

Waste in CI. Prior work has proposed areas of improvement for the CI user experience. Gallaba et al. [5] analyzed CircleCI builds, observing that non-signal-generating failures (i.e., failures that do not indicate quality concerns with the change under scrutiny) may occur due to misconfigurations or a lack of service availability. Weeraddana et al. [14] studied timeout builds in CI systems, observing their strong tendency to occur in clusters. Bouzenia and Pradel [32] analyzed GitHub Actions workflows, finding that 91.2% of CI/CD resource usage is tied to testing and building. While existing optimizations like caching help, their adoption remains limited.

Broadly, these studies show that build failures, timeouts, and their associated costs negatively affect CI operations and user experience. Building on them, this paper shows that, if a model such as ours were adopted, 86.49% of the CI waste generated by failing rechecks could be avoided.

Build outcome prediction. Several methods have been proposed to predict build outcomes, with the goal of skipping unnecessary builds. Jin and Servant [33], [34] proposed SmartBuildSkip and PreciseBuildSkip, which skip builds predicted as successful and execute ones predicted as failures. Kwan et al. [35] studied the effects of socio-technical congruence on build outcomes, observing complex relationships. Saidani et al. [36] applied evolutionary search algorithms to predict build failures using historical data and code metrics. Pan and Pradel [12] proposed a model to predict test suite failures on a continuous basis, while Sun et al. [37] introduced the Raven-Build model, which integrates contextual and dependency-aware features to predict CI build outcomes.

Previous studies found that historical data played a significant role in predicting build outcomes [36], [38], [39]. Ni and Li [38] identified committer and project history as key predictors. Similarly, Chen et al. [39] developed BuildFast—a history-aware approach to predict build outcomes. Rausch et al. [40] linked overall stability in the recent build history to outcomes. Hassan and Zhang [41] combined project attributes to predict success with 95% accuracy.

Like prior work, our study highlights the important role historical data plays in understanding build outcomes. Just as build outcome prediction needed to adapt from a regime in which build requests were infrequent (e.g., daily or even less frequent [41]) to modern continuous settings [33], our application scenario—minimizing wasted CI resources and developer time on failing rechecks—presents new challenges for build outcome prediction.

Test case selection and prioritization. Prior work introduced test case prioritizing methods to rank test cases by their tendency to lead to build failures, and executed the test cases in the order of highest tendency to the least [42], [43]. Elbaum et al. [44] focused on deciding which test cases to include in a CI job and in what order they should be executed. For the same task, Ling et al. [45] compared how test case prioritization works in open-source and closed-source projects.

Our work focuses on a different part of the CI process, which is deciding whether to rerun a failed build (i.e., a set of jobs) at all. Instead of focusing on which tests to run or in what order, we study whether a recheck request is likely to succeed. This question has not been addressed in the papers mentioned and adds a new perspective to CI research.

Flaky test detection. To reduce the need for costly reruns to determine if tests are flaky, researchers have developed flaky test detection methods based on historical test execution data. Herzig et al. [46] 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. [47] analyzed temporal variations in test results to detect flakiness statistically. Gruber et al. [48] relied on historical data related to code evolution and test history, incorporating code evolution and pull request metrics.

Unlike these studies, which focus on test-level flakiness, this paper processes recheck requests at the build level, where multiple factors beyond individual test behaviour may influence the outcome. This difference makes a direct comparison between our approach and prior work impractical and inappropriate.

Repeated execution of CI. Durieux et al. [10] studied flaky builds on Travis CI, observing that only 46.77% of 56,522 re-executed builds changed failing builds into passing ones (or vice versa). In prior work [11], we observed that 55% of OpenStack code reviews included at least one recheck request, generating 187.4 years of computational waste.

Recent studies on the repeated execution of CI builds focus primarily on their impact, offering limited insights into how to mitigate the waste that they produce. Our work expands this line of research by identifying predictive features that can determine when recheck requests are likely to succeed (or continue to fail), enabling more efficient CI operations by avoiding developer interruptions for successful rechecks and saving CI resources for unnecessary rechecks.

VIII. Concluding Remarks

In this paper, we characterize rechecks in CI. We analyze a dataset of 314,947 rechecks spanning a ten-year period. We use statistical models to access the impact of bots, jobs, patches, user, and timestamp characteristics on recheck outcomes. Building on these findings, we outline how our model could be used to enforce dynamic penalties and rewards to mitigate misbehaviour and incentivize good practice. For instance, bots whose rechecks our model predicts will fail more than a threshold could have their jobs or rechecks rate-limited or by lowering their status from mandatory to optional. Conversely, users or bots that consistently issue effective rechecks could be rewarded through preferential treatment or increased execution priority. Below, we recommend directions for future work:

Understanding the impact of human factors on recheck requests. During our analysis of recheck requests, we 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"[7] and comment, "Shoot, I rechecked against the wrong bug. It should have been bug 1292105".[8] 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 [49] found a correlation between negative sentiment in commit messages and build outcomes, while Fucci et al. [50] 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. We hypothesize that a similar relationship may exist for recheck outcomes.

Another influencing factor is user expertise, as experienced developers may show overconfidence bias [51], relying on personal judgment over data, potentially triggering unnecessary rechecks. Agile team pressures, especially near sprint deadlines, can also drive wasteful rechecks in an attempt to speed up the integration process. Investigating emotions, expertise, and time pressure may help to further reduce unnecessary rechecks and optimize CI efficiency.

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.

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

References

[1] M. Fowler and M. Foemmel, "Continuous integration," 2006.

[2] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. 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, (New York, NY, USA), p. 426–437, Association for Computing Machinery, 2016.

[3] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. 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, (New York, NY, USA), p. 805–816, Association for Computing Machinery, 2015.

[4] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. 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)*, pp. 60–71, 2017.

[5] K. Gallaba, M. Lamothe, and S. 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, (New York, NY, USA), p. 1330–1342, Association for Computing Machinery, 2022.

[6] D. Olewicki, M. Nayrolles, and B. Adams, "Towards language-independent brown build detection," in *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, (New York, NY, USA), p. 2177–2188, Association for Computing Machinery, 2022.

[7] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), p. 643–653, Association for Computing Machinery, 2014.

[8] M. Eck, F. Palomba, M. Castelluccio, and A. 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, (New York, NY, USA), p. 830–840, Association for Computing Machinery, 2019.

[9] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, (New York, NY, USA), p. 1471–1482, Association for Computing Machinery, 2020.

[10] T. Durieux, C. Le Goues, M. Hilton, and R. Abreu, "Empirical study of restarted and flaky builds on travis ci," in *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, (New York, NY, USA), p. 254–264, Association for Computing Machinery, 2020.

[11] 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)*, (Los Alamitos, CA, USA), pp. 153–165, IEEE Computer Society, sep 2023.

[12] C. Pan and M. Pradel, "Continuous test suite failure prediction," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, (New York, NY, USA), p. 553–565, Association for Computing Machinery, 2021.

[13] I. Saidani, A. Ouni, M. Chouchen, and M. W. 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, (New York, NY, USA), p. 313–314, Association for Computing Machinery, 2020.

[14] N. Weeraddana, M. Alfadel, and S. McIntosh, "Characterizing timeout builds in continuous integration," *IEEE Transactions on Software Engineering*, vol. 50, no. 6, pp. 1450–1463, 2024.

[15] S. Mcintosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Softw. Engg.*, vol. 21, p. 2146–2189, oct 2016.

[16] K. Cabello-Solorzano, I. Ortigosa de Araujo, M. Peña, L. Correia, and A. 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*, pp. 344–353, Springer, 2023.

[17] S. McIntosh and Y. 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, (New York, NY, USA), p. 560, Association for Computing Machinery, 2018.

[18] C. Spearman, "The proof and measurement of association between two things," *The American Journal of Psychology*, vol. 15, no. 1, pp. 72–101, 1904.

[19] T. A. Ghaleb, D. A. Da Costa, and Y. Zou, "An empirical study of the long duration of continuous integration builds," *Empirical Softw. Engg.*, vol. 24, p. 2102–2139, aug 2019.

[20] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), p. 345–355, Association for Computing Machinery, 2014.

[21] X. Tan, M. Zhou, and Z. 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, (New York, NY, USA), p. 398–409, Association for Computing Machinery, 2020.

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

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

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

[25] F. E. Harrell Jr, K. L. Lee, R. M. Califf, D. B. Pryor, and R. A. Rosati, "Regression modelling strategies for improved prognostic prediction," *Statistics in medicine*, vol. 3, 1984.

[26] F. E. Harrell Jr, K. L. Lee, D. B. Matchar, and T. A. Reichert, "Regression models for prognostic prediction: advantages, problems, and suggested solutions.," *Cancer treatment reports*, vol. 69, 1985.

[27] T. Saito and M. Rehmsmeier, "The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets," *PloS one*, vol. 10, 2015.

[28] B. Efron, "How biased is the apparent error rate of a prediction rule?," *Journal of the American Statistical Association*, vol. 81, no. 394, pp. 461–470, 1986.

[29] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, 2017.

[30] M. Pighin and A. Marzona, "An empirical analysis of fault persistence through software releases," in *Proceedings of the 2003 International Symposium on Empirical Software Engineering*, ISESE '03, (USA), p. 206, IEEE Computer Society, 2003.

[31] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 683–711, 2019.

[32] I. Bouzenia and M. 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), Association for Computing Machinery, 2024.

[33] X. Jin and F. Servant, "A cost-efficient approach to building in continuous integration," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, (New York, NY, USA), p. 13–25, Association for Computing Machinery, 2020.

[34] X. Jin and F. Servant, "Which builds are really safe to skip? maximizing failure observation for build selection in continuous integration," *J. Syst. Softw.*, vol. 188, jun 2022.

[35] I. Kwan, A. Schroter, and D. Damian, "Does socio-technical congruence have an effect on software build success? a study of coordination in a software project," vol. 37, p. 307–324, may 2011.

[36] I. Saidani, A. Ouni, and M. W. Mkaouer, "Improving the prediction of continuous integration build failures using deep learning," *Automated Software Engg.*, vol. 29, may 2022.

[37] G. Sun, S. Habchi, and S. McIntosh, "Ravenbuild: Context, relevance, and dependency aware build outcome prediction," *Proc. ACM Softw. Eng.*, vol. 1, jul 2024.

[38] A. Ni and M. Li, "Cost-effective build outcome prediction using cascaded classifiers," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 455–458, 2017.

[39] B. Chen, L. Chen, C. Zhang, and X. 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, (New York, NY, USA), p. 42–53, Association for Computing Machinery, 2021.

[40] T. Rausch, W. Hummer, P. Leitner, and S. 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, p. 345–355, IEEE Press, 2017.

[41] A. E. Hassan and K. Zhang, "Using decision trees to predict the certification result of a build," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pp. 189–198, 2006.

[42] M. Bagherzadeh, N. Kahani, and L. Briand, "Reinforcement learning for test case prioritization," *IEEE Trans. Softw. Eng.*, vol. 48, p. 2836–2856, Aug. 2022.

[43] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Softw. Engg.*, vol. 19, p. 182–212, Feb. 2014.

[44] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), p. 235–245, Association for Computing Machinery, 2014.

[45] X. Ling, R. Agrawal, and T. Menzies, "How different is test case prioritization for open and closed source projects?," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2526–2540, 2022.

[46] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 39–48, 2015.

[47] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. 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, (New York, NY, USA), p. 110–119, Association for Computing Machinery, 2020.

[48] M. Gruber, M. Heine, N. Oster, M. Philippsen, and G. Fraser, " Practical Flaky Test Prediction using Common Code Evolution and Test History Data ," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, (Los Alamitos, CA, USA), pp. 210–221, IEEE Computer Society, Apr. 2023.

[49] R. Souza and B. Silva, "Sentiment analysis of travis ci builds," in *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, p. 459–462, IEEE Press, 2017.

[50] G. Fucci, N. Cassee, F. Zampetti, N. Novielli, A. Serebrenik, and M. 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)*, pp. 403–414, 2021.

[51] R. Mohanani, I. Salman, B. Turhan, P. Rodríguez, and P. Ralph, "Cognitive biases in software engineering: A systematic mapping study," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1318–1339, 2020.