TU Delft

Delft University of Technology

Document Version
Final published version

Licence
Dutch Copyright Act (Article 25fa)

Citation (APA)
Özdemir, Ö., Aydoğan, R., & Sözer, H. (2026). On the use of machine learning for failure prediction after collective changes in automated continuous integration testing. *Journal of Systems and Software*, *236*, Article 112791. https://doi.org/10.1016/j.jss.2026.112791

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
In case the licence states “Dutch Copyright Act (Article 25fa)”, this publication was made available Green Open Access via the TU Delft Institutional Repository pursuant to Dutch Copyright Act (Article 25fa, the Taverne amendment). This provision does not affect copyright ownership.
Unless copyright is transferred by contract or statute, it remains with the copyright holder.

Sharing and reuse
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.

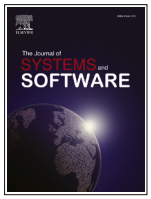
The Journal of Systems and Software 236 (2026) 112791



Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss



On the use of machine learning for failure prediction after collective changes in automated continuous integration testing

Ömer Özdemir [a], Reyhan Aydoğan [b,c], Hasan Sözer [b,*]

[a] Vestel Elektronik A.Ş., Manisa, 45030, Türkiye
[b] Ozyegin University, Department of Computer Science, İstanbul, 34794, Türkiye
[c] Delft University of Technology, Interactive Intelligence Group, Delft, 2628 CD, The Netherlands

ARTICLE INFO

Editor: Dr T. Mårtensson

Keywords:
Failure prediction
Continuous integration
Collective changes
Machine learning
Build process
Industrial case study

ABSTRACT

Continuous Integration (CI) is a development practice where developers regularly merge their code changes into a central repository, enabling simultaneous collaboration across a shared codebase. This frequent integration and automated building process in CI helps to detect and resolve conflicts or errors early in development. However, in large-scale systems, the build process can be costly. Each build incurs expenses, while skipping builds can increase the risk of undetected failures. Accurate predictions can help to identify builds that can be safely skipped to reduce CI costs. This paper presents an empirical study within an industrial setting, investigating the use of machine learning techniques to predict build failures after a set of collective changes. Unlike many existing works that apply random data splitting, our results show that chronological (time-based) splitting offers a more realistic and reliable assessment of model performance in CI environments. We evaluate various models and feature combinations on a dataset derived from real-world industrial projects. We observe high precision but low recall in predicting failed builds, allowing hundreds of successful builds to be correctly skipped, with around a dozen failures potentially being missed. Our analysis shows that this yields substantial time savings of approximately 2.5 h per build on average, while missed failures necessarily result in delayed failure detection, whose practical impact depends on application criticality and operational context.

1. Introduction

Industrial software development companies often prefer Continuous Integration (CI) systems to speed up delivery times (Fitzgerald and Stol, 2017; Donca et al., 2022; Singh, 2023). These systems automate the build and test processes as code changes made by users are integrated into a common repository. This automation makes it easier to detect integration issues at an early stage, reducing the risk of errors in the final product. In particular, nightly builds are widely used, allowing extensive automated testing to be run throughout the night. These builds ensure that the codebase remains stable, functional, and ready for deployment (Suhas and K, 2021; Arachchi and Perera, 2018). CI is especially useful in collaborative environments with multiple contributors, where integration scenarios can be complex and prone to errors (Stahl and Bosch, 2014; Ståhl et al., 2017).

CI systems require frequent building of projects, which is especially challenging for building multi-layered systems, such as Android-based platforms contributed by different stakeholders. It may take extensive time to build the entire platform. Prolonged build durations can significantly extend the CI testing process, limiting the number of projects that can be evaluated even with nightly builds (Vassallo et al., 2017). Therefore, foreseeing build failures in advance and skipping builds predicted to be successful may increase the efficiency of CI systems (Xie and Li, 2018; Marijan and Sagar, 2018).

We investigate the effective use of Machine Learning (ML) models to predict build failures in an industrial setting accurately. While many studies have focused on open source projects, only a recent few have investigated this topic in industrial settings (Kawalerowicz and Madeyski, 2023; Hong et al., 2024). Our study is based on a case study of Android-based digital TV software projects within the consumer electronics domain. We perform extensive empirical analyses using various models trained with combinations of existing and novel features. These features include those that are derived from the categorization of developers and modified files. In contrast, existing studies that rely solely on developer identities and modified files (rather than their feature-based representations) are not applicable in dynamic industrial environments, where both developers and files change frequently over time (Gligoric et al., 2015; Abdalkareem et al., 2019).

Another important difference of our study is that it focuses on predicting build outcomes involving multiple changes by

* Corresponding author.
E-mail addresses: Omer.Ozdemir2@vestel.com.tr (Özdemir), reyhan.aydogan@ozyegin.edu.tr (R. Aydoğan), hasan.sozer@ozyegin.edu.tr (H. Sözer).

https://doi.org/10.1016/j.jss.2026.112791
Received 19 July 2025; Received in revised form 12 January 2026; Accepted 17 January 2026
Available online 20 January 2026
0164-1212/© 2026 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

developers, rather than build outcomes triggered by a single change (Sharif et al., 2021; Marijan, 2023; Liang, 2024; Camacho, 2024). In large-scale, multi-layered industrial projects, where software builds are time-consuming, taking action for every individual code change is often impractical (Memon et al., 2017). As a result, CI tests that encompass contributions from multiple developers and their collective changes become unavoidable. Our evaluation also considers the temporal structure of real-world CI processes. Supervised ML models are typically evaluated by splitting data into training and testing sets. Although random data splitting yielded promising results in prior studies (Yaraghi et al., 2022; Abdalkareem et al., 2020; Lee et al., 2019), our findings suggest that chronological data splitting offers a more realistic and reliable assessment. We observe high precision but low recall in predicting failed builds in general, resulting in F1 scores of up to 60%. This trade-off allows hundreds of successful builds to be correctly skipped, with around a dozen failures potentially being missed. We performed an analysis to quantify this trade-off from the developer's perspective, focusing on the time gained through skipped builds and the implications of delayed failure detection. Our findings show that developers can save approximately 2.5 h per build on average. In all false negative cases, skipping a build necessarily delayed failure detection by at least one build cycle; however, this delay could not always be meaningfully quantified, as most failures were fixed before the next build was executed. Independently of measurement challenges, the impact of delayed failure detection varies with application criticality, development practices, and operational context. The contributions of this study are summarized as follows:

- An ML-based approach to predict build failures in CI that comprise collective changes contributed by multiple developers.
- Categorization of developers and modified files to obtain features that are resilient to dynamic changes in industrial environments.
- Release of a new, anonymized dataset derived from industrial software projects, made publicly available.
- A comprehensive evaluation with various ML algorithms and feature combinations.
- Evidence that chronological (time-based) data splitting yields more realistic and reliable model performance compared to random splits.

The remainder of this paper is organized as follows. We summarize the related work in Section 2. We explain the overall framework, research questions and the dataset in Section 3. We detail our experimental setup and evaluation in Section 4. We present and discuss the results in Section 5. Finally, we conclude the paper in Section 5.3.

2. Related work

Most studies on build failure prediction primarily focus on utilizing features related to source code changes such as number of modified lines and number of added code lines (Kawalerowicz and Madeyski, 2023; Hong et al., 2024; Chen et al., 2020; Saidani et al., 2022; Pan and Pradel, 2021; Yaraghi et al., 2022; Abdalkareem et al., 2020; Lee et al., 2019; Grano et al., 2019). In addition to these features, other studies take into account number of modified files or directories (Hong et al., 2024; Saidani et al., 2022; Pan and Pradel, 2021; Abdalkareem et al., 2020), developer profiles (Pan and Pradel, 2021; Yaraghi et al., 2022; Abdalkareem et al., 2020), and the previous build status (Kawalerowicz and Madeyski, 2023; Hong et al., 2024; Chen et al., 2020) in order to increase prediction accuracy. However, previous studies have not analyzed the effect of combinations of these features on build failure prediction by testing them with various ML algorithms in an industrial setting. A related approach by (Vale et al., 2023) also explores the combination of multiple types of features for predictive modeling. Their work is relevant in two ways. First, it considers multiple contributors applying changes concurrently. Second, it combines features derived from technical assets with those derived from a social perspective (e.g., developer roles). However, their study focuses on predicting merge conflicts rather than build failures. There exits studies that highlight the importance of features related to either developers or file modifications in predicting build failures (Silva et al., 2023; Hilton et al., 2017; Kerzazi et al., 2014). For instance, some studies (Pan and Pradel, 2021; Yaraghi et al., 2022; Abdalkareem et al., 2020) consider the seniority and experience level of developers as predictive features. However, these studies typically focus on per-change analysis, where each code modification is examined in isolation. In contrast, our goal is to predict build outcomes for multiple changes collectively. There is another study that attempts to predict outcomes for batches of changes (Ananthanarayanan et al., 2019). In that work, the authors aimed to identify individual changes that were unlikely to break the build and grouped such safe changes together before triggering a build. Such approaches are particularly relevant when the volume of changes exceeds the processing capacity of the integration pipeline, making batching unavoidable. However, these studies do not employ the specific developer- and file-specific properties explored in our work. Although our primary focus is not on commit selection or filtering, the proposed prediction approach and feature combinations can be effectively applied to optimize batch formation and scheduling. Developer-related features such as experience and file ownership have been examined before (Abdalkareem et al., 2020); however, their impact has not been analyzed within multi-developer, multi-change build contexts. Moreover, prior work has largely neglected the interplay between developer-related factors and file-type categorizations. In this study, we specifically classify modified files based on their types (i.e., file extensions), as this aspect has received limited attention in the literature on build failure prediction (Abdalkareem et al., 2020). We empirically investigate how incorporating such file-type categorizations, in combination with developer-related features, affects prediction accuracy.

Table 1 presents and overview and comparison of existing studies for build failure prediction. We observe that generally a small number of ML algorithms or deep learning models were used (Benjamin and Mathew, 2025). In our study, we test a wide range of both traditional and ensemble (boosting) algorithms. Features related to source code changes are utilized as features by all studies. However, a different subset of the remaining features is utilized by each study.

Existing approaches proposed for build failure prediction react on individual code changes, but a high volume of changes can lead to increased costs. To reduce these high costs, Jin et al. proposed two new approaches called SmartBuildSkip (Jin and Servant, 2020) and PreciseBuildSkip (Jin and Servant, 2022). Both approaches are based on ML algorithms, with PreciseBuildSkip additionally leveraging rule-based methods. They have been evaluated on large-scale open source datasets and have provided significant cost reductions by predicting failed builds and skipping the others. Kamath et al. (2024) combine two heuristics: grouping multiple builds and skipping those predicted to succeed. However, their approach still predicts build failures at the level of individual commits. In contrast, our study focuses on collective changes made by multiple developers. Some recent studies (Kawalerowicz and Madeyski, 2023; Saidani et al., 2022) have addressed the case of multiple changes made by a single developer. However, these studies do not explicitly address collaborative, multi-developer contributions within a build.

There exist recent studies that investigated various factors affecting build failure prediction. Kola-Olawuyi et al. (2024) examined 900,000 builds of 1689 open source projects. They investigated the effects of code ownership on build results and observed that build success rate increases as code ownership increases. Shimmi and Rahimi (2024) investigated the effects of different change types on build failure prediction. Their prediction model showed that maintenance-related changes, especially those containing bug fixes, cause the most build failures. In addition, Hong et al. (2024) conducted a large-scale industrial study at Atlassian, demonstrating that previous build results are effective predictors of build failures. Consistent with their findings, our study identifies the previous build status as the most influential feature. However, we extend our scope by providing a comprehensive analysis of

Table 1
Comparison of related Studies based on criteria adapted and extended from Arani et al. (2023) (OS: Open Source, I: Industrial).

Criteria	Our work	Kawalerowicz and Madeyski (2023)	Hong et al. (2024)	Saidani et al. (2022)	Pan and Pradel (2021)	Yaraghi et al. (2022)	Abdalkareem et al. (2020)	Lee et al. (2019)
Data Source	Vestel Projects (I)	Hidden Projects (I)	Atlassian Projects (I)	Travis Torrent (OS)	Apache (OS)	GitHub (OS)	GitHub (OS)	Samsung Projects (I)
Source Code Change Feature(s)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Number of modified file(s) or directory	**Yes**	No	**Yes**	**Yes**	**Yes**	No	**Yes**	No
Commit Messages	No	No	No	No	No	**Yes**	**Yes**	No
Previous Build Status	**Yes**	**Yes**	**Yes**	No	No	No	No	No
Categorized Developer	**Yes**	No	No	No	**Yes**	**Yes**	**Yes**	No
Type of the modified files	**Yes**	No	No	No	No	No	**Yes**	No
Feature Engineering	Multilabel Binarizer, Frequency encoding	Not reported	Not reported	Not reported	Not reported	Scaling and Weighting -Normalization and TF/IDF	Weighting -TF/IDF	Tagging -word2vec
Learning algorithm(s)	LR, KNN, DT, SVM, RF, GBM, XGBoost, LightGBM, AdaBoost	RF	Logistic Regression	LSTM -RNN	DT, RF, MLP, SVM, NB, LR	RF, SVM, XGBoost	DT (C4.5)	NN
Evaluation methods / Metrics	Expanding Window CV / F1, Precision, Recall, Precision-Recall AUC	Accuracy, F1, MCC and AUC	Accuracy, Precision, Recall, F1	Expanding Window CV / Accuracy and AUC	Stratified K-fold, Windows Cross and Time Series CV / FP, FN, and TP	Random K-fold / APFDC	Random K-fold / F1, Precision, Recall and AUC	Random K-fold / F1, Precision, Recall and Accuracy

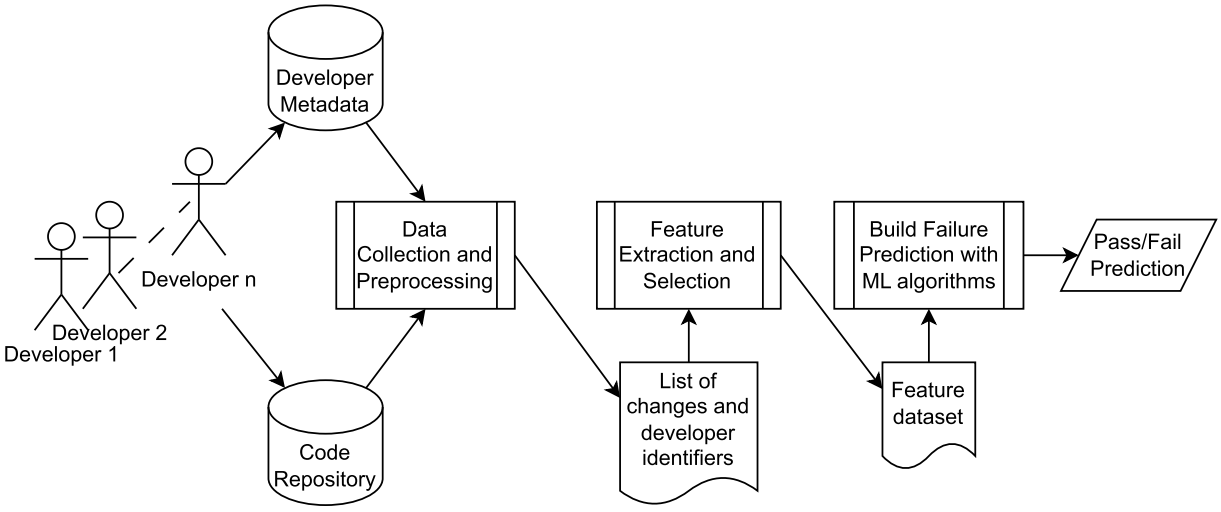


Fig. 1. The overall framework for build failure prediction.

multiple features and examining how their combinations affect build failure prediction across various ML algorithms.

Some organizations in the software industry prefer to run tests after every change (Memon et al., 2017). To reduce costs, they skip a subset of tests rather than skipping the entire build, or they schedule tests by predicting and prioritizing those that are more likely to fail (Kondareddy et al., 2025). Hereby, the tests that are associated with the modified parts of the code are selected/prioritized. The codebase is modeled as a code dependency graph for identifying test targets that are closer to changed code. This approach is similar to those adopted in other industrial systems such as SubmitQueue (Ananthanarayanan et al., 2019) and, much earlier, by Rothermel and Harrold (1994), who analyze dependency graphs to focus regression testing on modified classes and the components that depend on these classes in object-oriented software.

It has been reported that the total test execution time can grow quadratically, and eventually it may become infeasible to run tests after every change, even when only the affected tests are executed (Memon et al., 2017). Therefore, ML-based techniques are employed to further reduce the size of selected tests based on historical test outcomes (Machalica et al., 2019). Various other methods have been proposed for test case selection and prioritization (Pan et al., 2021), some of which are language-independent (Schwendner et al., 2025) while others are language-specific (Hundsdorfer et al., 2025). Also, several of these approaches have been adopted in industrial

Table 2
Feature categories and the corresponding data types.

Category Name / Common Name in Literature	Feature Name (Data Type)
Source Code Core Features (SCF) / Size	Total Addition Lines (int) Total Deletion Lines (int) Android Code Addition Lines (int) Android Code Deletion Lines (int) Total Lines Changed (int) Android Code Lines Changed (int) Total Lines (int) Android Code Lines (int) Android System Version (int)
Difference Count Features (DCF) / Diffusion	Number of Modified Files (int) Number of Modified Directories (int) Number of Modified Subsystems (int) Number of Modified Apps (int)
Categorized Developer Features (CDF) / Experience	Age Group (string array) Gender Group (string array) Seniority Group (string array) Experience Group (string array) Checking Control Status (string array)
File Type Feature (FTF) / Purpose	Changed Files (string array)
Previous Status Feature (PSF) / History	Previous Build Status (bool)

settings (Schwendner et al., 2025; Son et al., 2025; Khan et al., 2025). These test case selection and prioritization methods are complementary to approaches that skip entire builds based on failure prediction. HybridCISave (Jin and Servant, 2023) has been introduced as a hybrid approach that combines both strategies: it predicts and skips the execution of entire builds that are likely to pass, as well as selectively skips tests within builds when those tests are predicted to pass.

3. Build failure prediction framework and research questions

The overall build failure prediction process is shown in Fig. 1, which is formulated as a classification problem. A group of developers collectively work on a shared code repository, where they commit their changes. These commits are tracked and stored by the version control system. Data is then collected from this system along with developer-related metadata, such as the list of modified files, lines of code changed, commit timestamps, and developer identifiers. This collected data is analyzed for feature extraction and selection. Hereby, additional features like developer seniority are derived based on the available metadata. The resulting feature dataset serves as input to a set of ML algorithms trained to predict the outcome of a build as either pass or fail.

There are several important design decisions to be made as part of this framework. One key consideration is the selection of meaningful and predictive features that accurately capture the characteristics of a series of commits made by multiple developers and their potential impact on the build process. Another critical aspect is the choice of ML algorithms, as different models may vary in effectiveness depending on the structure and distribution of the data. In addition, evaluation metrics and methodology significantly influence the overall performance assessment of the build failure prediction process. For instance, adopting a time-series evaluation methodology rather than classic randomized K-fold evaluation makes a difference for data with temporal dependencies.

Our primary objective in this study is to investigate the effectiveness of ML in predicting the outcome of builds (i.e., whether a given build will succeed or fail) based on features derived from collective code changes across multiple software projects. Additionally, we aim to identify which ML algorithms and feature combinations yield the most accurate predictions. K-fold cross-validation has been widely employed in prior studies (Yaraghi et al., 2022; Abdalkareem et al., 2020; Lee et al., 2019), wherein datasets are randomly partitioned into k subsets to evaluate model performance across varying training and testing configurations. However, for practical applications in an industrial setting, models are typically trained only on data available up to a given point in time to predict future outcomes. Therefore, we also want to test the model in a more realistic setup, where the training and testing sets are formed without breaking the actual time order of data samples. In addition, given the limited number of ML algorithms tested so far, we aim to compare a broader range, particularly to assess whether significant differences exist between traditional and ensemble (boosting) algorithms. Finally, we are interested in evaluating the potential benefits and costs of selectively skipping builds using ML predictions from a developer-centric perspective, considering metrics such as time savings, resource usage, and delays in failure detection. To address these goals, we formulated the following research questions:

- **RQ1**: To what extent does the choice between random and chronological train-test splits affect the effectiveness of ML models in predicting build failures from features of collective changes?
- **RQ2**: Which ML algorithms and combinations of features lead to the most accurate predictions?
- **RQ3**: Can we improve the prediction accuracy by adopting ensemble learning approaches?
- **RQ4**: What are the potential benefits and costs of selectively skipping builds using ML predictions?

To address these questions, we conducted an industrial case study in the context of Vestel,[1] which is one of the largest TV manufacturers in Europe. Vestel manufactures TV products for 157 different brands from 145 different countries (Sozer and Gebizli, 2017). Therefore, there are multiple projects being maintained in different repositories. When developers commit their changes (e.g., add, modify or remove files) to the version control system (Subversion - SVN), these changes are subsequently merged daily into the main source code repository (Git). Once all project commits have been consolidated, they are processed through an Android nightly build system. The outcomes of this build process (success or failure) are recorded as part of the stored data.

The collected data is preprocessed, with a portion used for training the prediction model and the remainder reserved as test data to evaluate its predictive accuracy. Hereby, numerical features such as the number of modified files and total addition lines were not preprocessed. Discrete features such as the file type feature were enumerated.

[1] http://www.vestel.com.tr

The feature groups representing developer characteristics, such as *Seniority Group* and *Age Group*, were expanded into sub-features based on all possible combinations they contain. In this way, the attributes of the developers involved in each build and the number of developers possessing each attribute were numerically encoded into the dataset. Categorical file type features were encoded using the multi-label binarizer, while developer-related categorical features were encoded using frequency encoding (Charte et al., 2016; Dahouda and Joe, 2021; Pargent et al., 2022). Overall, the collected features are categorized into *Source Code Features (SCF)*, *Difference Count Features (DCF)*, *Categorized Developer Features (CDF)*, *File Type Features (FTF)*, and *Previous Status Feature (PSF)* as listed in Table 2.

SCF represent features that contain changes made to the source code, including lines added, deleted or changed. Special emphasis is given to Android-specific code (e.g., Java or Kotlin), and changes related to the core functionality of the system are isolated (Linares-Vásquez et al., 2014). In addition, SCF include Android System Version, which reflects the version of the middleware provided by the chip vendor.

DCF comprises features that numerically express daily changes of developers. It includes the number of changed files, directories and subsystems. Apart from these, since we are examining industrial Android projects, our data also includes the number of changed apps.

CDF categorizes developers according to various attributes; these include age, gender, seniority, experience level and *Checking Control Status*, which refers to information from developers about whether their changes were reviewed before being committed. There exist empirical studies regarding the potential impact of developer demographic features such as age and gender on developer productivity and team dynamics. For instance, Guzmán et al. (2023) identified that women in software development experience micro-inequities such as interruptions and lack of recognition more frequently than their male counterparts, which can impact their productivity and job satisfaction. Similarly, Murphy-Hill et al. (2022) observed that older developers face more pushback during code reviews compared to their younger counterparts, which can lead to increased cognitive load and reduced productivity. Developers are categorized according to the following characteristics:

- Age Group: {young, middle-aged, mature-aged} (-30, 30–40, 40 + years)
- Gender Group: {male, female}
- Seniority Group: {project manager, technologist, software architect, senior expert, advanced expert, expert, normal}
- Experience Group: {beginner, intermediate, expert} (3, 3–6, 6 + years)
- Checking Control Status: {no, mixed, yes}

While CDF categorizes developers using multi-label attributes, FTF captures the types of files modified by developers. Files are categorized based on predefined patterns, such as bin files, C files, json files, makefiles, xml files, gradle files, android source code (Java or Kotlin) files, and software configuration files (e.g., with extensions like .kl, .rc, .te or .ini). Finally, the dataset includes PSF *(Previous Build Status)* that reflects the outcome (i.e., success, fail) of the previous build. In our industrial context, feature selection was guided by the data that were reliably and consistently available across projects. All commits were systematically collected and stored, where structural and process related metadata such as lines of code changed, files modified, timestamps, and developer related attributes were consistently captured and formed the basis of our analysis. Although information such as commit messages or issue tracker discussions can provide valuable insights, these data sources were not uniformly captured or accessible in a standardized format across all projects. Commit messages, for example, may be written in different languages, be extremely brief such as “fixed”, or occasionally be omitted altogether, reflecting realistic industrial practices (Tian et al., 2022). Consequently, our analysis focused on features that could be consistently extracted from the existing datasets. Integrating unstructured textual data would also require substantial preprocessing, anonymization, and standardization to preserve data privacy and ensure comparability.

Table 3
An overview of the collected dataset.

Metric	Value
Total Software Build Count	1522
Total Project Count	9
Total Developer Count	13
Total Commit Count	3215
Total Number of Files Changed	28,563
Total Lines of Code Changed (LOC)	392,648
Total Build Success	1448
Total Build Fail	74

Table 4
Collected data for various project groups.

Project	Total Commits	# of Files Changed	LOC Changed
P1	695	9140	368,856
P2-P3	1000	8386	2743
P4-P7	1474	10,216	11,342
P8-P9	46	821	9707

Table 5
The list of projects and their build results.

Project	Total Builds	Successful Builds (%)	Failed Builds (%)
P1	105	88 (83.81)	17 (16.19)
P2	348	334 (95.98)	14 (4.02)
P3	59	57 (96.61)	2 (3.39)
P4	317	300 (94.64)	17 (5.36)
P5	258	245 (94.96)	13 (5.04)
P6	315	311 (98.73)	4 (1.27)
P7	80	76 (95.00)	4 (5.00)
P8	27	25 (92.59)	2 (7.41)
P9	13	12 (92.31)	1 (7.69)
Total	1522	1448 (95.13)	74 (4.87)

Our dataset contains over 1500 builds from 9 different projects (See Table 3). In total, 1584 builds were initially gathered. However, some flaky tests were identified as being caused by infrastructural issues. We filtered out these cases and excluded the corresponding builds from our dataset. As a result, 62 builds were removed, leaving 1522 builds for our analysis. In total, 13 developers contributed with 3215 commits. Properties of data collected for various groups of projects is listed in Table 4, where projects are enumerated from P1 to P9. Grouping is based on the organization of the company. P1 exhibits the highest volume of code changes. In fact, all the projects other than P1 existed before the CI system was established, resulting in fewer code changes despite being older than P1. The total number of builds, the number of successful builds and the number of those associated with failures are listed in Table 5. We can see that the dataset is unbalanced, with 95.13% of the builds being successful and only 4.87% resulting in failure.

The dataset as well as the source code used in this work are accessible through a public repository.[2] In the following section, we explain our experimental setup for evaluation.

4. Experimental setup and evaluation

Since we advocate adopting a chronological train- test splits (i.e., trained only on data available up to a given point in time to predict future outcomes) for a more realistic setting, we first examine how data splitting (i.e., randomly shuffled or chronologically ordered data) affects the prediction performance of the models for build failure prediction. Accordingly we set up our first experiment as illustrated in Fig. 2.

[2] http://doi.org/10.6084/m9.figshare.28930103

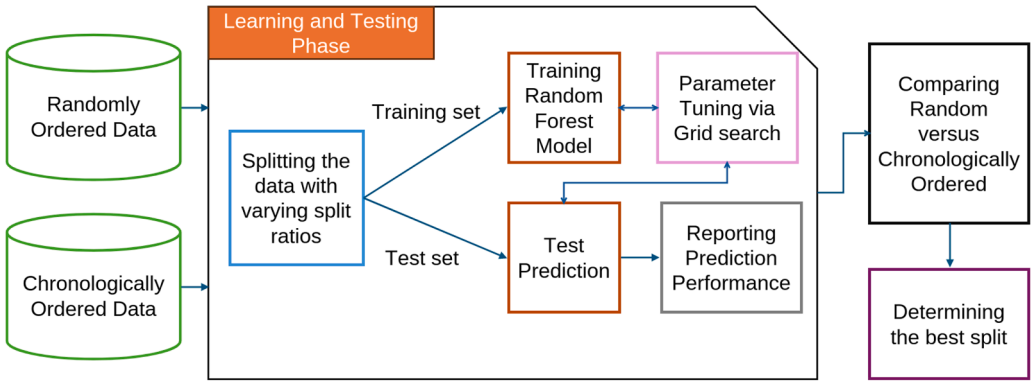


Fig. 2. Experimental setup Phase-1.

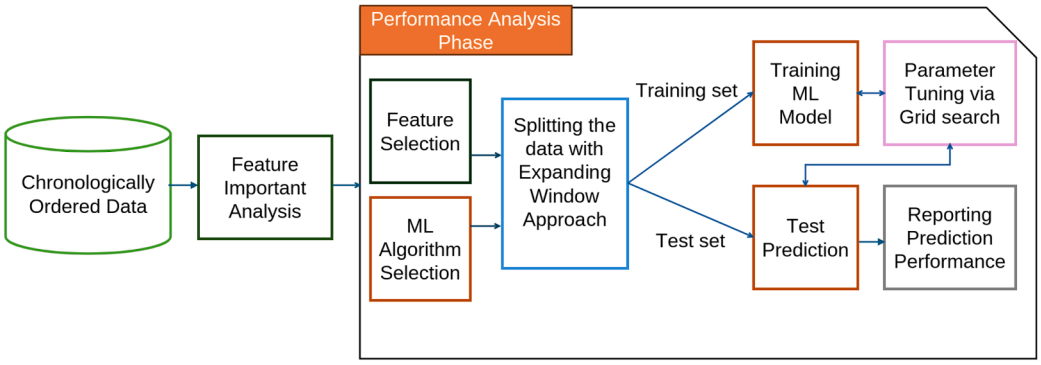


Fig. 3. Experimental setup Phase-2.

The dataset is split by considering two splitting approaches to construct training and testing datasets: *i)* Randomized data with varying split ratios, and *ii)* Chronological train-test split with an expanding window. For the randomized split, the procedure was performed once using a fixed random seed. A chronological train-test split mirrors real-world scenarios by ensuring that training data precedes test data temporally, allowing predictions to be based on the relationship between historical changes and future failures (Bisong et al., 2017; Shimmi and Rahimi, 2024). In this phase, we train a Random Forest model by using the training dataset. This model is tested on the test set by considering the performance metrics such as precision, recall, F1 Score, and Area Under the Precision-Recall Curve (PR-AUC).

In the second, *main* experimental setup, we focus on assessing the prediction performance of different ML methods and ensembling techniques. As depicted in Fig. 3, we first perform a feature importance analysis, and then for the selected combination of features, we train a separate model on the same training dataset. The same performance metrics listed in the first phase are used to assess the performance of each model for each feature combination on the test set. It is worth noting that we adopt chronologically ordered data along with the expanding window approach for simulating a realistic temporal evaluation setting.

In order to understand the relevance of each feature in our model, we employed various feature selection and importance evaluation techniques. These techniques help identify the most influential features, reducing the dimensionality of the dataset and improving model interpretability and performance (Chelvan and Perumal, 2017). In the feature selection process, we employed the Random Forest classifier. In this work, we utilize three feature analysis techniques to evaluate and select the most relevant features for prediction: *i) Feature Importance* identifying key predictors based on model-derived scores; *ii) Correlation Analysis* assessing the relationships between features and the target variable; and *iii) Select K-Best*. We detail the use of these techniques as follows:

Feature importance. This method determines the importance score according to how much the feature improves performance in different decision trees (Fabris et al., 2018) based on the given Random Forest model. Note that we encode FTF and CDF, which are represented as multilabel data (e.g., arrays of strings), into numeric matrices. When calculating feature importance, the contributions of all related numeric columns were summed to obtain a single importance score for the entire multi-label feature (Shi et al., 2020).

Correlation analysis. We used Pearson Correlation Analysis to assess the linear relationship between each numerical feature and the target variable (Xia and Li, 2017). A correlation value close to 1 or -1 indicates a strong relationship, whereas a value close to 0 indicates a weak or no relationship. While calculating the correlation values, we transformed multi-label representation for FTF and CDF into a format suitable for numeric correlation analysis (using Label Encoder) (Shi et al., 2020). This decision was motivated by the observation that features with above-average correlations hold greater significance (Wang et al., 2020).

Select K-best. We applied the Select K-Best method, a statistical method that ranks and selects the top-performing features, with the ANOVA F-test to select the top K features and to select the most relevant features for classification (Abid-Althaqafi and Alsalamah, 2024).

For the prediction model, we employed both traditional ML algorithms and ensemble boosting algorithms as listed in Table 6. Traditional algorithms are widely used for their simplicity and interpretability. Ensemble boosting algorithms combine the predictions of multiple models to achieve higher accuracy and robustness. Logistic regression (LR) is a highly effective model for binary classification (Zhang et al., 2018), particularly when there is a linear relationship between the dependent and independent variables. K-Nearest Neighbor (KNN) algorithm classifies data based on the most similar examples, and it performs well

Table 6
The set of ML algorithms used for experimentation.

Traditional Algorithms	Ensemble (Boosting) Algorithms
Logistic Regression	Random Forest
K Neighbors	Gradient Boost
Decision Tree	XGBoost
SVM	LightGBM
	AdaBoost

when the classes are clearly separable (Chamorro-Atalaya et al., 2022). Support Vector Machines (SVMs) are effective for complex classification tasks, particularly when data is not linearly separable. However, they can be sensitive to imbalanced datasets, often requiring techniques such as class weighting or resampling to improve performance (Qadrini et al., 2024). Decision Trees show a non-linear approach by dividing the data according to the most meaningful feature at each node, which provides transparency in the decision-making process.

Regarding the ensemble techniques, Random Forest constructs multiple decision trees using random subsets of the data. This approach reduces variance and mitigates overfitting by averaging the predictions of uncorrelated trees. Gradient Boosting and its optimized versions, XGBoost and LightGBM, build models sequentially, each correcting the errors of its predecessor. This iterative approach makes them highly effective in dealing with complex and imbalanced datasets (Yaacob et al., 2019). AdaBoost, on the other hand, enhances performance by combining multiple weak classifiers, adjusting their weights to focus more on previously misclassified instances (Das et al., 2015).

The parameters of each algorithm used in our model must be selected separately to give the best results under all conditions. Therefore, hyperparameter optimization was performed for each algorithm and feature set using Grid Search method (Bischl et al., 2023). Since our data is inbalanced, prediction accuracay may mislead us. Therefore, we employ F1 score as the main evaluation metric that allows the selection of the most appropriate hyperparameters (Gomiasti et al., 2024). It is important to scale data when using certain algorithms, such as KNN and SVMs, as these algorithms depend on distance calculations. To achieve better predictions, we applied Standard Scaling for KNN, and Maximum Absolute Scaling for SVM, and LR.

We would like to note that a direct baseline comparison using previously published datasets was not included in our setup, as these datasets are primarily designed for single-commit (per-change) analyses. In contrast, our framework targets the collective-change level, where multiple commits from different developers are integrated before a build. This fundamental difference in granularity prevents the direct reuse of prior datasets or reported results. Instead, our evaluation systematically compares a broad range of machine-learning algorithms, feature configurations, and temporal data-splitting strategies within this multi-commit context, providing insights that are complementary to existing research. In principle, individual commits from existing public datasets could be grouped together. However, doing so would prevent a direct and fair comparison with prior work, as existing studies analyze these datasets at the single-commit level using their original structure and feature definitions. Aggregating multiple commits and their messages into a single instance would fundamentally alter the feature representation and learning problem, and there is no guarantee that models trained under such conditions would be comparable to those reported in the literature. More importantly, public datasets do not provide a natural or semantically meaningful way to group commits into collective changes. In our setting, collective changes correspond to actual build units defined by the development process, where multiple commits are intentionally evaluated together. Public datasets lack this build-level context, making any aggregation strategy (such as grouping by time window, number of commits, or author) inherently arbitrary and potentially biased.

Existing public datasets are not only incompatible with our framework in terms of analysis granularity (i.e., single-commit versus collective-change level), but also differ substantially in their feature composition (See Table 7). In particular, our study is conducted on an industrial-scale Android continuous integration environment, and the corresponding feature sets are not available in commonly used benchmark datasets. As a result, these datasets cannot adequately represent the characteristics of collective changes integrated before a build, nor support a fair evaluation of the feature sets employed in our approach. It is worth noting that other related studies do not use public datasets. Therefore, Table 7 only provides the feature comparison with those open datasets. We present and discuss the results in the following section.

5. Results and discussion

In this section, we first examine the accuracy of prediction for both random and chronological splitting of training and test data. Then, the set of features are analyzed with feature analysis techniques and the performance of traditional ML algorithms is evaluated with various feature sets combined based on their importance. The effectiveness of ensemble algorithms are then investigated for varying combinations of feature sets. Threats to validity are discussed at the end of the section.

5.1. Effectiveness of prediction for randomly and chronologically ordered data (RQ1)

On one hand, most of the studies mentioned in Table 1, train and evaluate their predictive models using randomized datasets, primarily employing a K-fold cross-validation approach. On the other hand, a sequence of builds are not entirely time-independent of one another. Collective code changes develop over time, which means that training and testing on randomized datasets can be misleading, as the model inadvertently leverages future data instances (i.e., nearby test instances) during the training phase. In real-world scenarios, we can only use instances that have been observed thus far and assess their predictive performance on future instances that have yet to be encountered. In this context, it would be more beneficial to take the temporal relationships within the dataset into account and to operate on a chronologically ordered dataset.

Accordingly, we aimed to investigate the prediction performance for the random and chronological splitting of training and test data. For the prediction model, we used the Random Forest algorithm, as it has been widely used and shown to deliver satisfactory performance in previous studies (Kawalerowicz and Madeyski, 2023; Pan and Pradel, 2021; Yaraghi et al., 2022). We conducted experiments using different training and test dataset split ratios: 90-10, 80-20, 70-30, and 60-40. We compared two approaches: *i) Random:* randomly selecting the training and test samples, and *ii) Chronological:* using the initial portion of the timely-sorted data for training and the remaining portion for testing. For each split, the performance on both data splits is given in Table 8, where the last two columns denote the number and percentage of build failures and success instances in the corresponding test set.

Although the number/percentage of failure instances in the test sets are almost the same, the results differ depending on whether random or chronological splitting is used. Chronological splitting generally yields higher precision, while random splitting tends to produce higher recall. This result supports our claim that the performance of the prediction model is misleading due to the use of future instances in the training phase. Although random splitting has been applied in previous studies (Yaraghi et al., 2022; Abdalkareem et al., 2020; Lee et al., 2019), yielding promising results, this approach is not realistic for real-world applications. Models can only be trained on data collected until a certain point of time to make predictions regarding the builds in the future.

To better reflect a real-world scenario, we tested the effectiveness of the expanding window approach as depicted in Fig. 4. Hereby, we initially allocated 40% of the data for the training set. Subsequently, the

Table 7
Comparison of features in public datasets used by related studies.

Feature	Yaraghi et al. (2022)	Abdalkareem et al. (2020)	Pan and Pradel (2021)	Saidani et al. (2022)
Source Code Core Features (SCF)				
Total Addition Lines	✓	✓	✓	×
Total Deletion Lines	✓	✓	✓	×
Android Code Addition Lines	×	×	×	×
Android Code Deletion Lines	×	×	×	×
Total Lines Changed	✓	✓	×	✓
Android Code Lines Changed	×	×	×	×
Total Lines	×	×	×	✓
Android Code Lines	×	×	×	×
Android System Version	×	×	×	×
Difference Count Features (DCF)				
Number of Modified Files	×	✓	✓	✓
Number of Modified Directories	×	✓	✓	×
Number of Modified Subsystems	×	✓	✓	×
Number of Modified Apps	×	×	×	×
Categorized Developer Features (CDF)				
Age Group	✓	×	×	×
Gender Group	×	×	×	×
Seniority Group	×	×	×	×
Experience Group	✓	✓	✓	×
Checking Control Status	✓	×	×	×
File Type Feature (FTF)				
Changed Files	×	✓	×	×
Previous Status Feature (PSF)				
Previous Build Status	×	×	×	×

Table 8
Prediction performance of the random forest algorithm for random and chronological splitting of training and test data.

Splitting	Train-Test Ratio	Precision	Recall	F1 Score	PR-AUC	Test Success	Test Fail
Random	90-10	0.3334	0.1667	0.2223	0.2154	147 (96.08%)	6 (3.92%)
Chronological		1.0000	0.1429	0.2500	0.2190	146 (95.42%)	7 (4.58%)
Random	80-20	0.5000	0.2143	0.3000	0.2410	291 (95.41%)	14 (4.59%)
Chronological		1.0000	0.1111	0.1920	0.5250	287 (94.10%)	18 (5.90%)
Random	70-30	0.6364	0.3182	0.4242	0.2806	435 (95.19%)	22 (4.81%)
Chronological		1.0000	0.3077	0.4706	0.4306	431 (94.31%)	26 (5.69%)
Random	60-40	0.7857	0.3253	0.4583	0.3480	575 (94.42%)	34 (5.58%)
Chronological		1.0000	0.2188	0.3590	0.3505	577 (94.75%)	32 (5.25%)

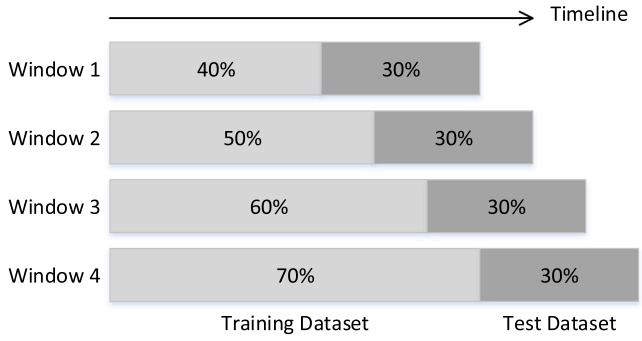


Fig. 4. Our Application of the expanded window approach.

Table 9
Results obtained with the expanding window approach.

Window	Train Range (%)	Test Range (%)	Precision	Recall	F1 Score	PR-AUC
1	0–40	40–70	1.0000	0.1053	0.1905	0.1469
2	0–50	50–80	0.6667	0.1250	0.2105	0.2048
3	0–60	60–90	1.0000	0.2400	0.3871	0.3670
4	0–70	70–100	1.0000	0.3077	0.4706	0.4306
Avg	–	–	**0.9167**	**0.1945**	**0.3147**	**0.2873**

training set was incrementally expanded by 10% at each step. Throughout the process, the test set size was consistently maintained at 30% of the total data. To evaluate overall performance, we present the average prediction results across these test sets.

Results are listed in Table 9. We observe that the precision rate is significantly higher compared to recall further indicates that the learned model avoids false positives (i.e., predicting failure for a successful build), yet it often overlooks genuine positive cases (i.e., failed builds). Since the test dataset contains a minority of failing classes (i.e., positive classes), this makes it difficult for our model to predict failures. Recall that our dataset is imbalanced. That is, if we were to retrain our model to focus on detecting successful builds, we would likely see a significant increase in recall.

We also observe that the expanding window approach allows the model to enhance its ability to predict the minority class by progressively encountering a larger and more representative sample of minority class instances over time. Performance metrics, particularly PR-AUC scores, gradually improve as the training set size increases. One can

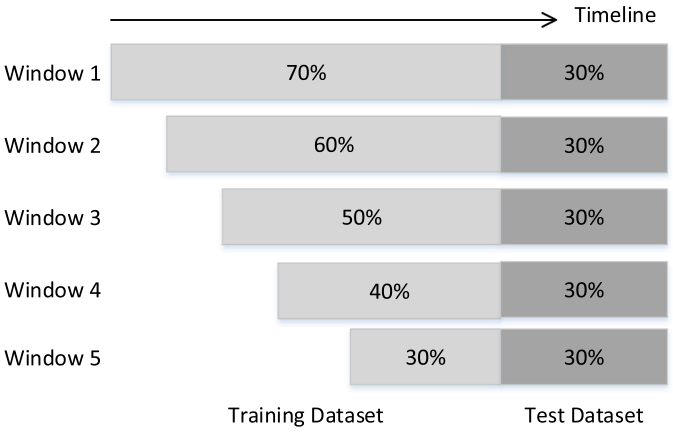


Fig. 5. Expanding window evaluation on a fixed test set.

Table 10
Results with a fixed test set as the training set is progressively reduced by discarding older samples as depicted in Fig. 5.

Window	Train Range (%)	Test Range (%)	Precision	Recall	F1 Score	PR-AUC
1	0–70	70–100	1.0000	0.3077	0.4706	0.4306
2	10–70	70–100	1.0000	0.3077	0.4706	0.4318
3	20–70	70–100	1.0000	0.1154	0.2069	0.3864
4	30–70	70–100	1.0000	0.1154	0.2069	0.3647
5	40–70	70–100	0.5000	0.0385	0.0714	0.3159
Avg	–	–	**0.9000**	**0.1769**	**0.2853**	**0.3859**

question this interpretation since the test dataset is different for each window in our application of the expanded approach, where the window of the test dataset is shifted in each iteration (See Fig. 4). We performed another experiment to observe the effect of increasingly larger training dataset by fixing the test dataset. The approach adopted for this experiment is depicted in Fig. 5.

The experiment depicted in Fig. 5 evaluates the prediction performance of the model on the last part of the dataset (70–100) taken as the test dataset. The training dataset is progressively reduced by discarding older samples, starting from the full history (0-70) down to only the most recent (40-70) portion of the dataset. Results obtained with this experiment is listed in Table 10, confirming that prediction accuracy improves as the training dataset grows by incorporating more historical data over time.

One might argue that the observed F1 scores (up to 0.47) are relatively low for adopting such an approach in an industrial setting. However, the approach can still be valuable depending on the specific domain, as it enables the correct prediction of hundreds of successful builds, thereby allowing them to be safely skipped, at the cost of potentially missing a dozen of failed builds. To sum up, utilizing a chronologically ordered dataset for predicting build failures based on collective code changes is more reliable. It is challenging to predict potential failures since the number of positive instances is much less than that of negative instances. Since the training and testing split ratio significantly affects the prediction performance, we suggest adopting an expanding window approach to report the generalized performance.

Furthermore, we investigate the results of Principal Component Analysis (PCA), where the distribution of the dataset is projected on a two dimensional plain to examine the difficulty of the prediction task for this dataset. Fig. 6 presents the PCA visualization, illustrating the two-dimensional projection of the temporally separated training and test datasets (70-30 chronologically ordered). When the graphs are examined, it is seen that there are very few positive class examples. Also, it is seen that positive and negative class examples overlap with each other. These observations support that the classification task on this dataset is challenging due to both class overlap and significant class imbalance.

Table 11
Feature importance scores obtained using random forest.

Feature Category	Feature	Importance
Previous Status Feature	**Previous Build Status**	**0.1916**
File Type Feature	**Changed Files**	**0.1683**
Difference Count Features	Number of modified files	0.0596
Source Code Features	Total Addition Lines	0.0457
Difference Count Features	Number of modified directories	0.0439
Source Code Features	Total Lines	0.0427
Source Code Features	Total Lines Changed	0.0421
Source Code Features	Total Deletion Lines	0.0414
Source Code Features	Android Code Lines	0.0396
Source Code Features	Android Code Deletion Lines	0.0382
Source Code Features	Android Code Lines Changed	0.0366
Source Code Features	Android Code Addition Lines	0.0342
Difference Count Features	Number of modified subsystems	0.0272
Categorized Developer Features	Gender Group	0.0380
Categorized Developer Features	Experience Group	0.0522
Categorized Developer Features	Checking Control Status	0.0398
Categorized Developer Features	Seniority Group	0.0434
Categorized Developer Features	Age Group	0.0378
Difference Count Features	Number of modified apps	0.0219
Source Code Features	Android System Version	0.0160

The visualizations retain a similar pattern even when only the top two features are considered according to the feature importance analysis performed in the following.

5.2. Effectiveness of feature selections and ML algorithms (RQ2)

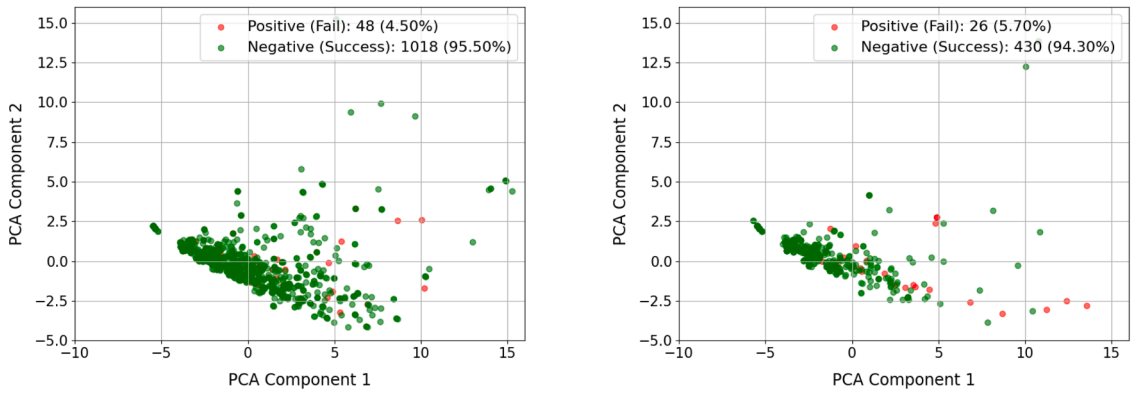
In the following, we first present the results of our feature importance analysis. We evaluate these results to guide the construction of our feature combinations. Then, we report the outcomes of experiments conducted using traditional ML algorithms with various combinations of features.

5.2.1. Feature importance analysis

In this section, we analyze the importance of dataset features by adopting three feature selection techniques. Tables 11–13 present the feature importance scores obtained through Random Forest, Correlation Analysis, and the Select K-Best method, respectively.

We see that *Previous Build Status* and *Changed Files* stand out across all three tables. *Previous Build Status* ranks first in feature importance, followed by *Changed Files* in second place. The remaining features have significantly lower and nearly identical importance scores (see Table 11). The same pair of features also emerge as the most prominent in the correlation analysis results shown in Table 12. In this case, their correlation values exceed those of the other features; however, *Previous Build Status* is substantially higher than the rest, while *Changed Files* occupies the second rank but with a value comparable to several remaining features. We also analyzed the correlation between *Previous Build Status* and *Changed Files* since these two are the two most relevant features in terms of predictive power. The correlation between them turned out to be very low, nearly zero (0.0507 on average), indicating that these features are largely independent and they provide complementary information.

We observe similar results with the normalized score estimated by K-Best feature method, where *Previous Build Status* and *Changed Files* rank the first and second, respectively. Scores attributed to all the other features are significantly lower (See Table 13). Fig. 7 provides an overview of the results from three feature analysis metrics, Feature Importance (obtained using Random Forest), Correlation and Select K-Best. All values were normalized to a common scale between 0 and 1 for comparability. PSF and FTF stand out as the most valuable feature sets. These feature sets are followed by CDF and SCF, respectively. DCF feature set showed relatively worse performance in feature importance analysis compared to other feature sets.



(a) PCA Training Data Set (b) PCA Testing Data Set

Fig. 6. Principal component analysis (PCA) for chronological data split (70-30). (a) PCA training data set, (b) PCA testing data set.

Table 12
Feature correlation values.

Feature Category	Feature	Correlation
Previous Status Feature	**Previous Build Status**	**0.4702**
File Type Feature	**Changed Files**	**0.1284**
Categorized Developer Features	Experience Group	0.1040
Categorized Developer Features	Seniority Group	0.0704
Categorized Developer Features	Age Group	0.0608
Categorized Developer Features	Gender Group	0.0497
Categorized Developer Features	Checking Group	0.0452
Source Code Features	Total Lines	0.0406
Source Code Features	Android Code Lines	0.0386
Difference Count Features	Number of modified apps	0.0322
Difference Count Features	Number of modified subsystems	0.0299
Difference Count Features	Number of modified files	0.0261
Difference Count Features	Number of modified directories	0.0230
Source Code Features	Total Deletion Lines	0.0192
Source Code Features	Android System Version	-0.0188
Source Code Features	Android Code Deletion Lines	0.0167
Source Code Features	Android Code Lines Changed	0.0093
Source Code Features	Total Lines Changed	0.0053
Source Code Features	Android Code Addition Lines	0.0032
Source Code Features	Total Addition Lines	-0.0027

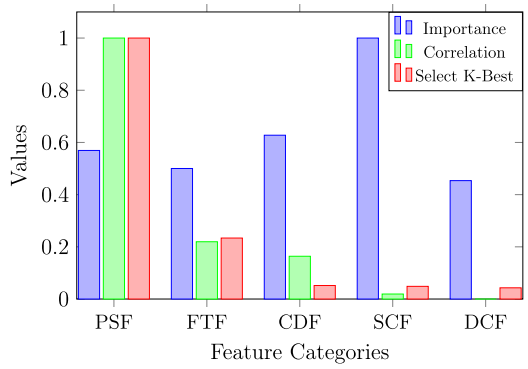


Fig. 7. Feature importance, correlation, and select K-best scores by feature categories.

5.2.2. *Performance of ML algorithms with varying feature combinations*

In this section, we explore well-established traditional ML algorithms and analyze the influence of various feature sets on prediction performance. The previous section provided insights into which features may affect the prediction model. Given that it is impractical to evaluate all possible combinations of these features, we adopt an approach that involves combining some primary feature categories, particularly PSF, to examine their impact on the prediction model. PSF, identified as the most important feature, is selected as the core feature. To systematically assess the impact of feature combinations, we start with PSF alone and incrementally add the other most important feature sets, FTF and CDF. Rather than considering every possible triple combination, we concentrate on the most significant features aside from PSF, namely FTF and SCF, as highlighted in the previous section. Specifically, the SCF feature set, as the features in this category are widely used in most of the ML-based CI build studies (See Table 1), is also included in the best-performing feature set combination. We grouped features mainly based on their source, which guided the construction of the feature sets. Although features within a set can be heterogeneous, performing a finer-grained combinatorial selection would lead to an exponential increase in the number of experiments. Additionally, our analysis indicates that the performance differences among individual features within the same set are minor. Ultimately, we assess the prediction performance when all available features are utilized (SCF + PSF + FTF + DCF + CDF). It is important to note that we have not included other combinations involving CDF and DCF, as preliminary evaluations indicated limited potential, making their inclusion less justified given the trade-off with runtime.

Table 14 present the results obtained using traditional ML algorithms with various feature combinations, assessed through the expanding window evaluation technique. All evaluation metrics in the table, except for F1 Max, represent the average across window folds, while F1 Max is the highest F1 score obtained among the window folds. Notably, the highest recall values are observed when all features are utilized in the prediction models for LR and SVM, whereas the opposite trend is seen for Decision Tree and KNN, where using fewer features yields better recall. The best performance with KNN is observed in feature set combinations that include PSF + FTF, PSF + CDF, and PSF + FTF + CDF, indicating that combining PSF with either or both of the other top-ranked feature sets enhances the model's effectiveness. Among the ML algorithms considered, KNN exhibits the poorest performance, with a recall of 0.2176. This algorithm is particularly sensitive to high dimensionality, a phenomenon known as the curse of dimensionality. In our dataset, we have a significant number of features, many of which may be irrelevant or contain noise, leading to skewed distance calculations. Additionally, KNN tends to perform poorly with imbalanced datasets, where there is a substantial disparity between class distributions. Since we employ majority voting to classify samples, the combination of high dimensionality and the scarcity of positive samples in both training and test sets adversely affects performance. Conversely, LR yields the best results in terms of both recall and F1 scores. One of the key advantages of LR is its ability to generalize well on small to moderate-sized datasets, particularly when

Table 13
Features based on select K-best with min-max normalized scores.

Feature Category	Feature	Score	Normalized Score
Previous Status Feature	**Previous Build Status**	**518.8367**	**1.000000**
File Type Feature	**Changed Files**	**121.2420**	**0.233668**
Categorized Developer Features	Age Group	16.0419	0.030920
Source Code Features	Total Lines	9.5657	0.018435
Source Code Features	Android Code Lines	9.1613	0.017655
Difference Count Features	Number of modified subsystems	7.1259	0.013731
Categorized Developer Features	Seniority Group	6.3079	0.012152
Difference Count Features	Number of modified directories	6.0608	0.011683
Difference Count Features	Number of modified apps	4.9352	0.009510
Difference Count Features	Number of modified files	4.2045	0.008101
Categorized Developer Features	Checking Control Status	3.6674	0.007066
Source Code Features	Android System Version	2.7057	0.005215
Source Code Features	Total Deletion Lines	1.8081	0.003485
Source Code Features	Android Code Deletion Lines	1.2314	0.002373
Categorized Developer Features	Gender Group	0.8146	0.001570
Source Code Features	Android Code Lines Changed	0.4347	0.000838
Source Code Features	Total Lines Changed	0.2524	0.000487
Source Code Features	Android Code Addition Lines	0.0762	0.000147
Categorized Developer Features	Experience Group	0.0579	0.000112
Source Code Features	Total Addition Lines	0.0001	0.000000

Table 14
Performance metrics for traditional ML algorithms.

Algorithm	Feature Set	Precision	Recall	F1	F1 Max	PR-AUC
Logistic Regression	PSF	0.75	0.0591	0.1059	0.2727	0.2707
	PSF + FTF	**1**	0.202	0.3246	0.4706	0.3014
	PSF + CDF	**1**	0.1888	0.3138	0.4375	0.2776
	PSF + FTF + CDF	0.9097	0.2529	0.3894	0.5143	0.349
	PSF + FTF + SCF	**1**	0.202	0.3246	0.4706	0.3014
	PSF + FTF + CDF + SCF	0.7917	0.2625	0.3834	0.5556	0.3504
	PSF + FTF + CDF + SCF + DCF	0.7054	**0.3586**	**0.4737**	**0.6**	**0.411**
KNN	PSF	0.5439	**0.2176**	0.3032	0.4571	0.3441
	PSF + FTF	**1**	**0.2176**	**0.3508**	**0.4706**	0.3065
	PSF + CDF	**1**	**0.2176**	**0.3508**	**0.4706**	0.3378
	PSF + FTF + CDF	**1**	**0.2176**	**0.3508**	**0.4706**	0.3441
	PSF + FTF + SCF	**1**	**0.2176**	**0.3508**	**0.4706**	0.3065
	PSF + FTF + CDF + SCF	**1**	0.1976	0.3247	**0.4706**	0.2976
	PSF + FTF + CDF + SCF + DCF	0.7258	0.178	0.2787	0.3784	**0.3584**
Decision Tree	PSF	0.3083	0.2273	0.2607	0.4091	0.1936
	PSF + FTF	0.2338	0.2273	0.2292	0.375	0.1957
	PSF + CDF	0.3718	**0.2685**	0.2957	0.4324	0.2342
	PSF + FTF + CDF	**0.3971**	0.2273	0.2811	0.383	0.2393
	PSF + FTF + SCF	0.3729	0.2273	0.2793	0.439	**0.2655**
	PSF + FTF + CDF + SCF	**0.3971**	0.2273	0.2811	0.3830	0.2393
	PSF + FTF + CDF + SCF + DCF	0.3903	0.2561	**0.301**	**0.5217**	0.2613
SVM	PSF	0	0	0	0	0.2824
	PSF + FTF	**1**	0.2176	0.3508	0.4706	0.2985
	PSF + CDF	**1**	0.2176	0.3508	0.4706	0.3285
	PSF + FTF + CDF	0.7875	0.2373	0.3626	0.5143	0.3108
	PSF + FTF + SCF	**1**	0.2176	0.3508	0.4706	0.2985
	PSF + FTF + CDF + SCF	0.7875	0.2373	0.3626	0.5143	0.3108
	PSF + FTF + CDF + SCF + DCF	0.8239	**0.339**	**0.4678**	**0.5946**	**0.4111**

classes can be separated by a linear boundary. Decision Trees may overfit in small datasets, while SVMs can be adversely affected by outliers, especially when using hard margins.

As is the case with LR and SVM, the Decision Tree algorithm appears to yield the most successful results in terms of F1 score when all feature sets are utilized. In addition, Decision Tree offers a transparent and interpretable structure, as illustrated in Fig. 8, making it valuable for understanding feature-based decision logic. In general, incorporation of PSF (the top node in the Decision Tree) significantly improves the results, which is consistent with the feature importance analysis.

Fig. 9 shows another decision tree constructed to analyze the impact of various features on the prediction power and to analyze those leading to false (negative or positive) results. This tree is constructed from the confusion matrix of the LR model, which produced the best results. The leaf nodes of the tree represent prediction outcomes, i.e., TN (True Negative), TP (True Positive), FN (False Negative), FP (False Positive), rather than predictions, i.e., Success, Fail. The root node indicates the same feature as the most influential factor in the decision process. FPs predominantly emerge along paths characterized by moderate changes in the number of modified directories and the number of lines deleted. In these cases, the model tends to overestimate the likelihood of failure, misclassifying successful builds as failed ones. FNs occur when the changes are small in size but involve a relatively large number of certain file types (i.e., File Type 11, which corresponds to the “others” category and primarily includes files without extensions like newly created folders). In such cases, the model incorrectly assumes low failure risk,

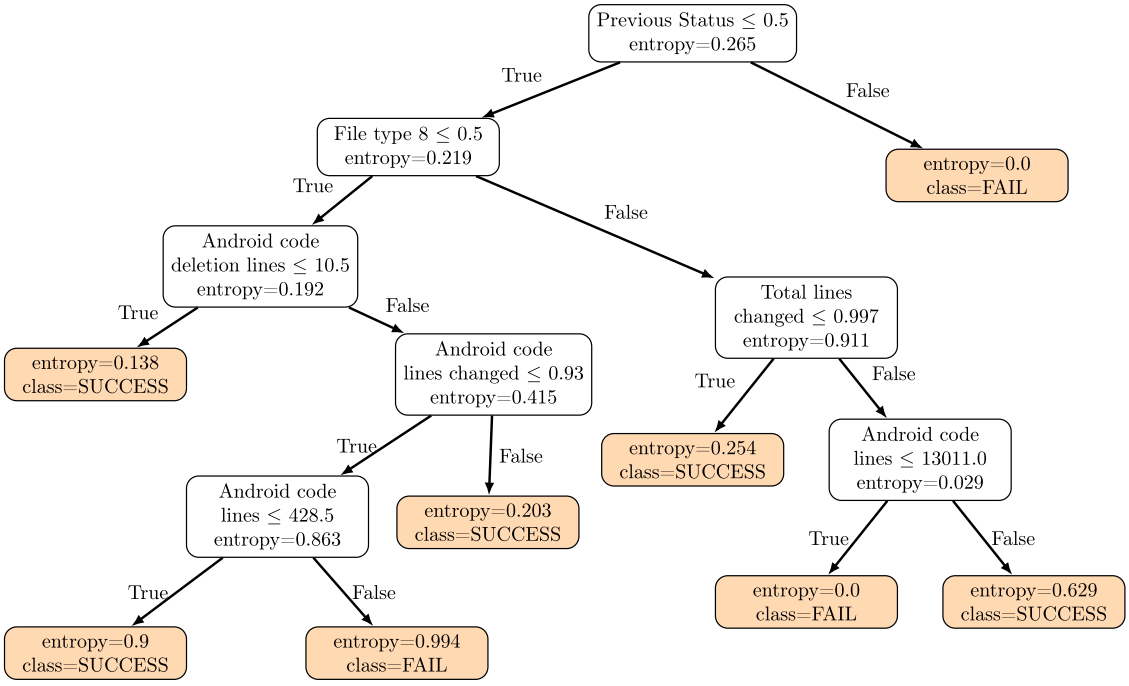


Fig. 8. Visualization of the constructed decision tree.

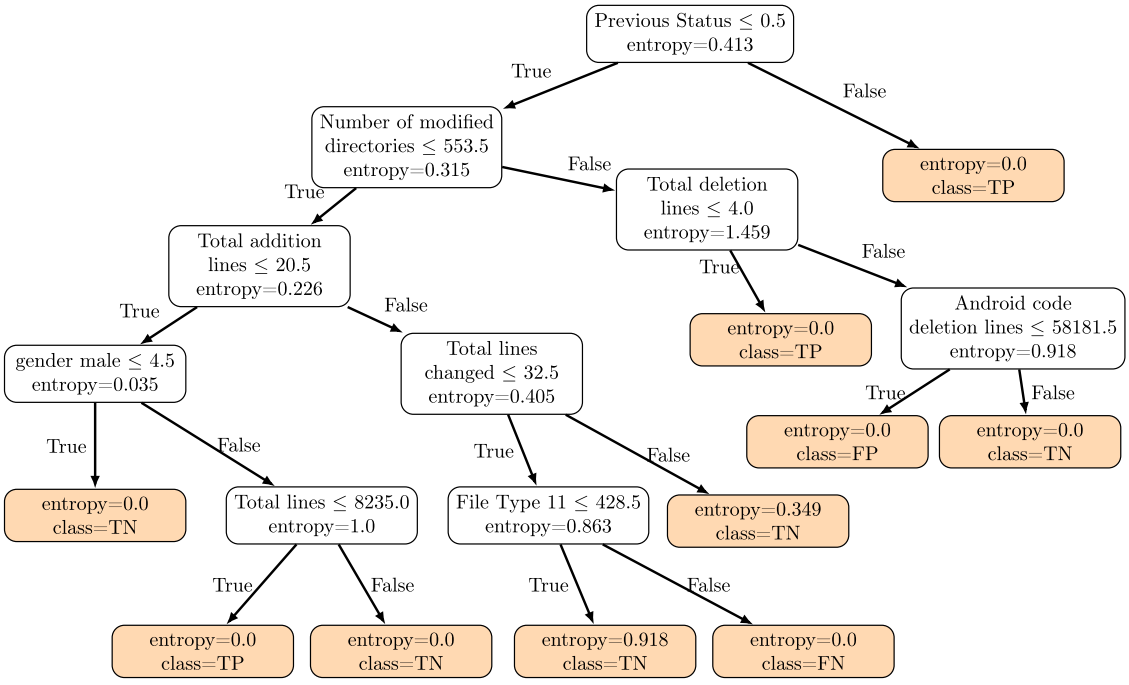


Fig. 9. Decision tree constructed based on the confusion matrix.

overlooking that small edits in structural parts of the project can still cause build failures.

5.3. Effectiveness of ensemble algorithms (RQ3)

This section investigates whether prediction performance can be improved by leveraging ensemble algorithms. Table 15 presents the results obtained using ensemble methods with feature sets combined based on their importance. Overall, we do not observe substantial improvements in performance compared to traditional algorithms. Notably, the best results are not necessarily achieved with the largest number of feature sets, which contrasts with the findings in Table 14. Combining all feature sets does not always yield better performance. Additionally, for AdaBoost, augmenting the PSF feature set with other sets results in no performance gain, suggesting that some ensemble algorithms may struggle to generalize effectively when the input space becomes overly complex.

An individual analysis of the ensemble boosting algorithms reveals that Adaboost showed the best performance by a considerable margin in PR-AUC values. XGBoost and Gradient Boosting can reach upto F1 value of 0.51. Gradient Boosting accompanied Adaboost in recall with the highest value. LightGBM showed the lowest performance among boosting algorithms in all metrics. It can be seen that LightGBM has difficulty learning on unbalanced and limited datasets.

Table 15
Performance metrics for ensemble algorithms.

Algorithm	Feature Set	Precision	Recall	F1	F1 Max	PR-AUC
Random Forest	PSF	0.9444	0.2045	0.3238	**0.4706**	0.2695
	PSF + FTF	**1**	0.1164	0.2085	0.2222	0.2729
	PSF + CDF	**1**	**0.2045**	**0.3302**	**0.4706**	0.2803
	PSF + FTF + CDF	**1**	0.1445	0.2401	**0.4706**	**0.2997**
	PSF + FTF + SCF	**1**	0.1945	0.3176	**0.4706**	0.2698
	PSF + FTF + CDF + SCF	**1**	0.1445	0.2401	**0.4706**	**0.2997**
	PSF + FTF + CDF + SCF + DCF	0.9167	0.1945	0.3147	**0.4706**	0.2873
Gradient Boosting	PSF	0.3234	0.2041	0.2467	0.4091	0.1579
	PSF + FTF	0.5455	0.2076	0.2872	0.3721	0.2440
	PSF + CDF	0.6597	0.2176	0.3224	0.4571	0.2697
	PSF + FTF + CDF	**0.9410**	0.2045	**0.3235**	0.4571	0.2544
	PSF + FTF + SCF	0.6637	0.1937	0.2737	0.4651	0.2320
	PSF + FTF + CDF + SCF	0.3424	**0.3331**	0.2852	0.4878	0.2582
	PSF + FTF + CDF + SCF + DCF	0.3386	0.2657	0.2936	**0.5098**	**0.2847**
XGBoost	PSF	**1**	0.1913	0.3076	0.4706	0.3008
	PSF + FTF	**1**	0.1913	0.3076	0.4706	0.2952
	PSF + CDF	**1**	0.2176	0.3508	0.4706	0.3132
	PSF + FTF + CDF	0.9444	0.198	0.3205	0.4	0.2922
	PSF + FTF + SCF	**1**	0.1913	0.3076	0.4706	0.2952
	PSF + FTF + CDF + SCF	**1**	**0.2273**	**0.3617**	**0.5143**	**0.3215**
	PSF + FTF + CDF + SCF + DCF	**1**	0.1909	0.3059	**0.5143**	0.3032
LightGBM	PSF	0.2014	**0.0549**	**0.0857**	0.125	0.0671
	PSF + FTF	0.093	0.0488	0.0619	0.1081	0.0499
	PSF + CDF	0.05	0.0132	0.0208	0.0833	0.0528
	PSF + FTF + CDF	0.0625	0.0132	0.0217	0.087	0.0591
	PSF + FTF + SCF	**0.3438**	0.0396	0.064	**0.1818**	0.0679
	PSF + FTF + CDF + SCF	0.2500	0.0096	0.0185	0.0741	0.0635
	PSF + FTF + CDF + SCF + DCF	0.25	0.0192	0.0357	0.1429	**0.0766**
AdaBoost	PSF	**1**	**0.2176**	**0.3508**	**0.4706**	**0.6269**
	PSF + FTF	1	0.2176	0.3508	0.4706	0.6269
	PSF + CDF	1	0.2176	0.3508	0.4706	0.6269
	PSF + FTF + CDF	1	0.2176	0.3508	0.4706	0.6269
	PSF + FTF + SCF	1	0.2176	0.3508	0.4706	0.6269
	PSF + FTF + CDF + SCF	1	0.2176	0.3508	0.4706	0.6269
	PSF + FTF + CDF + SCF + DCF	1	0.2176	0.3508	0.4706	0.6269

5.4. Benefits and costs of prediction-based build skipping (RQ4)

The benefit of selectively skipping builds is measured in terms of time savings, which also translate into reduced computational resource usage and energy consumption. Specifically, we quantify the benefit as the average build duration saved per build, which we refer to as the Mean Build Time Saved (MBTS), and compute it as follows:

$$\text{MBTS} = \frac{\text{Number of builds skipped} \times \text{Average build duration}}{\text{Total number of builds}}$$

When applying this formula with for *LR - Expanding Window Last Fold* configuration, which produced the best results, we consider a total of 456 builds in the test set. The number of builds skipped corresponds to the sum of true negatives (428) and false negatives (14), resulting in a total of 442. Given that the average build duration is 151 min, MBTS is calculated as $(442 \times 151)/456 \approx 146.36$ min, which is around 2.5 h per build. These results reflect a meaningful reduction in build duration, leading to considerable savings in computational resources and energy consumption across the project. These efficiency improvements are consistent with prior work on ML-based build skipping, which also reports notable cost reductions while accepting some degree of delayed failure detection (Jin and Servant, 2020, 2022). Such delays constitute the primary cost that arises from potential missed failures. When a build is incorrectly predicted as non-failing and therefore skipped, any failures that would have been detected in that build are instead discovered in a later one, leading to a delay in failure detection. However, obtaining a generic and accurate measurement of actual cost is inherently difficult, if not impossible. The same detection delay may be inconsequential in one setting and highly detrimental in another, depending on factors such as application domain, deployment practices, and risk tolerance. Therefore, the perceived cost/benefit trade-off depends on application criticality, domain, and operational context. Even the detection delay itself cannot always be meaningfully quantified. In our analysis of 14 false negative cases, the failure was fixed before the next build was executed in 13 instances, making it impossible to observe or measure a detection delay, as the failure did not persist long enough to reoccur. In the remaining case, the failure remained unfixed and was detected only in the subsequent build, corresponding to a delay of one build cycle, which in our setting typically amounts to approximately one day.

Beyond identifying failures early, it is critical to pinpoint the specific change responsible for a failure. In projects with hundreds of daily commits, such delays can significantly complicate debugging and root-cause analysis (Hilton et al., 2017). However, quantifying these indirect costs precisely remains challenging. As these effects vary widely across projects and development contexts, we refrain from collapsing them into a single scalar cost metric.

5.5. Threats to validity

Our empirical evaluation is subject to several internal validity threats. First, the observed outcomes can be attributed to external factors that are not captured by features. Projects and developers change over time. We included data from multiple projects to reduce project-specific bias. We also applied consistent data collection procedures and utilize feature-based representations (categorizations) of developers and files to mitigate this risk. Nevertheless, unobserved confounding variables can exist. Second, the study was conducted with a limited number of developers who were available and willing to participate at the time

of data collection. We acknowledge that the sample size is small; however, it reflects the practical constraints of obtaining detailed, developer-specific information in an industrial context. The participants were recruited on a voluntary basis, and informed consent was obtained from each of them prior to data collection. All personal information, including demographic details such as gender and experience, was anonymized to ensure privacy and confidentiality. Finally, we acknowledge the limitations related to dataset size and class imbalance. However, these characteristics reflect the real-world industrial setting in which the study was conducted. Hereby, failures are naturally infrequent and available data are limited. Therefore, we worked with the data that were accessible and representative of actual project conditions. Nevertheless, the imbalanced nature of the dataset provides a realistic evaluation scenario, illustrating how the proposed approach performs under practical conditions.

An external validity threat arises from the fact that all projects in our dataset are from a specific domain, namely Android-based TV software. Although the dataset includes multiple projects, the findings may not generalize to other industries, technologies, or application domains. Observing similar effects in more diverse settings, including different software domains and industry contexts, would be valuable to assess the broader applicability of our results.

Our study is subject to a potential construct validity threat related to how build failures are represented in our dataset. In practice, developers are primarily concerned with novel failures rather than repeated failures caused by the same underlying fault. Since the root causes of failures were not explicitly tracked, some consecutive failures could, in principle, stem from a previously unfixed fault, which might lead to overcounting and affect the interpretation of what constitutes a distinct failure. However, given the long build durations (averaging nearly half a day), it is unlikely that the same fault would persist across multiple builds. Conversely, we also cannot assume that two consecutive failures are always caused by the same issue, as a new fault might be introduced while or after fixing the previous one. To assess this risk on construct validity, we analyzed our dataset to determine the frequency of consecutive failures. We found that such cases account for roughly a quarter of all failures: among the 74 builds that failed, 19 had a failed preceding build, while 55 followed a successful one. A related concern is that Previous Build Status emerged as the most influential feature, which could bias the model toward detecting repeated failures. If the model were a trivial rule that predicted failure whenever the previous build failed and success otherwise, it would systematically miss all novel failures, that is failures with Previous Build Status equal to zero. To evaluate this risk, we examined the true positive predictions in the test set. Of the 12 correctly predicted failures, 8 were preceded by a failed build, while 4 followed a successful one. This indicates that the model is capable of predicting novel failures and does not rely exclusively on a single feature. Overall, while reliance on Previous Build Status helps to explain the model's high precision and limited recall (particularly for novel failures) the results indicate that the model does not collapse into a pure repeated failure detector. Other features meaningfully contribute to its predictions, mitigating (though not eliminating) this construct validity threat.

Our approach focuses on predicting the build status after a collection of commits rather than after each individual commit. As a result, the size of the dataset is significantly smaller compared to studies based on single commits. This introduces a conclusion validity threat regarding the inferences drawn about the relationships between feature sets, ML algorithms, and prediction outcomes. To mitigate this risk, we employed varying test sets and reported multiple evaluation metrics to provide a comprehensive assessment of model performance.

6. Conclusion and future work

In this study, we investigated the application of ML techniques to predict build outcomes in a Continuous Integration (CI) environment within an industrial setting. Our goal was to identify builds that could be safely skipped to reduce CI costs without significantly increasing the risk of undetected failures. We proposed a prediction approach based on collective code changes contributed by multiple developers and extracted resilient features through the categorization of developers and modified files.

Through comprehensive experiments involving various ML algorithms and feature combinations, we observed that ensemble-based algorithms do not significantly outperform traditional models. Among the traditional algorithms, LR yielded the best performance. In terms of feature importance, the previous build status was found to be the most informative. Our results demonstrate high precision but low recall in predicting failed builds, leading to F1 scores of up to 60% when all features are used with LR. While such F1 scores might initially seem low for deployment in an industrial context, the approach remains valuable: it allows hundreds of successful builds to be correctly predicted and skipped, with only a small number of failures potentially being missed. Overall, predicting build status based on collective changes is a challenging task, especially due to class imbalance and small sample size. Nevertheless, it presents a promising opportunity for optimizing CI processes in industrial environments. Unlike prior studies that primarily focus on predicting failures at the level of single commits, our work explores the collective-commit perspective. Because of this fundamental difference in granularity, a direct baseline comparison with previous datasets was not feasible. Instead, we focused on providing a systematic evaluation of diverse models and feature configurations for multi-commit contexts, offering insights complementary to existing research.

Selectively skipping builds provides substantial efficiency gains, reducing build times (by an average of 2.5 h per build), computational resource usage, and energy consumption. We characterize the primary risk as delayed failure detection due to false negative predictions. In nearly all observed cases, such delays could not be meaningfully measured because the failure was fixed before the next build was executed; in the single remaining case, the failure persisted and was detected in the subsequent build cycle. The impact of such delays varies widely depending on application criticality, deployment practices, and organizational risk tolerance. Failure detection delays may also complicate debugging and root-cause analysis, particularly in projects with frequent commits. Accordingly, the suitability of predictive build skipping depends on application criticality, domain, and operational context.

As future work, it would be interesting to leverage time-based ML models such as Long Short-Term Memory (LSTM) to capture the temporal relations in data. One can also replicate the same experiments for other projects in various application domains.

CRediT authorship contribution statement

Ömer Özdemir: Writing – original draft, Validation, Software, Resources, Investigation, Data curation, Conceptualization; **Reyhan Aydoğan:** Writing – review & editing, Writing – original draft, Validation, Supervision, Software, Methodology, Investigation; **Hasan Sözer:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Methodology, Investigation, Conceptualization.

Data availability

The anonymized dataset has been made available: http://doi.org/10.6084/m9.figshare.28930103

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Abdalkareem, R., Mujahid, S., Shihab, E., 2020. A machine learning approach to improve the detection of CI skip commits. IEEE Trans. Software Eng. 47 (12), 2740–2754.

Abdalkareem, R., Mujahid, S., Shihab, E., Rilling, J., 2019. Which commits can be CI skipped? IEEE Trans. Software Eng. 47 (3), 448–463.

Abid-Althaqafi, N.R., Alsalamah, H.A., 2024. The effect of feature selection on the accuracy of X-platform user credibility detection with supervised machine learning. Electronics 13, 205.

Ananthanarayanan, S., Ardekani, M.S., Haenikel, D., Varadarajan, B., Soriano, S., Patel, D., Adl-Tabatabai, A.R., 2019. Keeping master green at scale. In: Proceedings of the 14th EuroSys Conference, pp. 1–15.

Arachchi, S., Perera, I., 2018. Continuous integration and continuous delivery pipeline automation for agile software project management. In: Proceedings of the 4th Moratuwa Engineering Research Conference, 156–161.

Arani, A.K., Zahedi, M., Le, T.H.M., Babar, M.A. , 2023. Sok: machine learning for continuous integration. In: Proceedings of the IEEE/ACM International Workshop on Cloud Intelligence & AIOps, pp. 8–13.

Benjamin, J., Mathew, J., 2025. Enhancing continuous integration predictions: a hybrid LSTM-GRU deep learning framework with evolved DBSO algorithm. Computing 107 (1), 9.

Bischl, B., Binder, M., Lang, M., Pielok, T., Richter, J., Coors, S., Thomas, J., Ullmann, T., Becker, M., Boulesteix, A.-L., et al., 2023. Hyperparameter optimization: foundations, algorithms, best practices, and open challenges. Wiley Interdiscip. Rev. Data Min. Knowl. Discov. 13 (2), e1484.

Bisong, E., Tran, E., Baysal, O., 2017. Built to last or built too fast? Evaluating prediction models for build times. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). IEEE, pp. 487–490.

Camacho, N.G., 2024. Unlocking the potential of AI/ML in devsecops: effective strategies and optimal practices. J. Artif. Intell. Gener. Sci. 3 (1), 106–115.

Chamorro-Atalaya, O., Alvarado-Bravo, N., Aldana-Trejo, F., Poma-García, C., Aliaga-Valdez, C., Peralta-Eugenio, G., Jala, A.A.T., 2022. Supervised learning through k-nearest neighbor, used in the prediction of university teaching performance. Indones. J. Electr. Eng. Comput. Sci. 27, 1625.

Charte, F., Rivera, A., del Jesus, M.J., Herrera, F., 2016. On the impact of dataset complexity and sampling strategy in multilabel classifiers performance. In: Proceedings of the 11th International Conference on Hybrid Artificial Intelligent Systems, 500–511.

Chelvan, M., Perumal, K., 2017. On the relationship between feature selection stability and change in statistical properties of datasets. Int. J. Comput. Appl. 7, 46–56.

Chen, B., Chen, L., Zhang, C., Peng, X., 2020. 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, pp. 42–53.

Dahouda, M.K., Joe, I., 2021. A deep-learned embedding technique for categorical features encoding. IEEE Access 9, 114381–114391.

Das, P., Sadhu, A.K., Konar, A., Bhattacharya, B.S., Nagar, A.K., 2015. Adaptive parameterized adaboost algorithm with application in EEG motor imagery classification. In: Proceedings of the International Joint Conference on Neural Networks, 1–8.

Donca, I.-C., Stan, O.P., Misaros, M., Gota, D., Miclea, L., 2022. Method for continuous integration and deployment using a pipeline generator for agile software projects. Sensors 22 (12), 4637.

Fabris, F., Doherty, A., Palmer, D., De Magalhaes, J.P., Freitas, A.A., 2018. A new approach for interpreting random forest models and its application to the biology of ageing. Bioinformatics 34 (14), 2449–2456.

Fitzgerald, B., Stol, K.J., 2017. Continuous software engineering: a roadmap and agenda. J. Syst. Software 123, 176–189.

Gligoric, M., Eloussi, L., Marinov, D., 2015. Practical regression test selection with dynamic file dependencies. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, pp. 211–222.

Gomiasti, F.S., Warto, W., Kartikadarma, E., Gondohanindijo, J., Setiadi, D.R.I.M., 2024. Enhancing lung cancer classification effectiveness through hyperparameter-tuned support vector machine. J. Comput. Theories Appl. 1, 396–406.

Grano, G., Titov, T.V., Panichella, S., Gall, H.C., 2019. Branch coverage prediction in automated testing. J. Software Evol. Process 31 (9), e2158.

Guzmán, E., Fischer, R.A.L., Kok, J., 2023. Mind the gap: gender, micro-inequities and barriers in software development. Empirical Software Eng. 29 (17), 1–25.

Hilton, M., Nelson, N., Tunnell, T., Marinov, D., Dig, D., 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 197–207.

Hong, Y., Tantithamthavorn, C., Pasuksmit, J., Thongtanunam, P., Friedman, A., Zhao, X., Krasikov, A., 2024. Practitioners' challenges and perceptions of CI build failure predictions at Atlassian. In: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, pp. 370–381.

Hundsdorfer, S., Würsching, R., Pretschner, A., 2025. RustyRTS: regression test selection for rust. In: Proceedings of the 18th IEEE Conference on Software Testing, Verification and Validation, pp. 338–348.

Jin, X., Servant, F., 2020. A cost-efficient approach to building in continuous integration. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 13–25.

Jin, X., Servant, F., 2022. Which builds are really safe to skip? Maximizing failure observation for build selection in continuous integration. J. Syst. Software 188, 111292.

Jin, X., Servant, F., 2023. HybridCISave: a combined build and test selection approach in continuous integration. ACM Trans. Software Eng. Methodol. 32 (4).

Kamath, D.M., Fernandes, E., Adams, B., Hassan, A.E., 2024. On combining commit grouping and build skip prediction to reduce redundant continuous integration activity. Empirical Software Eng. 29 (6), 145.

Kawalerowicz, M., Madeyski, L., 2023. Continuous build outcome prediction: an experimental evaluation and acceptance modelling. Appl. Int. 53 (8), 8673–8692.

Kerzazi, N., Khomh, F., Adams, B., 2014. Why do automated builds break? An empirical study. In: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, pp. 41–50.

Khan, M.A., Azim, A., Liscano, R., Smith, K., Chang, Y.-K., Seferi, G., Tauseef, Q., 2025. ML-based test case prioritization: a research and production perspective in CI environments. In: Proceedings of the 18th IEEE Conference on Software Testing, Verification and Validation, pp. 476–486.

Kola-Olawuyi, A., Weeraddana, N.R., Nagappan, M., 2024. The impact of code ownership of devops artefacts on the outcome of devops CI builds. In: Proceedings of the 21st International Conference on Mining Software Repositories, pp. 543–555.

Kondareddy, A., Azad, S., Singh, A., Henderson, T.A.D., 2025. Speculative testing at google with transition prediction. In: Proceedings of the 18th IEEE Conference on Software Testing, Verification and Validation, pp. 510–521.

Lee, S., Hong, S., Yi, J., Kim, T., Kim, C.-J., Yoo, S., 2019. Classifying false positive static checker alarms in continuous integration using convolutional neural networks. In: Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification, pp. 391–401.

Liang, P., 2024. Automating the training and deployment of models in MLOps by integrating systems with machine learning. Appl. Comput. Eng. 67 (1), 1–7.

Linares-Vásquez, M., McMillan, C., Poshyvanyk, D., Grechanik, M., 2014. On using machine learning to automatically classify software applications into domain categories. Empirical Software Eng. 19, 582–618.

Machalica, M., Samylkin, A., Porth, M., Chandra, S., 2019. Predictive test selection. In: Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, pp. 91–100.

Marijan, D., 2023. Comparative study of machine learning test case prioritization for continuous integration testing. Software Qual. J. 31 (4), 1415–1438.

Marijan, D., Sagar, S., 2018. Devops enhancement with continuous test optimization. In: Proceedings of the 30th International Conference on Software Engineering and Knowledge Engineering, pp. 535–536.

Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R., Micco, J., 2017. Taming google-scale continuous testing. In: Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track, p. 233–242.

Murphy-Hill, E., Jaspan, C., Egelman, C., Cheng, L., 2022. The pushback effects of race, ethnicity, gender, and age in code review. Commun. ACM 65 (3), 52–57.

Pan, C., Pradel, M., 2021. Continuous test suite failure prediction. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 553–565.

Pan, R., Bagherzadeh, M., Ghaleb, T.A., Briand, L., 2021. Test case selection and prioritization using machine learning: a systematic literature review. Empirical Software Eng. 27 (29), 1573–7616.

Pargent, F., Pfisterer, F., Thomas, J., Bischl, B., 2022. Regularized target encoding outperforms traditional methods in supervised machine learning with high cardinality features. Comput. Stat. 37, 2671–2692.

Qadrini, L., Hikmah, H., Tande, E., Presda, I., Maghfirah, A.A., Nilawati, N., Handayani, H., 2024. Ensemble resampling support vector machine, multinomial regression to multiclass imbalanced data. BAREKENG: J. ILMU Matematika Dan Terapan 18, 0269–0280.

Rothermel, G., Harrold, M.J., 1994. Selecting regression tests for object-oriented software. In: Proceedings of the International Conference on Software Maintenance, pp. 14–25.

Saidani, I., Ouni, A., Mkaouer, M.W., 2022. Improving the prediction of continuous integration build failures using deep learning. Autom. Software Eng. 29 (1).

Schwendner, D., Jungwirth, M., Gruber, M., Knoche, M., Merget, D., Fraser, G., 2025. Practical pipeline-aware regression test optimization for continuous integration. In: Proceedings of the 18th IEEE Conference on Software Testing, Verification and Validation, pp. 371–381.

Sharif, A., Marijan, D., Liaaen, M., 2021. Deeporder: deep learning for test case prioritization in continuous integration testing , 525–534.

Shi, E., Sun, L., Xu, J., Zhang, S., 2020. Multilabel feature selection using mutual information and ML-relieff for multilabel classification. IEEE Access 8, 145381–145400.

Shimmi, S., Rahimi, M., 2024. On association of code change types and CI build failures in software repositories. Eur. J. Inf. Technol. Comput. Sci. 4 (2), 1–15.

Silva, G., Bezerra, C., Uchôa, A., Machado, I., 2023. What factors affect the build failures correction time? A multi-project study. In: Proceedings of the 17th Brazilian Symposium on Software Components, Architectures, and Reuse, pp. 41–50.

Singh, N., 2023. CI/CD pipeline for web applications. Int. J. Res. Appl. Sci. Eng. Technol. 11 (5), 5218–5226.

Son, J., An, G., Hong, J., Yoo, S., 2025. Evaluating machine learning-based test case prioritization in the real world: an experiment with SAP HANA. In: Proceedings of the 18th IEEE Conference on Software Testing, Verification and Validation, pp. 522–532.

Sozer, H., Gebizli, C.S., 2017. Model-based testing of digital TVs: an industry-as-laboratory approach. Software Qual. J. 25 (4), 1185–1202.

Stahl, D., Bosch, J., 2014. Modeling continuous integration practice differences in industry software development. J. Syst. Software 87, 48–59.

Ståhl, D., Mårtensson, T., Bosch, J., 2017. The continuity of continuous integration: correlations and consequences. J. Syst. Software 127, 150–167.

Suhas, T.M., K, S.N., 2021. Continuous integration and continuous deployment with jenkins in c++ software development. J. Univ. Shanghai Sci. Technol. 23, 805–810.

Tian, Y., Zhang, Y., Stol, K.-J., Jiang, L., Liu, H., 2022. What makes a good commit message? In: Proceedings of the 44th International Conference on Software Engineering, p. 2389–2401.

Vale, G., Costa, H., Apel, S., 2023. Predicting merge conflicts considering social and technical assets. Empirical Software Eng. 29 (24), 1–33.

Vassallo, C., Schermann, G., Zampetti, F., Romano, D., Leitner, P., Zaidman, A., Di Penta, M., Panichella, S., 2017. A tale of CI build failures: an open source and a financial organization perspective. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 183–193.

Wang, X., Yan, Y., Ma, X., 2020. Feature selection method based on differential correlation information entropy. Neural Process. Lett. 52, 1339–1358.

Xia, J., Li, Y., 2017. Could we predict the result of a continuous integration build? An empirical study. In: Proceedings of the IEEE International Conference on Software Quality, Reliability and Security Companion. IEEE, pp. 311–315.

Xie, Z., Li, M., 2018. Cutting the software building efforts in continuous integration by semi-supervised online AUC optimization. In: IJCAI, pp. 2875–2881.

Yaacob, W. F.W., Nasir, S. A.M., Yaacob, W. F.W., Sobri, N.M., 2019. Supervised data mining approach for predicting student performance. Indonesian J. Electr. Eng. Comput. Sci. 16 (3), 1584–1592.

Yaraghi, A.S., Bagherzadeh, M., Kahani, N., Briand, L.C., 2022. Scalable and accurate test case prioritization in continuous integration contexts. IEEE Trans. Software Eng. 49 (4), 1615–1639.

Zhang, K., Geng, W., Zhang, S., 2018. Network-based logistic regression integration method for biomarker identification. BMC Syst. Biol. 12, 113–122.