BuildFast: History-Aware Build Outcome Prediction for Fast Feedback and Reduced Cost in Continuous Integration

Bihuan Chen
School of Computer Science and Shanghai Key Laboratory of Data Science
Fudan University
Shanghai, China

Linlin Chen
School of Computer Science and Shanghai Key Laboratory of Data Science
Fudan University
Shanghai, China

Chen Zhang
School of Computer Science and Shanghai Key Laboratory of Data Science
Fudan University
Shanghai, China

Xin Peng
School of Computer Science and Shanghai Key Laboratory of Data Science
Fudan University
Shanghai, China

ABSTRACT

Long build times in continuous integration (CI) can greatly increase the cost in human and computing resources, and thus become a common barrier faced by software organizations adopting CI. Build outcome prediction has been proposed as one of the remedies to reduce such cost. However, the state-of-the-art approaches have a poor prediction performance for failed builds, and are not designed for practical usage scenarios. To address the problems, we first conduct an empirical study on 2,590,917 builds to characterize build times in real-world projects, and a survey with 75 developers to understand their perceptions about build outcome prediction. Then, motivated by our study and survey results, we propose a new history-aware approach, named BuildFast, to predict CI build outcomes cost-efficiently and practically. We develop multiple failure-specific features from closely related historical builds via analyzing build logs and changed files, and propose an adaptive prediction model to switch between two models based on the build outcome of the previous build. We investigate a practical online usage scenario of BuildFast, where builds are predicted in chronological order, and measure the benefit from correct predictions and the cost from incorrect predictions. Our experiments on 20 projects have shown that BuildFast improved the state-of-the-art by 47.5% in F1-score for failed builds.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

KEYWORDS

Continuous Integration, Build Failures, Failure Prediction

ACM Reference Format:
Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. 2020. BuildFast: History-Aware Build Outcome Prediction for Fast Feedback and Reduced Cost in Continuous Integration. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416616

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416616

1 INTRODUCTION

Continuous integration (CI) is a software development practice where developers are required to merge their code into a shared repository frequently [15, 19]. Each integration is then verified through an automated build, including dependency installation, code compilation and test case execution. CI brings multiple benefits to a software organization; e.g., it helps to find and fix integration errors earlier and faster, improve developer productivity, improve product quality and reduce development and delivery time [15, 27, 28, 52].

Apart from the benefits, CI can incur high costs [28]. In particular, one of the well-recognized costs in CI is caused by the time duration of a build (a.k.a. build time) [22, 28]. As reported by a recent study on open-source projects, over 40% of the builds have a time duration of over 30 minutes [22], which far exceeds the acceptable build time of 10 minutes [19, 27]. Such long build times greatly increase the cost in human and computing resources, and hence become a common barrier faced by software organizations adopting CI [27, 53].

On the one hand, developers need to wait for a long time to get integration feedback before they continue to work on the verified, latest code base. As a result, developers lose focus and become less productive, which hinders parallel development and overshadows the benefits of CI. On the other hand, computing resources required for running builds are usually in proportion to build times [42]. Hence, a tremendous investment in computing resources (e.g., millions of dollars in Google [28]) is needed to support slow builds.

To reduce such cost in CI, a number of techniques have been proposed from different perspectives. One line of work is focused on developing test case prioritization techniques [9, 16, 36, 39, 60] and test case selection techniques [41, 49] into CI in order to minimize test execution times and speed up builds. Complementary to them, one line of work attempts to skip specific builds (e.g., only having non-source code changes) for saving their whole build times via manual configurations [12, 13] or automated rule-based/learning-based methods [3, 4]. More aggressively, build outcome prediction [18, 25, 26, 33, 44, 47, 56, 59] leverages machine learning techniques to predict build outcomes such that the cost of the builds that are predicted to pass can be reduced. As our empirical study reports that over 70% of the builds are passed (Sec. 2.1), build outcome prediction can potentially lead to high cost reduction.

Despite recent advances, build outcome prediction still suffers the following problems, heavily hindering their practical adoption in CI. First, *failed builds have a poor prediction performance.* Since passed builds often account for a very large portion of all builds in a project, existing techniques tend to predict builds as passed such that they can still yield an overall good performance although they have a poor performance on failed builds. However, failed builds, if incorrectly predicted, can incur high cost. More importantly, existing techniques fail to utilize features that can better capture the characteristics of build failures. Specifically, some techniques [25, 33, 47, 56] leverage social and technical factors to learn prediction models without distinguishing passed and failed builds. More recently, some techniques [26, 44] try to leverage failure-specific features, but in a coarse-grained way (e.g., failure ratio [44] and types of build failures [26]).

Second, *practical usage scenarios are not well considered.* As CI builds arrive in chronological order, a build's outcome should be predicted based on a prediction model learned from its previous builds. Hence, the performance of existing techniques obtained by widely-used cross-validation deviates the performance in practical *online* scenarios. Such negative deviations have also been empirically reported [57]. Moreover, the cost from incorrect predictions and the benefit from correct predictions are important indicators, which are closely relevant to practical usage scenarios. However, without accounting for usage scenarios, existing techniques only measure the prediction performance, but do not systematically analyze the cost and benefit.

In this paper, we first conduct a large-scale empirical study, using 2,590,917 builds from 1,621 GitHub projects, to investigate the time duration of CI builds. Our study is designed to characterize the severity of slow builds in practice and motivate the potential of build outcome prediction. We also conduct an online survey with 75 developers to retrieve first-hand information about developers' perceptions of build outcome prediction. Our survey results reveal consistent concerns with the above two problems of build outcome prediction.

Then, to address the two problems, we propose a history-aware approach, BuildFast, to predict CI build outcomes cost-efficiently and practically. It can help to obtain fast integration feedback and reduce integration cost. Specifically, to address the first problem, we design multiple failure-specific features via digging deep into historical builds, i.e., analyzing build logs and changed files from closely related historical builds. We also develop an adaptive prediction model to switch between two models based on the outcome of the previous build. These two models are separately trained, respectively using a representative set of builds. To address the second problem, we investigate a practical online usage scenario of BuildFast, where the builds are predicted in chronological order, to measure the benefit from correct predictions and the cost from incorrect predictions.

To evaluate the effectiveness and efficiency of BuildFast, we compared BuildFast with three state-of-the-art approaches [26, 44, 59] on 20 Java open-source projects. Our evaluation results have demonstrated that BuildFast can significantly improve the best of the state-of-the-art approaches by 47.5% in F1-score for failed builds without losing F1-score for passed builds. The benefit of BuildFast exceeds its cost; and the average time overhead to predict a build is 1.3 seconds, which is practical. We also demonstrated the contribution of each component in BuildFast to its effectiveness improvement.

In summary, this paper makes the following contributions.

- We conducted an empirical study to characterize build times in real-world projects as well as a developer survey to understand their perceptions on build outcome prediction.
- We proposed a history-aware approach, named BuildFast, to predict CI build outcomes cost-efficiently and practically.
- We conducted large-scale experiments on 20 open-source projects to demonstrate the effectiveness and efficiency of BuildFast.

The rest of the paper is structured as follows. Section 2 presents an empirical study of build times and a developer survey to motivate build outcome prediction. Section 3 introduces the proposed approach in detail. Section 4 evaluates the proposed approach. Section 5 reviews related work before Section 6 draws conclusions.

2 MOTIVATION

In this section, we first present an empirical study of build times in a large corpus of open-source projects and then report our survey with developers to better motivate build outcome prediction.

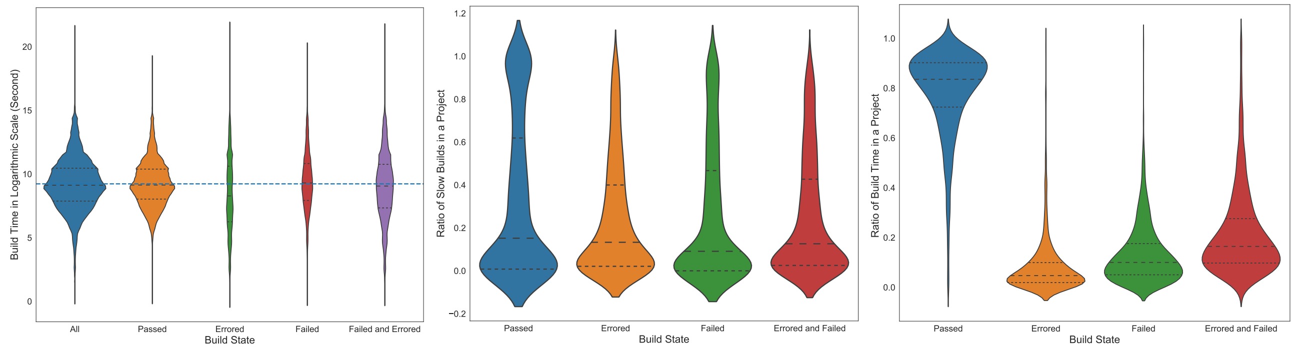
2.1 Build Time Study

Our empirical study of build times is focused on open-source projects due to their publicly available build data. We start with the dataset proposed by Zhang et al. [62], which contains the CI build history of 3,799 open-source Java projects hosted on GitHub. To the best of our knowledge, this is the largest dataset of CI builds. To further ensure that the projects use CI frequently, we exclude the projects that have less than 300 builds, which results in 1,621 projects with a total of 2,612,775 builds. In detail, 2,590,917 (99.2%) of them have a build state of *passed*, *errored* or *failed*. An errored or failed build is called a *broken* build. The difference is that the error that causes an errored build occurs in an earlier build phase than the error that causes a failed build. The remaining 21,858 (0.8%) of builds have uncommon states (i.e., *canceled* and *started*), and thus are not considered in this study.

Using 2,590,917 builds from 1,621 projects, our study is designed to answer the following three research questions.

- **RQ1:** How long is the time duration of passed, errored and failed CI builds across all the projects?
- **RQ2:** How many passed, errored and failed CI builds can be considered as slow in each project?
- **RQ3:** How much build time is consumed by the passed, errored and failed CI builds in each project?

In **RQ1**, we report the overall build time distribution respectively for all passed, errored and failed builds in the 2,590,917 builds. In **RQ2**, we measure for each project the ratio of slow builds among all passed, errored and failed builds respectively, and report the ratio distribution across all projects. Here, we regard a build as slow if it has a build time of more than 10 minutes, because the acceptable build time is 10 minutes [19, 27]. Our results from **RQ1** and **RQ2** aim to characterize the generality and severity of the incurred high costs by build times, and motivate the potential value of build outcome prediction in reducing costs. In **RQ3**, we measure for each project the total build time of all passed, errored and failed builds respectively, analyze its ratio to the



(a) Build Time **(b) Ratio of Slow Builds** **(c) Ratio of Build Time**

Figure 1: Distributions of Build Time, Ratio of Slow Builds and Ratio of Build Time w.r.t. Build States

total build time of all builds in each project, and report the ratio distribution across all projects. Our results from **RQ3** aim to represent the space of cost reduction that can be potentially explored by build outcome prediction. It is also worth mentioning that, of the 2,590,917 builds, 72.2%, 10.5% and 17.3% are passed, errored and failed, respectively. Only about one quarter of the builds are broken; and such imbalance between passed and broken builds can challenge learning-based build outcome prediction (as discussed in Sec. 1).

Overall Build Time (RQ1). Fig. 1a gives the overall build time distribution for all builds, passed builds, errored builds, failed builds and broken builds in violin plot in logarithmic scale. The three lines in each plot respectively denote the upper quartile, the median and the lower quartile. We observe that the median time duration of all builds is 9.3 minutes, which is much shorter than reported in a previous study [22] (i.e., 20 minutes). This large difference could be attributed to the small dataset (i.e., 104,442 builds in 67 projects) of the previous study [22]. We also observe that passed, errored, failed and broken builds have a median time duration of 9.4, 5.2, 10.5 and 8.9 minutes respectively. Except for errored builds, the median time duration of passed, failed and broken builds is very close to the acceptable 10-minute build time [19, 27], denoted by the blue line in Fig. 1a. More specifically, 47.7%, 40.7%, 51.4% and 47.4% of the passed, errored, failed and broken builds are slow builds. Further, one quarter of the passed, errored, failed and broken builds have a time duration of over 22.3, 26.2, 30.2 and 28.9 minutes, while 8.1%, 12.6%, 14.1% and 13.5% of the passed, errored, failed and broken builds even take more than an hour to run. These results demonstrate that CI builds often take a moderately long time to run. In that sense, developers need to wait for a moderately long time to get the integration feedback, which incurs moderately high costs.

Ratio of Slow Builds (RQ2). Fig. 1b shows the distribution of the ratio of slow builds among passed, errored, failed and broken builds across all projects in violin plot. Using the medians, we observe that at least 15.2%, 13.3%, 9.1% and 12.6% of the passed, errored, failed and broken builds are slow in half of the projects. 106 (6.5%) projects have no slow build. At first glance, this result seems to be inconsistent with the result in Fig. 1a (i.e., around half of the builds are slow). This can be explained by the observation that projects with a larger lines of code are more likely to have a larger number of builds and a higher ratio of slow builds, and the difference is statistically significant (i.e., p <0.0001 in Wilcoxon Signed-Rank test). Moreover, using the upper quartiles, we surprisingly observe that more than 61.9%, 40.0%, 46.7% and 42.7% of the passed, errored, failed and broken builds are slow in one quarter of the projects. These results indicate that slow builds are a moderately common problem faced by developers adopting CI, especially in large-scale projects.

Table 1: Survey Questions

Q1	Are you a professional or part-time software developer?
Q2	How large is your company?
Q3	How many years of Java programming experience do you have?
Q4	How many projects have you worked on?
Q5	How many years of CI experience do you have?
Q6	How often does your team trigger CI builds of your projects?
Q7	Are CI builds of your projects time-consuming?
Q8	Would CI build outcome prediction techniques be useful for CI-based software development?
Q9	Why would CI build outcome prediction be useful?
Q10	Why would CI build outcome prediction not be useful?

Ratio of Build Time (RQ3). Fig. 1c presents the distribution of the ratio of build time consumed by the passed, errored, failed and broken builds across all projects in violin plot. We can observe that more than 72.4%, 83.6% and 90.2% of the build time is consumed by passed builds in 75%, 50% and 25% of the projects, whereas at most 9.8%, 16.4% and 27.6% of the build time is consumed by broken builds in 25%, 50% and 75% of the projects. This is consistent with the imbalanced number of passed and broken builds. These results demonstrate that a considerably large amount of time is spent in passed builds, which represents the optimal cost reduction that can be potentially achieved by build outcome prediction (see Sec. 3.4 for a detailed discussion).

2.2 Developer Survey

Our online survey is designed for developers who participated in CI-based software development. Therefore, we randomly select 15,000 developers from 57,939 developers who triggered CI builds in the 1,621 projects used in our empirical study. We send an email to each of the 15,000 developers to introduce the background on build outcome prediction and invite them to take our online questionnaire survey. We promise that their participation would remain confidential, and our analysis and reporting would be based on aggregated responses. In response to our invitation, 75 developers finished the questionnaire within one week (i.e., a participation rate of 0.5%).

As reported in Table 1, our survey consists of 10 questions to learn about all the participants' professional background, CI usage, and perceptions of build outcome prediction. The complete questionnaire with options is available at our website [2].

Professional Background (Q1-Q4). Of all participants, 93.3% are professional developers, and only 6.7% are part-time developers. 45.3% work in a company of more than 100 employees, 12.0% work in a company of 51 to 100 employees, and 42.7% work in a company of up to 50 employees. 42.7% have over 10 years of experience in Java programming, 32.0% have 6 to 10 years, and 25.3% have up to 5 years. 58.7% have participated in the development of more than 15 projects, 5.3% have participated in 11 to 15 projects, and 36.0% have participated in up to 10 projects. We believe that the participants have considerably good experience in parallel software development.

CI Usage (Q5-Q7). 16.0% of the participants have used CI for over 10 years, 41.3% and 34.7% have respectively used CI for 6 to 10 years and 2 to 5 years, and only 8.0% have used CI for less than 2 years. With respect to the build frequency, for 52.0% of the participants, their team averagely triggers a CI build every hour, and for 34.7% of the participants, their team averagely triggers a CI build every minute. 9.3% also comment their team triggers a CI build for every commit. When asked about whether CI builds are time-consuming, 69.3% fully agree, while 26.7% clearly disagree and 4% are not sure.

Perception of Build Outcome Prediction (Q8-Q10). 48.0% of the participants think that build outcome prediction would be useful, but 26.7% think that it would not be useful. 25.3% are not sure mostly because it depends on how it works and how well it works. Further, the participants report three major reasons for the usefulness, i.e., obtaining fast feedback of CI builds (61.3%), saving time overhead of CI builds (50.7%), and accelerating software development (41.3%). On the other hand, the participants also reveal four major reasons for the uselessness, i.e., lacking prediction performance (especially for failed builds) (81.3%), delaying the discovery of bugs due to incorrect predictions (73.3%), lacking explainability (and hence developers do not trust it) (48.0%), and increasing the difficulty of bug fixing due to incorrect predictions (44.0%). Besides, around half of the participants commented that CI builds had to be ran to obtain the build artifacts that would be needed by other projects, especially for passed builds.

Insights. From our survey results, we believe that build outcome prediction has its own potential merit for fast feedback and reduced cost in CI. However, the prediction performance (especially for failed builds) should be taken great care of, as a majority of the developers have concerns on it. The cost and benefit of build outcome prediction should be holistically investigated under a practical usage scenario so that developers can have a holistic view rather than fearing the cost and can have more trust to try build outcome prediction.

Table 2: Features about the Current Build

ID	Feature	Description
C_1	src_churn	# of lines of production code changed
C_2	test_churn	# of lines of test code changed
C_3	**src_ast_diff**	whether production code is changed in AST
C_4	**test_ast_diff**	whether test code is changed in AST
C_5	line_added	# of added lines in all files
C_6	line_deleted	# of deleted lines in all files
C_7	files_added	# of files added
C_8	files_deleted	# of files deleted
C_9	files_modified	# of files modified
C_{10}	src_files	# of production files changed
C_{11}	test_files	# of test files changed
C_{12}	config_files	# of build script files changed
C_{13}	doc_files	# of documentation files changed
C_{14}	class_changed	# of classes modified, added or deleted
C_{15}	met_sig_modified	# of method signatures modified
C_{16}	met_body_modified	# of method bodies modified
C_{17}	met_changed	# of methods added or deleted
C_{18}	field_changed	# of fields modified, added or deleted
C_{19}	import_changed	# of import statements added or deleted
C_{20}	**class_modified**	# of classes modified
C_{21}	**class_added**	# of classes added
C_{22}	**class_deleted**	# of classes deleted
C_{23}	**met_added**	# of methods added
C_{24}	**met_deleted**	# of methods deleted
C_{25}	**field_modified**	# of fields modified
C_{26}	**field_added**	# of fields added
C_{27}	**field_deleted**	# of fields deleted
C_{28}	**import_added**	# of import statements added
C_{29}	**import_deleted**	# of import statements deleted
C_{30}	commits	# of commits included
C_{31}	**fix_commits**	# of bug-fixing commits included
C_{32}	**merge_commits**	# of merge commits included
C_{33}	**committers**	# of unique committers
C_{34}	**by_core_member**	whether a core member triggers the build
C_{35}	**is_master**	whether the build occurs on master branch
C_{36}	time_interval	time interval since the previous build
C_{37}	day_of_week	day of week when the build starts
C_{38}	time_of_day	time of day when the build starts

3 METHODOLOGY

In this section, we first present an overview of BuildFast, and then elaborate each step of BuildFast in detail.

3.1 Overview

Our history-aware build outcome prediction approach uses machine learning techniques, and hence has two basic phases: training phase and prediction phase. In the training phase, BuildFast first extracts three sets of features for each build in a target project (i.e., feature extraction in Sec. 3.2). Then, BuildFast trains a novel adaptive prediction model with the extracted features from a set of builds (i.e., prediction model generation in Sec. 3.3). In the prediction phase, BuildFast extracts the same sets of features for a build under prediction, and uses the trained model to predict its build outcome. Moreover, we systematically explore a practical usage scenario of BuildFast to measure the cost and benefit (i.e., cost-benefit analysis in Sec. 3.4). Although currently implemented for Java projects that use Travis as the CI service, BuildFast can be easily extended to support other programming languages and other CI services by providing specific implementations for feature extraction.

3.2 Feature Extraction

We survey the features adopted in the state-of-the-art approaches [26, 44, 45, 59], and find that their features are mostly directly taken from the TravisTorrent database [7], which is a general-purpose database but is not specialized for build outcome prediction. As a result, high-level coarse-grained features are used without further digging deep

Table 3: Features about the Previous Build

ID	Feature	Description
P_1	pr_state	build state (i.e., passed, errored or failed)
P_2	pr_compile_error	whether compilation error occurs
P_3	pr_test_exception	whether tests throw exceptions
P_4	**pr_tests_ok**	# of tests passed
P_5	**pr_tests_fail**	# of tests failed
P_6	**pr_duration**	overall time duration of the build
P_7	pr_src_churn	# of lines of production code changed
P_8	pr_test_churn	# of lines of test code changed

into the characteristics about build failures. Therefore, we introduce several fine-grained failure-specific features to enhance the existing features based on a detailed analysis of build logs and changed files. Build logs contain historical knowledge about previous build failures [30, 32, 46, 54] which can be learned to predict future build outcomes, while how files are changed in a build can affect its build outcome. In general, we derive the features of a build (i.e., the current build) in three dimensions, i.e., features about the current build, features about the previous build, and features about historical builds.

Features about the Current Build. As the build log of the current build is unavailable (at prediction time), we derive the features from file changes in the current build. Table 2 gives the features with our new features in bold. C_1–C_6 represent line-level changes, where C_3 and C_4 are newly derived to analyze changes at the level of abstract syntax tree (AST) so that formatting changes (e.g., removing a space) that will not fail a build are distinguished. C_7–C_{13} denote file-level changes by distinguishing various kinds of files. C_{14}–C_{19} are class-, method-, field- and import-level changes. However, they fail to distinguish how a class, method, field and import is changed. For example, a deleted class has a higher probability to cause a build failure than an added class because the deleted class might be used but its usage is not accordingly updated. Hence, we derive new features C_{20}–C_{29} to distinguish modified, added and deleted classes, methods, fields and imports. C_{30}–C_{33} denote commit-level knowledge. As a build includes a set of commits, we introduce C_{31}–C_{32} to distinguish the types of commits as bug-fixing and merging commits have a high probability to cause build failures due to potential incomplete fix or merging conflict, and C_{33} to measure the degree of collaboration in the current build as a high degree of collaboration might lead to a high possibility of conflicts. Finally, C_{34}–C_{38} represent the meta data about the current build, i.e., who triggers the current build, and where and when the current build is triggered. Here we introduce C_{34} and C_{35} because core members may less likely to fail a build and developers work more carefully on master branches.

Features about the Previous Build. As build failures often consecutively occur [26], the characteristics of the previous build often serve as a good indicator. Table 3 reports the features about the previous build of the current build with our new features in bold. Specifically, P_1–P_6 are derived from the build log of the previous build. We introduce P_4 and P_5 to measure the degree of failure caused by testing. Intuitively, a larger number of failed tests indicates a higher difficulty to fix the failed build, and thus a higher probability to have a consecutive build failure. P_6 measures the build time of the previous build. A longer build time indicates a higher complexity of the code and thus a higher possibility to fail. P_7 and P_8 measure the degree of code changes in the previous build; and a high degree of code changes may also increase the difficulty to fix the failed build.

Table 4: Features about Historical Builds

ID	Feature	Description
H_1	fail_ratio_pr	% of broken builds in all the previous builds
H_2	**fail_ratio_pr_inc**	increment of fail_ratio_pr at last broken build to fail_ratio_pr at penultimate broken build
H_3	fail_ratio_re	% of broken builds in recent 5 builds
H_4	fail_ratio_com_pr	% of broken builds in all the previous builds that were triggered by the current committer
H_5	fail_ratio_com_re	% of broken builds in recent 5 builds that were triggered by the current committer
H_6	**last_fail_gap**	# of builds since the last broken build
H_7	**consec_fail_max**	maximum of # of consecutive broken builds
H_8	**consec_fail_avg**	average of # of consecutive broken builds
H_9	**consec_fail_sum**	sum of # of consecutive broken builds
H_{10}	**commits_on_files**	# of commits on the files in last 3 months
H_{11}	**file_fail_prob_max**	maximum of the probability of each changed file involved in previous broken builds
H_{12}	**file_fail_prob_avg**	average of the probability of each changed file involved in previous broken builds
H_{13}	**file_fail_prob_sum**	sum of the probability of each changed file involved in previous broken builds
H_{14}	**pr_src_files**	# of production files changed between the latest passed build and the previous build
H_{15}	**pr_src_files_in**	size of the intersection of src_files and pr_src_files
H_{16}	**pr_test_files**	# of test files changed between the latest passed build and the previous build
H_{17}	**pr_test_files_in**	size of the intersection of test_files and pr_test_files
H_{18}	**pr_config_files**	# of build script files changed between the latest passed build and the previous build
H_{19}	**pr_config_files_in**	size of the intersection of config_files and pr_config_files
H_{20}	**pr_doc_files**	# of documentation files changed between the latest passed build and the previous build
H_{21}	**pr_doc_files_in**	size of the intersection of doc_files and pr_doc_files
H_{22}	**log_src_files**	# of production files reported in the build log of the previous build
H_{23}	**log_src_files_in**	size of the intersection of log_src_files and src_files
H_{24}	**log_test_files**	# of test files reported in the build log of the previous build
H_{25}	**log_test_files_in**	size of the intersection of log_test_files and test_files
H_{26}	team_size	size of team contributing in last 3 months

Features about Historical Builds. Table 4 reports the features about historical builds with our new features in bold. In particular, H_1–H_5 represent statistics about previous broken builds by distinguishing all previous builds, the recent five builds, and all previous builds and the recent five builds triggered by the committer of the current build. Here we introduce H_2 to measure the increment between the failure ratio at the last and penultimate broken build. A positive value indicates an increasing trend in build failures. H_6–H_9 are newly introduced to model the distance to the last broken build, and the number of historical consecutive broken builds. A larger value of these features indicate a higher possibility of build failures. H_{10}–H_{25} are newly designed to measure the connection of the files changed in the current build (hereafter referred to as current files for the ease of presentation) to historical builds. In detail, H_{10} measures the number of commits in the last three months that change the current files. A high value of this feature denotes that the current build changes frequently changed files. As frequently changed files often have high potential of bugs [14], it is likely to fail the current build. H_{11}–H_{13} measure the probability that each current file is changed in previous broken builds. The higher the value, the higher possibility to fail the current build. H_{14}, H_{16}, H_{18} and H_{20} measure the number of production, test, build script and documentation files changed between the latest passed build and the previous build. If the previous build is broken, they actually measure the files changed in the previous consecutive broken builds. Therefore, a higher value indicates a higher difficulty to fix previous broken builds. H_{15}, H_{17}, H_{19} and H_{21} measure the intersection between the current files and the changed files in previous consecutive broken builds. The smaller the intersection, the lower possibility to fix previous broken builds. Similarly, H_{22} and H_{24} measure the number of production and test files reported in the build log of the previous build. Such files are listed in build logs mostly due to exceptions in production and test files, and hence indicate the potential root causes of exceptions. Therefore, a higher value indicates a higher difficulty to fix exceptions. H_{23} and H_{25} measure their intersection to current production and test files. A smaller intersection indicates a lower possibility to fix exceptions. H_{26} measures the degree of collaboration in the last three months.

Due to space limitation, we omit the implementation detail of feature extraction. The implementation and a detailed explanation of each feature are available at our website [2].

3.3 Prediction Model Generation

Our prediction model is designed to have two characteristics, i.e., feature selection and adaptive model, to improve the performance.

Feature Selection. As shown in Table 2, 3 and 4, a total of 72 features are introduced from three dimensions. Considering the potentially different characteristics of different projects, we leverage feature selection methods [24] to automatically select the features that contribute most to build outcome prediction for a specific project, instead of manually determining a fixed set of features for all projects. As will be discussed in our evaluation (see Sec. 4.4), different sets of features are selected for different projects.

Adaptive Model. Whether the previous build fails or passes has a different impact on the development activities in the current build. If the previous build fails, developers mainly conduct corrective or preventive activities during the current build. If the previous build passes, developers mainly perform adaptive or perfective activities during the current build. Thus, to learn such differences without confusing the model, we separate our training dataset into two representative datasets; i.e., the first dataset includes the builds whose previous build fails and the second dataset includes the builds whose previous build passes. However, both datasets still have imbalanced data for passed and failed builds, which might hinder the prediction performance for failed builds. To partially solve this problem, we include all the failed builds into the two datasets without distinguishing the build outcome of their previous build; i.e., we further include the failed builds whose previous build passes into the first dataset, and further include the failed builds whose previous build fails into the second dataset. Based on these two datasets, we respectively train a model to predict build outcomes. In this way, in the prediction phase, if the build under prediction has a failed previous build, we use the first model, and if the build under prediction has a passed previous build, we use the second model.

3.4 Cost-Benefit Analysis

Practical usage scenarios have to be analyzed to measure the cost and benefit of build outcome prediction. As CI builds arrive in chronological order, build outcome has to be predicted in an online way in chronological order. Except for [18, 45, 59], all the existing approaches do not predict in chronological order but in a cross-validation way (i.e., a build may be predicted based on a model learned from future builds).

Following this online scenario, build outcome prediction can be used in two scenarios, depending on whether the predicted-to-pass builds are ran or not. First, each build is actually ran. However, team members and project managers could have more confidence to start using the latest code base and conducting project plan without waiting for the build to finish if it is predicted to pass. Hence, computing resources are not reduced; but waiting times are reduced, promoting parallel development and speeding up the release cycle. Second, the predicted-to-fail builds are actually ran, while the predicted-to-pass builds are skipped. Therefore, computing resources are also reduced. In both scenarios, however, developers may work on the buggy code base and need to redo or roll back their work if the prediction is not correct (i.e., predicted-to-pass builds actually fail). In the latter scenario, those integration errors may accumulate for a long time without timely correction, increasing the fixing efforts.

As indicated by our survey (see Sec. 2.2), developers have more concerns on the second aggressive usage scenario, e.g., delaying the discovery of bugs due to incorrect predictions, increasing the difficulty of bug fixing due to incorrect predictions, and requiring the build artifacts that would be needed by other projects. Therefore, we decide to take the first conservative usage scenario. Under this scenario, the benefit comes from the correct prediction for passed builds. Here we use the build time of such builds as the indicator of the *benefit*. As we do not directly pinpoint the root cause of build failures, we consider no benefit from the correct prediction for failed builds. Correspondingly, the cost comes from the incorrect prediction for failed builds. Here we use the build time of such builds as the indicator of the *cost*, and consider no cost from the incorrect prediction for passed builds because developers would wait for the build to complete in the same way as no build outcome prediction approach is used. Finally, we define the *gain* as the difference between benefit and cost.

4 EVALUATION

We have implemented BuildFast in 13.1K lines of Python, Ruby and Java code, using *scikit-learn* [1] for machine learning and ClDiff [29] for code change analysis. We have released the code of BuildFast at our website [2] with the dataset used in our evaluation.

4.1 Evaluation Setup

To evaluate the effectiveness and efficiency of the proposed approach, we compared our approach with three state-of-the-art build

Table 5: Project Statistics

ID	Project	Creation Date	LOC	Stars	Passed Builds	Failed Builds
P1	HikariCP	2013-10	12.5K	6,091	1,388	158
P2	caelum/vraptor4	2013-05	26.6K	322	1,770	227
P3	Checkstyle	2013-08	210.7K	2,940	2,261	235
P4	Achilles	2012-11	48.3K	192	632	94
P5	DSpace	2012-03	172.9K	342	1,791	175
P6	jackson-databind	2011-12	114.1K	1,550	1,720	718
P7	Closure Compiler	2014-04	368.9K	4,017	1,972	171
P8	Graylog	2010-05	175.8K	4,046	5,736	828
P9	jOOQ	2011-04	181.6K	2,181	1,168	583
P10	Optiq	2012-08	141.7K	225	559	152
P11	Kill Bill	2012-10	169.5K	1,609	1,227	1,851
P12	lWicket-Bootstrap	2012-02	1.4K	284	621	695
P13	Vectorz	2012-09	53.5K	137	1,147	64
P14	MyBatis-3	2013-02	58.1K	6,659	824	71
P15	OWL API	2013-02	154.4K	296	1,149	223
P16	Pushy	2013-08	7.2K	759	732	116
P17	QuickML	2011-09	19.3K	218	625	314
P18	Retrofit	2010-09	20.4K	26,096	1,617	299
P19	Rexster	2010-02	23.4K	446	495	93
P20	Weld	2010-08	151.5K	275	1,669	175

outcome prediction approaches, and measured the contribution of each component in BuildFast to its effectiveness, using 20 GitHub Java projects. Our evaluation is designed to answer the following research questions.

- **RQ4**: How is the effectiveness of BuildFast in predicting build outcomes, compared with the state-of-the-art approaches? (Sec. 4.2)
- **RQ5**: How is the efficiency of BuildFast in predicting build outcomes, compared with the state-of-the-art approaches? (Sec. 4.3)
- **RQ6**: How is the contribution of each component in BuildFast to the achieved effectiveness of BuildFast? (Sec. 4.4)

Dataset. We randomly selected 20 projects from the 1,621 projects used in our empirical study (see Sec. 2.1). The statistics about these projects are listed in Table 5, including their creation date, lines of code, the number of stars, and the number of passed and failed builds. We can see that these projects are mostly large in size, and have a long evolution history, which ensures diverse build data. For each project, we split the builds into the training and testing dataset by 3:1.

Comparison Approaches. For **RQ4** and **RQ5**, we selected $\mathbf{BS}_1$ [26], $\mathbf{BS}_2$ [44] and $\mathbf{BS}_3$ [59] as the baselines because $\mathbf{BS}_1$ and $\mathbf{BS}_2$ are the state-of-art approaches that predict in a cross-validation way and $\mathbf{BS}_3$ is the state-of-art approach that predicts in chronological order. For **RQ6**, we ran BuildFast by removing feature selection, by training only one model with all builds, by training two models without including all failed builds, and by excluding our new features.

Evaluation Metrics. Following prior works, we used precision, recall, F1-score and AUC to measure the accuracy of build outcome prediction. We distinguished precision, recall and F1-score for passed builds and failed builds for a detailed comparison across different approaches. We also used benefit, cost and gain (see Sec. 3.4) to measure the cost-efficiency. As $\mathbf{BS}_3$ is designed for optimizing AUC, we can only measure its AUC. In summary, we used accuracy and cost-efficiency to indicate the effectiveness.

Model Configuration. During model generation (see Sec. 3.3), we adopted Chi-Squared Testing [23] to select the top 30 features for our first model, and Information Gain [35] to select the top 25 features for our second model. Besides, we used XGBoost [11] with default parameters as the classifier. This configuration was empirically established as good. For space limitation, detailed comparisons to other configurations are available at our website [2].

4.2 Effectiveness Evaluation (RQ4)

Table 6 presents the results of $\mathbf{BS}_1$, $\mathbf{BS}_2$, $\mathbf{BS}_3$ and BuildFast with respect to the seven effectiveness metrics. The first column shows the build outcome prediction approaches, the second column lists the metrics, and the next twenty columns report the metric values for each project under each approach, and the last column gives the average for precision, recall, F1-score and AUC and the sum for benefit, cost and gain across all projects. The unit of benefit, cost and gain is hour.

Compared with $\mathbf{BS}_1$, BuildFast significantly improved the precision, recall and F1-score for failed builds by 16.5%, 60.2% and 47.5%; and such differences were statistically significant using Wilcoxon Signed-Rank test. Meanwhile, BuildFast slightly improved the F1-score for passed builds. Overall, BuildFast improved F1-score and AUC of $\mathbf{BS}_1$ by 3.9% and 5.5%, with the differences statistically significant. For benefit, cost and gain, there was no statistically significant difference due to the minority of failed builds and the variance of build times. Still, BuildFast had a total gain of 2,131 hours for all projects from one-fourth of the builds (i.e., testing data) with its benefit exceeding its cost. Thus, BuildFast is cost-efficient and can save CI cost.

Compared with $\mathbf{BS}_2$ which was designed to improve the accuracy for failed builds, BuildFast significantly improved all the accuracy metrics except for the recall for failed builds. Overall, BuildFast improved F1-score for failed builds, F1-score for passed builds, F1-score and AUC by 55.2%, 42.0%, 43.0% and 19.5%; and the differences were statistically significant. Due to such a large accuracy improvement for passed builds, BuildFast improved gain by 74.2%.

Compared with $\mathbf{BS}_3$ which was specifically designed to optimize AUC, BuildFast still significantly improved AUC by 27.7%, and the difference was statistically significant. Surprisingly, $\mathbf{BS}_3$ achieved the lowest AUC among the four approaches. This could be attributed to the seven coarse-grained features in their work.

> BuildFast significantly outperformed the best of the state-of-the-art approaches, $\mathbf{BS}_1$, by 47.5% in F1-score for failed builds without losing the F1-score for passed builds. Besides, BuildFast saved a sum of 2,131 hours for all the 20 projects.

4.3 Efficiency Evaluation (RQ5)

Table 7 reports the time overhead of the four approaches. The first column lists the specific approach phases, i.e., training phase and prediction phase. The time overhead of prediction phase is composed of two parts in form of $a + b$, where a denotes feature extraction time and b denotes outcome prediction time. We can see that $\mathbf{BS}_2$ took the longest time for training, i.e., averagely 469.8 seconds for each project, because it used cascaded classifiers, while BuildFast took 6.9 seconds, which was longer than $\mathbf{BS}_3$ but shorter than $\mathbf{BS}_1$. As training is a one-time job, this time overhead is acceptable. On the other hand, BuildFast took 1.3 seconds to extract features for each build, and another 0.004 seconds to obtain the predicted build outcome. While being the slowest due to the large number of used features, BuildFast is still practical for real-world projects.

Table 6: Effectiveness Comparisons to the State-of-the-Art

A.	Metric	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19	P20	All
BS$_1$	Pre.$_f$	.500	.000	.125	1.00	.267	.604	.500	.328	.833	.636	.862	.792	.667	.000	.571	.000	.769	.469	.500	.389	.491
	Pre.$_p$	.896	.879	.918	.943	.964	.899	.932	.925	.937	.795	.525	.824	.952	.889	.884	.797	.824	.855	.795	.954	.869
	Pre.	.852	.773	.852	.947	.934	.851	.900	.870	.910	.758	.779	.804	.936	.790	.845	.637	.807	.791	.720	.924	.834
	Rec.$_f$	.067	.000	.017	.250	.200	.457	.063	.227	.819	.179	.820	.913	.167	.000	.098	.000	.577	.185	.333	.113	.274
	Rec.$_p$	.992	1.00	.989	1.00	.975	.942	.995	.953	.942	.969	.602	.627	.995	1.00	.989	.979	.920	.958	.886	.990	.935
	Rec.	.890	.879	.909	.944	.942	.863	.928	.887	.910	.784	.766	.801	.948	.889	.877	.783	.811	.830	.745	.945	.867
	F1$_f$	.118	.000	.030	.400	.229	.520	.113	.268	.826	.280	.841	.848	.267	.000	.167	.000	.659	.266	.400	.175	.320
	F1$_p$	.941	.936	.952	.971	.970	.920	.962	.939	.939	.873	.561	.712	.973	.941	.934	.879	.869	.904	.838	.972	.899
	F1	.851	.823	.876	.928	.938	.855	.901	.878	.910	.735	.771	.795	.933	.837	.837	.703	.803	.798	.726	.931	.841
	AUC	.619	.568	.610	.873	.683	.808	.636	.773	.883	.785	.763	.853	.878	.515	.815	.713	.844	.788	.813	.643	.743
	Benefit	10.5	7.5	968.9	17.4	68.3	30.7	127.3	431.7	31.2	98.6	559.2	1.7	8.6	49.0	54.4	9.1	8.3	25.6	10.1	214.3	2,732
	Cost	1.1	0.6	14.3	0.6	3.3	1.8	5.3	17.4	1.0	16.8	520.8	0.4	0.5	5.2	4.0	2.5	0.8	2.8	6.3	6.1	611
	Gain	9.4	6.9	954.6	16.8	65.0	28.9	122.0	414.3	30.2	81.8	38.4	1.3	8.1	43.8	50.4	6.6	7.5	22.8	3.8	208.2	2,121
BS$_2$	Pre.$_f$	.102	.500	.072	.075	.107	.157	.228	.144	.271	.318	.848	.707	.088	.000	.128	.667	.596	.305	.500	.143	.298
	Pre.$_p$	.884	.883	.908	.933	.969	.786	.954	.949	.893	.797	.553	.839	.967	.886	.600	.812	.795	.860	.871	.927	.853
	Pre.	.798	.836	.839	.870	.932	.683	.902	.875	.731	.685	.775	.759	.917	.788	.540	.783	.732	.768	.776	.881	.793
	Rec.$_f$	.433	.036	.407	.950	.400	.871	.460	.680	.974	.359	.858	.952	.667	.000	.951	.083	.538	.290	.667	.597	.559
	Rec.$_p$	.531	.995	.528	.056	.849	.092	.879	.593	.076	.766	.534	.388	.583	.975	.211	.990	.830	.868	.771	.278	.590
	Rec.	.520	.879	.518	.123	.830	.219	.848	.601	.310	.671	.777	.731	.588	.867	.129	.808	.738	.772	.745	.294	.598
	F1$_f$	.166	.067	.123	.139	.168	.266	.305	.237	.424	.337	.853	.811	.155	.000	.215	.148	.566	.298	.571	.231	.304
	F1$_p$	.663	.935	.668	.106	.905	.164	.915	.730	.140	.781	.543	.531	.727	.929	.315	.892	.812	.864	.818	.428	.643
	F1	.609	.831	.623	.108	.873	.181	.871	.685	.214	.677	.776	.701	.695	.825	.145	.743	.734	.770	.755	.410	.611
	AUC	.640	.648	.409	.674	.596	.647	.448	.718	.827	.666	.768	.819	.846	.599	.517	.761	.741	.709	.552	.537	.656
	Benefit	5.6	7.4	520.2	0.9	61.7	3.1	115.3	267.8	2.5	82.9	492.1	1.0	4.9	47.6	0.5	9.2	7.1	22.9	8.8	45.8	1,707
	Cost	0.4	0.6	8.5	0.0	3.0	0.5	3.0	5.9	0.1	15.5	431.5	0.3	0.2	5.2	0.1	2.4	0.7	2.5	2.1	1.9	484
	Gain	5.2	6.8	511.7	0.9	58.7	2.6	112.3	261.9	2.4	67.4	60.6	0.7	4.7	42.4	0.4	6.8	6.4	20.4	6.7	43.9	1,223
BS$_3$	AUC	.389	.699	.560	.020	.704	.542	1.00	.970	.717	.167	.574	.293	1.00	.899	.630	.135	.531	.949	.859	.636	.614
BuildFast	Pre.$_f$	.600	.381	.227	.692	.263	.674	.200	.701	.819	.621	.870	.871	.889	.333	.400	.833	.759	.418	.733	.152	.572
	Pre.$_p$	.899	.905	.922	.957	.966	.898	.932	.935	.936	.848	.545	.945	.980	.894	.883	.870	.900	.895	.969	.957	.902
	Pre.	.866	.842	.864	.937	.936	.862	.879	.913	.906	.795	.789	.900	.975	.832	.822	.863	.855	.816	.909	.915	.874
	Rec.$_f$	.100	.286	.085	.450	.250	.443	.095	.314	.819	.462	.828	.971	.667	.067	.098	.417	.788	.492	.917	.226	.439
	Rec.$_p$	.992	.936	.974	.984	.968	.958	.970	.987	.936	.914	.625	.776	.995	.983	.979	.979	.884	.864	.886	.932	.926
	Rec.	.894	.858	.900	.944	.938	.874	.907	.925	.906	.808	.777	.895	.976	.881	.868	.867	.854	.802	.894	.895	.883
	F1$_f$	.171	.327	.123	.545	.256	.534	.129	.434	.819	.529	.848	.918	.762	.111	.157	.556	.774	.452	.815	.182	.472
	F1$_p$	.943	.920	.947	.970	.967	.927	.951	.960	.936	.880	.582	.852	.988	.937	.928	.922	.892	.879	.925	.944	.913
	F1	.858	.849	.879	.939	.937	.863	.891	.912	.906	.798	.782	.892	.975	.845	.831	.848	.854	.808	.897	.905	.874
	AUC	.683	.717	.654	.713	.755	.797	.711	.830	.907	.793	.795	.939	.898	.672	.737	.880	.885	.839	.924	.554	.784
	Benefit	10.5	6.9	954.0	17.1	67.7	31.3	124.2	445.4	31.0	92.9	576.9	2.0	8.6	48.2	53.9	9.0	8.1	23.4	10.2	201.9	2,723
	Cost	1.0	0.4	14.0	0.5	3.2	2.0	4.8	14.1	1.0	12.6	520.0	0.1	0.2	4.8	3.9	1.8	0.5	1.7	0.0	5.5	592
	Gain	9.5	6.5	940.0	16.6	64.5	29.3	119.4	431.3	30.0	80.3	56.9	1.9	8.4	43.4	50.0	7.2	7.6	21.7	10.2	196.4	2,131

Table 7: Efficiency Comparisons to the State-of-the-Art

Phase	**BS$_1$**	**BS$_2$**	**BS$_3$**	BuildFast
Training (sec)	9.4	469.8	0.4	6.9
Prediction (sec)	0.2 + 0.001	0.1 + 0.002	0.1 + 0.001	1.3 + 0.004

BuildFast took 6.9 seconds for training, and 1.3 seconds to predict for a build, which was acceptable for practical usages.

4.4 Ablation Study (RQ6)

Table 8 shows the result of our ablation study to measure the contribution of various settings in BuildFast to the effectiveness in Sec. 4.2.

Removing Feature Selection. BuildFast had a degradation in almost all the accuracy metrics after removing feature selection. Significantly, the precision for failed builds decreased by 9.7% from 0.572 to 0.516 , and the recall for passed builds decreased by 6.5% from 0.926 to 0.866. Overall, F1-score had a degradation of 5.0%. There was no significant difference for AUC, benefit, cost and gain. These results show that feature selection contributes to the improved accuracy for both failed and passed builds by selecting representative features.

Training One Model with All Builds. When only one model was trained in BuildFast with all builds, BuildFast suffered a significant degradation in all the precision, recall, F1-score and AUC metrics except for the recall for failed builds. Overall, F1-score for failed builds, F1-score for passed builds, F1-score, and AUC decreased by 20.8%, 20.3%, 18.2% and 5.5%. Because of such a large degradation for passed builds, gain decreased by 11.0%. These results indicate that our adoption of two models greatly contributes to accuracy and cost-efficiency by learning specialized knowledge from distinguishable build data.

Training Two Models without All Failed Builds. When we did not include all failed builds into the training data of the two separate models, BuildFast had a degradation in all metrics. Significantly,

Table 8: Contributions of Each Component in BuildFast

A.	Metric	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19	P20	All
BuildFast without Feature Selection	Pre.$_f$	.400	.119	.556	.421	.556	.221	.146	.685	.814	.800	.885	.787	.778	.000	.385	.846	.695	.444	.667	.123	.516
	Pre.$_p$	.896	.876	.923	.952	.967	.911	.932	.933	.939	.803	.617	.909	.975	.885	.885	.879	.895	.890	.886	.960	.896
	Pre.	.841	.785	.893	.913	.949	.798	.875	.910	.906	.802	.819	.835	.964	.787	.822	.872	.832	.816	.830	.917	.858
	Rec.$_f$	.067	.571	.085	.400	.250	.757	.111	.291	.828	.205	.865	.962	.583	.000	.122	.458	.788	.452	.667	.339	.440
	Rec.$_p$	.988	.417	.994	.956	.991	.481	.949	.987	.933	.984	.659	.597	.990	.967	.972	.979	.839	.888	.886	.869	.866
	Rec.	.886	.435	.919	.914	.959	.526	.889	.923	.906	.802	.814	.819	.967	.859	.865	.875	.823	.815	.830	.841	.833
	F1$_f$	.114	.196	.147	.410	.345	.342	.126	.408	.821	.327	.875	.866	.667	.000	.185	.595	.739	.448	.667	.180	.423
	F1$_p$	.939	.565	.957	.954	.979	.629	.941	.959	.936	.884	.637	.721	.983	.924	.926	.926	.866	.889	.886	.912	.871
	F1	.849	.520	.890	.913	.952	.582	.882	.909	.906	.754	.816	.809	.965	.822	.833	.860	.826	.816	.830	.874	.830
	AUC	.739	.505	.718	.807	.771	.721	.706	.825	.903	.804	.836	.924	.840	.560	.757	.821	.861	.835	.917	.608	.773
	Benefit	10.5	3.1	973.2	16.6	69.4	15.5	122.5	444.9	30.9	101.0	611.7	1.5	8.5	47.2	53.5	9.0	8.0	23.9	10.2	182.8	2,744
	Cost	1.1	0.3	13.8	0.5	3.2	0.8	5.2	14.9	1.0	17.2	392.1	0.2	0.3	5.2	3.9	1.6	0.3	1.9	2.1	4.1	470
	Gain	9.4	2.8	959.4	16.1	66.2	14.7	117.3	430.0	29.9	83.8	219.6	1.3	8.2	42.0	49.6	7.4	7.7	22.0	8.1	178.7	2,274
BuildFast with One Model	Pre.$_f$	.417	.145	.148	.500	.039	.556	.113	.163	.265	.392	.870	.608	.333	.000	.400	1.00	.833	.433	.636	.140	.400
	Pre.$_p$	.904	.915	.922	.949	.893	.916	.979	.939	1.00	.909	.574	.000	.989	.888	.883	.828	.828	.885	.861	.956	.851
	Pre.	.851	.822	.858	.916	.856	.857	.917	.868	.809	.788	.797	.370	.952	.789	.822	.862	.830	.810	.804	.914	.825
	Rec.$_f$	.167	.714	.136	.350	.850	.571	.873	.529	1.00	.795	.850	1.00	.833	.000	.098	.167	.577	.419	.583	.226	.537
	Rec.$_p$	.971	.422	.930	.972	.056	.911	.467	.728	.024	.625	.614	.000	.899	.992	.979	1.00	.946	.891	.886	.925	.712
	Rec.	.883	.457	.864	.926	.091	.856	.496	.709	.279	.665	.792	.608	.896	.881	.868	.833	.829	.813	.809	.889	.722
	F1$_f$	.238	.241	.142	.412	.075	.563	.200	.249	.420	.525	.860	.756	.476	.000	.157	.286	.682	.426	.609	.173	.374
	F1$_p$	.937	.577	.926	.960	.106	.914	.632	.820	.047	.741	.593	.000	.942	.937	.928	.906	.883	.888	.873	.940	.728
	F1	.860	.537	.861	.920	.105	.857	.601	.768	.144	.690	.794	.460	.916	.833	.831	.782	.819	.811	.806	.901	.715
	AUC	.644	.591	.634	.740	.589	.791	.695	.693	.904	.774	.807	.789	.853	.578	.749	.855	.830	.833	.898	.583	.741
	Benefit	10.3	3.2	909.5	16.9	3.5	29.7	73.7	328.9	0.7	65.1	574.6	0.0	7.7	48.6	54.1	9.2	8.7	24.1	10.1	196.7	2,375
	Cost	0.9	0.2	13.5	0.6	0.3	1.5	0.5	11.3	0.0	5.1	424.0	0.0	0.1	5.2	4.0	2.2	0.8	2.0	0.6	5.1	478
	Gain	9.4	3.0	896.0	16.3	3.2	28.2	73.2	317.6	0.7	60.0	150.6	0.0	7.6	43.4	50.1	7.0	7.9	22.1	9.5	191.6	1,897
BuildFast without All Failed Data	Pre.$_f$	.429	.316	.189	.636	.278	.636	.194	.662	.826	.708	.872	.855	.778	.333	.333	.917	.737	.414	.692	.152	.548
	Pre.$_p$	.898	.897	.926	.950	.966	.891	.932	.930	.936	.846	.533	.926	.975	.894	.880	.880	.907	.888	.912	.954	.896
	Pre.	.847	.827	.865	.926	.937	.850	.879	.906	.908	.814	.788	.883	.964	.832	.811	.887	.853	.809	.856	.913	.868
	Rec.$_f$	.100	.214	.169	.350	.250	.400	.095	.262	.819	.436	.816	.962	.583	.067	.073	.458	.808	.444	.750	.161	.411
	Rec.$_p$	.984	.936	.934	.984	.971	.956	.969	.987	.939	.945	.636	.746	.990	.983	.979	.990	.866	.875	.886	.951	.925
	Rec.	.886	.849	.871	.937	.940	.865	.906	.920	.908	.826	.772	.877	.967	.881	.865	.883	.848	.803	.851	.910	.878
	F1$_f$	.162	.255	.179	.452	.263	.491	.128	.375	.823	.540	.843	.905	.667	.111	.120	.611	.771	.428	.720	.156	.450
	F1$_p$	.939	.916	.930	.966	.969	.922	.950	.958	.938	.893	.580	.826	.983	.937	.927	.931	.886	.881	.899	.953	.909
	F1	.854	.836	.868	.928	.938	.852	.891	.904	.908	.810	.778	.874	.965	.845	.825	.867	.849	.806	.853	.912	.868
	AUC	.684	.692	.655	.762	.714	.780	.719	.825	.908	.803	.796	.933	.919	.609	.649	.843	.874	.833	.919	.629	.777
	Benefit	10.4	6.9	914.9	17.1	67.2	31.2	123.7	445.4	31.1	96.5	586.5	2.0	8.5	48.2	53.9	9.1	7.9	23.6	10.2	201.3	269,5
	Cost	1.0	0.5	13.4	0.6	3.2	2.0	4.8	15.4	1.0	12.7	534.0	0.2	0.3	4.8	4.0	1.7	0.4	2.0	0.4	5.9	608
	Gain	9.4	6.4	901.5	16.5	64.0	29.2	118.9	430.0	30.1	83.8	52.5	1.8	8.2	43.4	49.9	7.4	7.5	21.6	9.8	195.4	2,087
BuildFast without Our New Metrics	Pre.$_f$	.286	.269	.500	.471	.455	.597	.333	.625	.825	.588	.867	.864	.750	.000	.389	.750	.629	.402	.615	.268	.524
	Pre.$_p$	.895	.898	.919	.952	.967	.910	.930	.925	.934	.807	.576	.852	.970	.885	.890	.861	.873	.873	.882	.956	.888
	Pre.	.828	.822	.885	.917	.945	.859	.887	.898	.905	.756	.795	.859	.958	.787	.827	.839	.795	.795	.814	.921	.855
	Rec.$_f$	.067	.250	.034	.400	.250	.529	.048	.203	.810	.256	.854	.913	.500	.000	.171	.375	.750	.347	.667	.177	.380
	Rec.$_p$	.979	.907	.997	.964	.986	.931	.993	.988	.939	.945	.602	.776	.990	.967	.961	.969	.795	.897	.857	.974	.921
	Rec.	.879	.828	.917	.922	.955	.865	.924	.916	.906	.784	.792	.860	.962	.859	.862	.850	.780	.806	.809	.933	.870
	F1$_f$	.108	.259	.063	.432	.323	.561	.083	.307	.817	.357	.860	.888	.600	.000	.237	.500	.684	.372	.640	.214	.415
	F1$_p$	.935	.902	.957	.958	.977	.920	.960	.955	.936	.871	.589	.812	.980	.924	.924	.912	.832	.885	.870	.965	.903
	F1	.844	.825	.883	.919	.948	.862	.897	.896	.905	.751	.793	.858	.958	.822	.838	.829	.785	.800	.811	.926	.858
	AUC	.657	.591	.604	.838	.767	.813	.730	.823	.906	.750	.800	.893	.871	.608	.761	.842	.853	.824	.850	.730	.776
	Benefit	10.4	6.7	976.2	16.7	69.1	30.4	126.4	445.7	31.1	95.3	559.6	2.0	8.5	47.6	53.6	8.9	7.4	24.1	9.9	209.8	2,739
	Cost	1.1	0.4	14.2	0.4	3.2	1.7	5.3	17.5	1.1	16.1	442.8	0.4	0.4	5.2	3.7	1.8	0.5	2.2	2.1	5.8	526
	Gain	9.3	6.3	962.0	16.3	65.9	28.7	121.1	428.2	30.0	79.2	116.8	1.6	8.1	42.4	49.9	7.1	6.9	21.9	7.8	204.0	2,213

the F1-score for failed builds decreased by 4.7%. This is consistent to our motivation of including all failed builds into the two model training process, i.e., partially solving the unbalanced size of failed builds in order to improve the prediction accuracy for failed builds.

Excluding Our New Features. After we excluded new features, BuildFast had a significant degradation in accuracy metrics for failed builds. Overall, the F1-score for failed builds decreased by 12.1%. This indicates the importance of our new features to model the characteristics of build failures. To further look into the importance of our new features, we analyzed the most important features in our two models across all projects by accumulating a feature's importance value, computed during feature selection, across all the

Table 9: Important Features in Our Two Models

First Model			Second Model		
Feature	Imp.	Proj.	Feature	Imp.	Proj.
pr_state	.147	20	pr_state	.333	20
fail_ratio_pr	.111	1	fail_ratio_com_re	.073	20
log_src_files_in	.072	12	**log_src_files_in**	.067	20
pr_test_exception	.069	20	**file_fail_prob_sum**	.054	2
pr_src_files	.062	9	team_size	.053	14
field_modified	.059	2	files_added	.051	1
last_fail_gap	.055	9	class_changed	.050	1
pr_src_churn	.048	1	**met_deleted**	.043	2
commits_on_files	.047	3	**file_fail_prob_max**	.040	5
pr_tests_ok	.046	14	**test_ast_diff**	.039	7
file_fail_prob_sum	.044	8	**field_deleted**	.039	6
fail_ratio_com_re	.043	9	**merge_commits**	.039	5
consec_fail_sum	.039	14	src_churn	.037	2
file_fail_prob_max	.037	9	line_deleted	.036	1
pr_duration	.037	8	commits	.036	5
by_core_member	.036	4	**is_master**	.035	10
src_files	.034	5	**import_added**	.034	7
met_body_modified	.034	6	line_added	.034	2
log_src_files	.033	12	**last_fail_gap**	.034	3
import_added	.033	3	**commits_on_files**	.034	8

projects. The top 20 important features for our two models are reported in Table 9, where *Imp.* denotes the accumulated importance value of a feature, and *Proj.* denotes the number of projects that select a feature. We can see that more than half of the important features are newly introduced in this work (highlighted in bold). This indicates the usefulness of our new features. Besides, these important features are actually selected in various number of projects, meaning that different projects select different sets of features. This demonstrates the importance of feature selection.

> Feature selection, adaptive models, and newly introduced features all contribute positively to the achieved effectiveness of BuildFast, especially for failed builds.

4.5 Discussion

We discuss the threats to and limitations of this work.

Threats. First, we designed an online survey with GitHub developers instead of face-to-face interviews because it can allow us to recruit a relatively large number of participants (although the participant rate was low). Second, we decided to not offer compensation but ask participants to voluntarily take the survey. We expected that developers who were really interested in build outcome prediction and well motivated would participate in this survey and thus the survey quality could be improved. Third, BuildFast was only evaluated against open-source projects without developers' feedback. Experiments with industrial projects and developers are needed to better measure the practical usages of BuildFast.

Limitations. First, although BuildFast outperforms the state-of-the-art approaches significantly in prediction accuracy for failed builds, we have to admit that there is still a room for improvements. One potential way is to understand the semantics of code changes by recent advances in deep code representation learning [5], as we only focus on code changes at syntactic level. Second, BuildFast only predicts whether a build fails, but cannot identify the root causes which would be useful for developers to fix the failure in advance. We plan to extend BuildFast to classify a failed build into several root causes (e.g., compilation errors and testing failures).

5 RELATED WORK

We review the most closely related work on build prediction, cost reduction in CI, empirical studies about CI, and defect prediction.

Build Prediction. Hassan and Zhang [25] used decision trees to predict build outcome with combined features related to social, technical, coordination and prior-build factors. Their model correctly predicted 69% of the failed builds and 95% of the passed builds on a large project at the IBM Toronto Labs. Wolf et al. [56] adopted social network analysis to obtain communication structure measures, and leveraged such measures into a Bayesian classifier to predict the outcome of a build. They achieved precision and recall between 50% and 76% on IBM's Jazz project. Then, Schroter [47] extended Wolf et al.'s work [56] by adding technical dependencies into the social network. Kwan et al. [33] analyzed the effect of social-technical congruence (i.e., the match between the coordination needs established by technical domain and the coordination activities carried out by project members) on build outcome. Their study on the IBM Rational Team Concert project showed that social-technical congruence had a negative effect on integration build success rate. The social factors in these approaches are often organization-specific, greatly hindering the generalizability of predictive models over a wider audience. Instead, BuildFast is specifically designed for CI environment, and thus can be applied to any project as long as it adopts CI.

Finlay et al. [18] used data stream mining techniques based on code metrics (i.e., basic metrics, dependency metrics, complexity metrics, cohesion metrics, and Halstead metrics) to predict build outcome. They achieved 72% accuracy on IBM's Jazz project. As only source code files were included in metric computation, this approach could not predict build failures caused by errors in non-source code files (e.g., configuration files). Recently, Ni and Li [44] used cascaded classifiers to predict build outcome in CI based on file-level metrics from the current and previous build and failure statistics from historical builds. Similarly, Hassan and Wang [26] leveraged metrics from the current and previous build. Differently, they included metrics about failure type of the previous build and coarse-grained code changes in the current build, and did not consider historical builds. Different from such approaches, we extract fine-grained code change features from historical builds. Ni and Li [45] proposed to dynamically adapt a pool of classifiers learned from various projects to a new project that does not have sufficient data of builds. This approach is orthogonal to the previous approaches and our approach, because it reuses the classifiers trained by the previous approaches and our approach. Xia and Li [57] investigated the accuracy of nine classifiers in the online build outcome prediction scenario, and found that the accuracy fell to a fairly low level. Xie and Li [59] targeted the online scenario, and proposed a semi-supervised online AUC optimization method. However, the coarse-grained features hinder its effectiveness. Except for three approaches [18, 45, 59], all the previous build outcome prediction approaches were evaluated in the cross-validation way, and thus they might not work well in the practical online scenario. Instead, BuildFast targets the online scenario. Moreover, apart from the accuracy indicators, we analyze the benefit from correct predictions and the cost of incorrect predictions to systematically evaluate the cost-effectiveness of BuildFast. Recently, Jin and Servant [31] proposed SmartBuildSkip to predict the first builds in a sequence of build failures with a machine learning classifier and then determine that all subsequent builds will fail until it observes a passed build. This approach targets a different usage scenario of BuildFast, and our classifier can be integrated into their approach.

Besides, Bisong et al. [8] developed a predictive model to predict the build time of a build job in CI. McIntosh et al. [40], Xia et al. [58] and Macho et al. [37] used machine learning techniques to predict whether source code changes will induce changes in the build system (i.e., build configuration co-changes). These techniques target a different prediction problem than BuildFast.

Cost Reduction in CI. Apart from build outcome prediction, various techniques have been proposed to reduce cost in CI. For example, to reduce build cost, some plugins [12, 13] are designed into CI services for developers to skip some builds by manually configuring the build process; Abdalkareem et al. [4] proposed a rule-based technique to automatically identify commits that can be CI skipped; followed by a machine learning-based approach [3]; and Gambi et al. [21] developed a novel build system that can lazily retrieve parts of libraries that are needed during the execution of a build target. Tufano et al. [51] proposed to analyze developer's changes and predict whether it impacts the longest critical path, whether it may lead to build time increase and the delta, and the percentage of future builds that might be affected by such changes. To reduce testing cost, Celik et al. [10] consolidated repetitive and expensive setup activities into pre-configured testing virtual machines; and a number of test case prioritization [9, 16, 36, 39, 60] and test case selection [41, 49] have been developed into CI to minimize test execution cost. These techniques are orthogonal to BuildFast as they focus on different aspects in CI. Ideally, they can be combined together to achieve optimal cost reduction.

Empirical Studies about CI. With the widespread adoption of CI, empirical studies have been widely conducted to investigate different aspects of CI, e.g., usage, cost, benefits, barriers and needs when developers use CI [27, 28, 52], type and frequency of build failures in CI [30, 32, 46, 54], build failures caused by compilation [48, 62], testing [6, 34] and static violations in static analysis [61], noise and heterogeneity [20] in historical build dataset [7], characteristics of long build duration [22], anti-patterns in CI [53], and test code evolution in CI [43]. Some studies [22, 28] reported concrete evidence on expensive build cost, and some studies [27, 53] revealed that waiting for builds to finish is a common barrier faced by developers. Therefore, these studies motivate the need for build outcome prediction to save build cost. Besides, studies about build failures [30, 32, 46, 54] shed light on our feature selection.

Defect Prediction. Defect prediction has been widely studied. Generally, defect prediction methods (e.g., [14, 17, 38, 50, 55]) build machine learning models based on different kinds of metrics and predict defects at different granularity levels. As defect prediction mostly focuses on defects in source code files and build failures can be caused by errors in non-source code files, defect prediction techniques cannot directly translate to build outcome prediction in CI.

6 CONCLUSIONS

In this paper, motivated by our empirical study on build times and our developer survey on build outcome prediction, we propose a new history-aware approach, named BuildFast, to predict CI build outcomes cost-efficiently and practically. Our experiments on 20 projects have demonstrated that BuildFast can improve the state-of-the-art approaches by 47.5% in F1-score for failed builds without losing the accuracy for passed builds, and the benefit of BuildFast exceeds its cost, bringing fast feedback and reduced CI cost.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (Grant No. 61802067).

REFERENCES

[1] [n.d.]. *scikit-learn*. Retrieved May 6, 2020 from http://scikit-learn.github.io/stable
[2] [n.d.]. *BuildFast*. Retrieved May 6, 2020 from https://buildfastinci.github.io
[3] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. 2020. A Machine Learning Approach to Improve the Detection of CI Skip Commits. *IEEE Transactions on Software Engineering* (2020).
[4] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. 2019. Which commits can be CI skipped? *IEEE Transactions on Software Engineering* (2019).
[5] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *Comput. Surveys* 51, 4 (2018), 81.
[6] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 356–367.
[7] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 447–450.
[8] Ekaba Bisong, Eric Tran, and Olga Baysal. 2017. Built to last or built too fast? evaluating prediction models for build times. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 487–490.
[9] Benjamin Busjaeger and Tao Xie. 2016. Learning for Test Prioritization: An Industrial Case Study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 975–980.
[10] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. 2016. Build System with Lazy Retrieval for Java Projects. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 643–654.
[11] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.
[12] Travis CI. [n.d.]. Customizing the Build - Skipping a Build. Retrieved February 2, 2020 from https://docs.travis-ci.com/user/customizing-the-build/#skipping-a-build
[13] Cloudbee. [n.d.]. Jenkins Enterprise by CloudBees 14.5 User Guide - Using the Skip Next Build plugin. Retrieved February 2, 2020 from https://docs.huihoo.com/jenkins/enterprise/14/skip.html
[14] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering* 17, 4-5 (2012), 531–577.
[15] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.
[16] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 235–245.
[17] Norman E Fenton and Martin Neil. 1999. A critique of software defect prediction models. *IEEE Transactions on software engineering* 25, 5 (1999), 675–689.
[18] Jacqui Finlay, Russel Pears, and Andy M Connor. 2014. Data stream mining for predicting software build outcomes using source code metrics. *Information and Software Technology* 56, 2 (2014), 183–198.
[19] Martin Fowler. 2000. Continuous Integration. http://martinfowler.com/articles/originalContinuousIntegration.html
[20] Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. 2018. Noise and heterogeneity in historical build data: an empirical study of Travis CI. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 87–97.
[21] Alessio Gambi, Zabolotnyi Rostyslav, and Schahram Dustdar. 2015. Improving Cloud-Based Continuous Integration Environments. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. 797–798.
[22] Taher Ahmed Ghaleb, Daniel Alencar Da Costa, and Ying Zou. 2019. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering* 24, 4 (2019), 2102–2139.
[23] Priscilla E Greenwood and Michael S Nikulin. 1996. *A guide to chi-squared testing*. Vol. 280. John Wiley & Sons.

[24] Isabelle Guyon and André Elisseeff. 2003. An Introduction to Variable and Feature Selection. *Journal of Machine Learning Research* 3, Mar (2003), 1157–1182.
[25] Ahmed E Hassan and Ken Zhang. 2006. Using decision trees to predict the certification result of a build. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. 189–198.
[26] Foyzul Hassan and Xiaoyin Wang. 2017. Change-Aware Build Prediction Model for Stall Avoidance in Continuous Integration. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 157–162.
[27] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 197–207.
[28] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 426–437.
[29] Kaifeng Huang, Bihuan Chen, Xin Peng, Daihong Zhou, Ying Wang, Yang Liu, and Wenyun Zhao. 2018. ClDiff: Generating Concise Linked Code Differences. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 679–690.
[30] Md Rakibul Islam and Minhaz F Zibran. 2017. Insights into continuous integration build failures. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 467–470.
[31] Xianhao Jin and Francisco Servant. 2020. A Cost-efficient Approach to Building in Continuous Integration. In *Proceedings of the 42nd International Conference on Software Engineering*.
[32] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. 2014. Why do automated builds break? an empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. 41–50.
[33] Irwin Kwan, Adrian Schroter, and Daniela Damian. 2011. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *IEEE Transactions on Software Engineering* 37, 3 (2011), 307–324.
[34] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the cost of regression testing in practice: a study of Java projects using continuous integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 821–830.
[35] Changki Lee and Gary Geunbae Lee. 2006. Information gain and divergence-based feature selection for machine learning-based text categorization. *Information processing & management* 42, 1 (2006), 155–165.
[36] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining prioritization: continuous prioritization for continuous integration. In *Proceedings of the 40th International Conference on Software Engineering*. 688–698.
[37] Christian Macho, Shane McIntosh, and Martin Pinzger. 2016. Predicting build co-changes with source code change and commit categories. In *Proceedings of the IEEE 23rd international conference on software analysis, evolution, and reengineering*. 541–551.
[38] Lech Madeyski and Marcin Kawalerowicz. 2017. Continuous defect prediction: the idea and a related dataset. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 515–518.
[39] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*. 540–543.
[40] Shane Mcintosh, Bram Adams, Meiyappan Nagappan, and Ahmed E Hassan. 2014. Mining co-change information to understand when build changes are necessary. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. 241–250.
[41] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. 233–242.
[42] John Micco. 2013. Continuous integration at google scale. https://eclipsecon.org/2013/sites/eclipsecon.org.2013/files/2013-03-24%20Continuous%20Integration%20at%20Google%20Scale.pdf
[43] Gustavo Sizilio Nery, Daniel Alencar da Costa, and Uirá Kulesza. 2019. An Empirical Study of the Relationship between Continuous Integration and Test Code Evolution. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. 426–436.
[44] Ansong Ni and Ming Li. 2017. Cost-effective build outcome prediction using cascaded classifiers. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 455–458.
[45] Ansong Ni and Ming Li. 2018. Poster: ACONA: Active Online Model Adaptation for Predicting Continuous Integration Build Failures. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering: Companion*. 366–367.
[46] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 345–355.
[47] Adrian Schroter. 2010. Predicting build outcome with developer interaction in jazz. In *Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering*. 511–512.
[48] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*. 724–734.
[49] August Shi, Suresh Thummalapenta, Shuvendu K. Lahiri, Nikolaj Bjorner, and Jacek Czerwonka. 2017. Optimizing Test Placement for Module-Level Regression Testing. In *Proceedings of the 39th International Conference on Software Engineering*. 689–699.
[50] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online defect prediction for imbalanced data. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*. 99–108.
[51] Michele Tufano, Hitesh Sajnani, and Kim Herzig. 2019. Towards predicting the impact of software changes on building activities. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results*. 49–52.
[52] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. 805–816.
[53] Carmine Vassallo, Sebastian Proksch, Harald C Gall, and Massimiliano Di Penta. 2019. Automated reporting of anti-patterns and decay in continuous integration. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering*. 105–115.
[54] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. 2017. A tale of CI build failures: An open source and a financial organization perspective. In *Proceedings of the IEEE international conference on software maintenance and evolution*. 183–193.
[55] Zhiyuan Wan, Xin Xia, Ahmed E Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. 2018. Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering* (2018).
[56] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. 2009. Predicting build failures using social network analysis on developer communication. In *Proceedings of the IEEE 31st International Conference on Software Engineering*. 1–11.
[57] Jing Xia and Yanhui Li. 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*. 311–315.
[58] Xin Xia, David Lo, Shane McIntosh, Emad Shihab, and Ahmed E Hassan. 2015. Cross-project build co-change prediction. In *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering*. 311–320.
[59] Zheng Xie and Ming Li. 2018. Cutting the Software Building Efforts in Continuous Integration by Semi-Supervised Online AUC Optimization. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. 2875–2881.
[60] Shin Yoo, Robert Nilsson, and Mark Harman. 2011. Faster fault finding at Google using multi objective regression test optimisation. In *Proceedings of the 8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*.
[61] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. 334–344.
[62] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. 2019. A large-scale empirical study of compiler errors in continuous integration. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 176–187.