Continuous Test Suite Failure Prediction

Cong Pan*
The Key Laboratory on Reliability and Environmental Engineering Technology
School of Reliability and Systems Engineering
Beihang University
China
cong_pan@buaa.edu.cn

Michael Pradel
Department of Computer Science
University of Stuttgart
Germany
michael@binaervarianz.de

ABSTRACT

Continuous integration advocates to run the test suite of a project frequently, e.g., for every code change committed to a shared repository. This process imposes a high computational cost and sometimes also a high human cost, e.g., when developers must wait for the test suite to pass before a change appears in the main branch of the shared repository. However, only 4% of all test suite invocations turn a previously passing test suite into a failing test suite. The question arises whether running the test suite for each code change is really necessary. This paper presents *continuous test suite failure prediction*, which reduces the cost of continuous integration by predicting whether a particular code change should trigger the test suite at all. The core of the approach is a machine learning model based on features of the code change, the test suite, and the development history. We also present a theoretical cost model that describes when continuous test suite failure prediction is worthwhile. Evaluating the idea with 15k test suite runs from 242 open-source projects shows that the approach is effective at predicting whether running the test suite is likely to reveal a test failure. Moreover, we find that our approach improves the AUC over baselines that use features proposed for just-in-time defect prediction and test case failure prediction by 13.9% and 2.9%, respectively. Overall, continuous test suite failure prediction can significantly reduce the cost of continuous integration.

CCS CONCEPTS

• **Software and its engineering** **Software testing and debugging**.

KEYWORDS

continuous test suite failure prediction, continuous integration, cost model, machine learning

ACM Reference Format:
Cong Pan and Michael Pradel. 2021. Continuous Test Suite Failure Prediction. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460319.3464840

*Parts of this work were done while visiting University of Stuttgart.

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464840

1 INTRODUCTION

Continuous integration automates compiling, building, and testing of software. In recent years, continuous integration has been widely adopted to increase code quality and to release often [17, 54]. The key idea is to perform tests automatically and frequently, e.g., triggered by a fixed schedule or by each commit, allowing developers to identify mistakes after being introduced. The popularity of continuous integration, combined with the huge volume of code and code changes, leads to a significant cost imposed by continuously executing test suites. For example, a project at Microsoft has nearly 65,000 regression test cases, which take about three days to fully execute [1]. As another example, Facebook's mobile code base has tens of thousands of code changes per week that each potentially trigger tens of thousands of tests [30]. Even smaller projects contribute heavily to this cost, simply because there is a substantial overall number of projects, as evidenced by over 205 million projects on GitHub alone.[1] Besides computational costs, continuously executing test suites may also slow down development, e.g., when developers must wait for the test suite to finish before a code change shows up in the main branch of the shared repository.

Existing work on reducing the testing cost in continuous integration focuses on four ideas [1, 56]: (1) Test case selection tries to identify a subset of all available test cases that potentially trigger failures, e.g., based on coverage information [38], dependency analysis [10], or predictive models [1, 30]. (2) Test case prioritization aims at ordering test cases so that failing ones are executed first, e.g., based on coverage information [6]. (3) Test case failure prediction aims at predicting individual test cases as pass or failure [1, 30]. A weakness of these three approaches is to reason about each test case individually, which itself imposes a cost, e.g., to gather coverage information. (4) Test suite selection tries to identify a subset of multiple test suites associated with a code base [8, 49], which assumes that different modules in the code base are associated with different test suites. A weakness of all four existing ideas is that even after selecting and prioritizing tests, about one third of the test cases [30] and 30%–80% of the test suites [8] still remain to be run.

While the cost of continuously executing test suites is high, most of these executions do not reveal any problems. In a dataset we gather from 242 open-source projects, only 4.21% of all 15k test suite invocations turn a previously passing test suite into a failing

[1] https://github.com/search, January 2021

test suite. This cost-benefit ratio raises a question: *Do we really have to run the test suite for all code changes?* Intuitively, different kinds of code changes are more or less likely to trigger a test suite failure, e.g., based on what code elements are changed, how experienced the developer is, and how often the test suite has failed in the past. If it is acceptable to occasionally detect a test failure slightly later than immediately after the code change that triggers it, there is a potential for significant cost savings.

This paper presents *continuous test suite failure prediction*, an approach to use this potential through a model that predicts whether executing the test suite of a project is worthwhile at all. The prediction model uses features of the code change, the test suite, and the development history. Once trained, the model can be hooked into continuous integration to decide when to trigger the execution of the test suite. To better understand if and when our approach will reduce the overall cost, we also present a theoretical cost model of continuous test suite failure prediction. The cost model estimates the overall cost reduction depending on three project-specific parameters: the cost of running a test suite, the cost of late detection of a failure-inducing code change, and the test suite failure rate.

We apply our work to a newly gathered dataset of 15k test suite executions in 242 open-source projects. Our results show that the approach effectively predicts whether the test suite will fail, with an AUC of 0.836. A comparison with baselines that use features proposed for test case failure prediction and just-in-time defect prediction shows that our approach outperforms them by 13.9% and 2.9%, respectively. Finally, our cost model quantifies the theoretical reduction in cost, showing that the approach is worthwhile in a wide range of scenarios.

We envision our approach to be useful both for large-scale organizations with a centralized continuous integration infrastructure, such as Google and Facebook, and for continuous integration platforms that offer their services to open-source and other projects, such as Travis CI. Because there is not publicly available dataset of test suite executions with the features our model learns from, our empirical results focus on open-source projects hosted at Travis CI. Applying and evaluating the approach in a large-scale organization with a centralized continuous integration infrastructure is left for future work.

In summary, the main contributions of this paper are:

- The *problem of continuous test suite failure prediction*, i.e., a new take on the old problem of reducing the cost of continuous integration.
- An *effective prediction model* to address this problem based on a combination of features from past work and *new features*.
- A *theoretical cost model* that helps decide if and when the approach will reduce the overall cost in a specific project or organization.
- A *large-scale dataset* gathered from hundreds of open-source projects, to fuel future research on the problem.[2]

[2]Our dataset and other supplementary information are available at https://zenodo.org/record/4742337.

Table 1: Comparison with related problems.

Problem	Goal	Prediction
Continuous test suite failure prediction	Reduce testing cost by skipping code changes	Whether the entire test suite will fail
Test case failure prediction [1, 30]	Reduce testing cost by skipping test cases	Whether a specific test case will fail
Just-in-time defect prediction [20, 32]	Find potentially bug-inducing code changes	Whether the changed code later gets fixed
Build outcome prediction [14, 19]	Reduce continuous integration cost by skipping builds	Whether a specific build will fail
Test suite selection [8, 49]	Reducing testing cost by skipping test suites	Whether a test suite (out of many) will fail

2 APPROACH

2.1 Terminology and Problem Statement

We start by defining the problem of *continuous test suite failure prediction* and by distinguishing it from related problems. Given a code change c and a test suite s, the problem is to predict whether triggering s as part of continuous integration upon c will pass or fail s.

A *code change* is the result of one or more commits into a shared code repository. We consider the combination of all commits between two invocations of the continuous integration system as a code change. Moreover, we focus on changes that delete, add, or modify at least one source code file, excluding changes that affect only configuration files, third-party libraries, and so forth.

The problem is to predict the *test suite outcome*, summarized as passing or failing the test suite. The entire process triggered by the continuous integration system is called a *build*, which usually includes running the test suite. The build fails if the test suite fails, but it may also fail due to other reasons, e.g., a compilation error, a broken external service, or a broken dependency. We focus on predicting the test suite outcome only and do not attempt to predict any other kind of build failures, in which the test suite may not be executed at all. We also focus on test failures that occur after the previous test suite execution was passing, rather than all test suite failures, because we aim to predict code changes that trigger a new test failure, not code changes that fail to fix an existing test failure.

Table 1 compares continuous test suite failure prediction to four problems studied in prior work. Test case failure prediction also aims at reducing the cost of continuous testing, but by predicting for each possibly executed test case whether it is likely to fail. Instead, we predict whether the entire test suite passes or fails. When our approach successfully predicts a test suite pass, the developers can immediately move on. In contrast, a typical rate of test cases that need to be run with test case failure prediction is one third of all test cases [30], i.e., the developers will still have to wait for at least some minutes, and often much longer [34]. Of course, test case failure prediction and test suite failure prediction can be combined to further reduce costs: When our approach predicts a test suite failure, then querying a test case failure prediction model

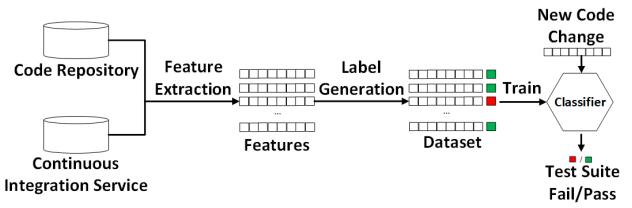


Figure 1: Overview of the approach.

can further reduce the testing effort. We leave combining both techniques for future work.

Our work relates to, but clearly differs from, just-in-time defect prediction, which tries to predict whether a code change introduces a bug that later gets fixed. While defect prediction judges whether a code change leads to a bug at some point in the future, we predict whether a code change causes an immediate test suite failure.

Our work is also similar to build outcome prediction, which aims to predict build failures. While build outcome prediction considers all forms of code changes and all reasons why a build may fail, we narrow down the scope to source code changes and test suite failures. By simplifying the problem compared to build outcome prediction, our approach can find stronger relationships between source code changes and test failures.

Finally, the problem we address is similar to test suite selection [8, 49], which aims at predicting which test suites out of multiple test suites in a code base are worth executing. The key difference is that we assume a project to have a single test suite, and our approach recommends to either run it or to skip it entirely. In contrast, test suite selection assumes a code base that is broken down into modules, which each have their own test suite. Experiments on a dataset of test executions from Google shows that test suite selection keeps 30%–80% of all test suites for execution, i.e., it is still imposing a significant testing effort. Similar to test case failure prediction, future work could combine both approaches: Our work can provide a course-grained, first-step decision, following – if tests should be run at all – by test suite selection to select which modules to test.

2.2 Overview

Our learning-based approach to the test suite failure prediction problem is shown in Figure 1. To learn a predictive model, the approach gathers information from a code repository and a continuous integration service. The information is used to extract a set of features (Section 2.3) for each pair (c, s) of a code change c and test suite s, and to label the pairs as passing or failing (Section 2.4). Given the labeled feature vectors, the approach trains a classification model, which then predicts for previously unseen (c, s) pairs whether the test suite will pass or fail.

2.3 Feature Extraction

Our predictive model uses a set of 44 features listed in Table 2. Some of these features are adapted from related work on just-in-time defect prediction [20, 32, 35, 50] and test case failure prediction [1, 30]. We also present 19 new features, which we find to be suitable for the task of continuous test suite failure prediction. The features are organized into nine categories as below.

Table 2: Features used by the predictive model. The last column indicates whether the feature has been used in defect prediction (DP), test case failure prediction (TC), or is newly proposed in this work (New).

Dimension	Name	Description	Category
Diffusion	NS	Number of modified subsystems	DP
	ND	Number of modified directories	DP
	NF	Number of modified files	DP, TC
	Entropy	Distribution of modified code across files	DP
Size	LA	Lines of code added	DP
	LD	Lines of code deleted	DP
	NC	Number of commits in a code change	TC
	LT	Lines of code in a file before the change	DP
Quality	CL	Number of comment lines in modified files after the code change	New
	CCN	Cyclomatic code complexity of methods in modified files	New
	CCNDelta	Change of cyclomatic code complexity	New
	GC	Number of GitHub commits involved in the code change	DP
AST	NClass	Number of class-related AST nodes	New
	NImport	Number of import-related AST nodes	New
	NType	Number of type-related AST nodes	New
	NConFlow	Number of control-flow-related AST nodes	New
	NExc	Number of exception-related AST nodes	New
	Refactor	Whether a code change only includes refactoring-related AST nodes	New
Change clusters	CD	Number of deleted clusters of code	New
	CI	Number of inserted clusters of code	New
	CM	Number of moved clusters of code	New
	CU	Number of updated clusters of code	New
	DC	Atomic changes count of code deleted	New
	IC	Atomic changes count of code inserted	New
	MC	Atomic changes count of code moved	New
	UC	Atomic changes count of code updated	New
Purpose	FIX	Whether the code change fixes a bug	DP
	NFIX	Number of bug-fixing commits in the code change	DP
Experience	EXP	The number of changes made by the developer before the current change	DP
	REXP	Like EXP, but giving higher weight to recent experience	DP
	SEXP	Like EXP, but for the subsystems modified in the code change	DP
	Awareness	The fraction of all prior changes to the modified subsystems that the developer has participated in	DP
History	NDEV	Number of developers that changed the modified files	DP
	AGE	Average time interval between the last and the current change	DP
	NUC	Number of unique changes to modified files	DP, TC
Test	TP10	Number of times the test suite passes in the last 10 builds that execute test cases	TC
	TF10	Number of times the test suite fails in the last 10 builds that execute test cases	TC
	TFR10	Test suite failure rate for the last 10 builds	TC
	TFR20	Test suite failure rate for the last 20 builds	TC
	TFR40	Test suite failure rate for the last 40 builds	TC
	TLOC	Lines of code of test files in the test suite	TC
	TC	Number of executed test cases in the test report	TC
	TF	Number of test files in the test report	New
	TU	Whether the change updates a test case	New

Diffusion. The rationale behind these features is that changes that affect more subsystems, files, etc. are more likely to cause a

failure, e.g., because the developer may not know all involved files well [35]. For details on computing these features, see [20].

Size. These features are motivated by the observations that larger code changes are more likely to introduce mistakes [35, 36] and that changes in larger files contribute more bugs than changes in smaller ones [23].

Quality. These features measure code quality in terms of comments and code complexity, and how it changes due to a code change. The rationale is that an increase in complexity correlates with post-release defects [37] and that high-quality code is less likely to fail. Both CCN and CCNDelta are calculated as the average function complexity within a file.

AST. These features encode the number of specific types of AST nodes involved in a code change. The rationale is that changing some types of AST nodes is more likely to cause breaking changes than others. Each feature is a combination of several Java AST node types and indicates how many of them a code changes contains.

- Class-related nodes *NClass*: AnonymousClassDeclaration, ClassInstanceCreation, TypeDeclarationClass
- Import-related nodes *NImport*: ImportDeclaration
- Type-related nodes *NType*: ParameterizedType, SimpleType, WildcardType, ArrayType, PrimitiveType, UnionType
- Control-flow-related nodes *NConFlow*: SynchronizedStatement, ConditionalExpression, EnhancedForStatement, DoStatement, ReturnStatement, WhileStatement, IfStatement, ContinueStatement, LabeledStatement, ForStatement
- Exception-related nodes *NExc*: ThrowStatement, TryStatement, AssertStatement
- Refactoring-related nodes *Refactor*: ImportDeclaration, EmptyStatement, Javadoc, PackageDeclaration

To extract the AST node types, we use GumTree [9], a source code differencing tool that generates an edit script for a given pair of old and new code. In total, there are over 100 AST node types, which we select and combine into the above features using the RELIEF feature selection method [22].

Change clusters. These features decompose code changes into atomic changes and cluster them. For example, when deleting two methods, the value of the CD feature is two because the code deletions are decomposed into two clusters. The value of the DC feature is the sum of the number of AST nodes within all clusters in CD. The intuition is that some clusters of code changes are more likely to cause test suite failures. We use GumTree to compute the atomic changes and clusters.

Purpose. These features indicate whether and to what extent a code change attempts to fix a bug. The rationale is that bug-fixing code changes are more likely to introduce new bugs than other kinds of code changes [11, 35, 43]. We include the NFIX feature because code changes may have multiple commits and attempt to fix multiple bugs. FIX and NFIX are calculated by searching keywords related to bug fixes, e.g., "fix", "patch", and "bug" [20].

Experience. These features encode the experience a developer has with the modified code base [20], motivated by the hypothesis that more experienced programmers may cause fewer test failures [35].

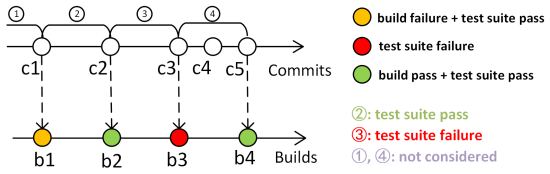


Figure 2: Extraction of labels.

History. These features encode how the files touched in a code change have been changed in the past [20].

Test. These features represent several properties of the test suite and its past behavior. The rationale for considering the size of the test suite is that a larger test suite may be more likely to fail. The rationale for considering the number of passing/failing test suite executions is that frequent failures in the past may suggest frequent failures in the future. We include the TU feature because code changes that change testing code may be more likely to trigger a test suite failure.

The selection of features aims at information that is potentially helpful for the model, yet can be gathered at moderate cost. Another potentially helpful feature would be the coverage achieved by the test suite and how the covered code relates to the changed code. We refrain from including this feature because gathering coverage information during test executions imposes a cost itself, which conflicts with our goal of reducing the overall testing cost.

2.4 Label Extraction

In addition to extracting features for each pair (c, s), we also determine whether the pair is passing or failing the test suite. The features and the label together provide the dataset for training our supervised prediction model.

Figure 2 illustrates the label extraction process with a motivating example. The figure shows two concurrent timelines: the commits into the shared repository at the top and the builds triggered in the continuous integration systems at the bottom. Each build corresponds to a code change, which consists of one or more commits. For example, code change ③ consists of one commit, c3, and it corresponds to build b3; code change ④ consists of two commits, c4 and c5, and corresponds to build b4. We determine the label of each code change based on the following two criteria.

1) Predict test suite outcome. In addition to running a test suite, a build also triggers other processes, e.g., deploying code to a target server. For our label extraction, only the outcome of the test suite matters, and we use it as the label. That is, even if a build fails because of some errors after testing, we extract the label according to the test suite outcome. If a build fails before testing, it is not included in our dataset.

2) Count only first test failures. We consider only test failures that turn a previously passing test suite into a failing test suite, a concept called "first test failures" by Jin et al.[19]. One reason is that developers typically strive to fix the build when it fails, and they will likely want to check whether the new build now passes the test suite. Therefore, there is no need for predicting the test

suite outcome after a build with a failing test suite. Another reason is that, given two subsequent builds with test suite failures, it is hard to tell whether the code change of the first or the second build causes the second test suite failure. Our model aims to reasons about code changes and whether they are the direct cause of test suite failures. The dataset would become noisy if we included cases where a test suite fails, but the failure is not actually caused by the code change. In the following, the term test failure refers to first test failure.

Given these two criteria, we label a code change as "Fail" if a test suite was passing in the previous build, but the test suite fails in the current build. In contrast, we label a code change as "Pass" if the test suite has passed in both the previous and the current build. Any other builds do not obtain a label and are not considered in our dataset.

For example in Figure 2, code change ② is labeled as "Pass" because b2 and the previous build, b1, have passing test suites. Build b3 has a failing test suite, and hence, code change ③ is labeled as "Fail". The other two code changes are ignored: ① because there is no known previous build, and ④ because it follows a build with a failing test suite.

2.5 Prediction Model

After generating the features as well as the labels, the final step is to build a prediction model. Our approach can use any classification model, and Section 5.1 evaluates several commonly used models. The dataset is split into a training set and a test set, and the model is trained using the training set. After training, the model is used to predict whether a code change would trigger a test suite failure.

3 COST MODEL

This section presents a mathematical cost model that theoretically reasons about the cost effectiveness of continuous test suite failure prediction. The goal of the cost model is to help understand the tradeoffs involved in test suite failure prediction and to help decide whether the approach is beneficial in a specific project or organization. We first describe the input parameters of the model (Section 3.1), then present several strategies for handling test suite executions during continuous integration (Section 3.2), and finally provide boundary conditions that indicate when one strategy outperforms another (Section 3.3).

3.1 Input Parameters

The cost model relies on three parameters that are specific to a project or an organization: (i) the cost r of running a test suite, (ii) the cost d of identifying a failure-inducing code change later than immediately after the change, and (iii) the rate f at which code changes cause test suite failures. While determining concrete values for these parameters is beyond the scope of this work, we give some guidelines for determining them and discuss cost estimates given in the literature.

3.1.1 Cost of Running a Test Suite. One relevant cost is the cost r of running a test suite during continuous integration, which is influenced by several factors:

- The computational cost of running test suites, which depends on the size of the project and its test suite. For large organizations with millions of lines of code stored in a single monolithic repository, the computational costs can be surprisingly high. Google's test automation platform (TAP) is reported to run 800 thousand builds and 150 million test runs on an average day [34], which amounts to "millions of dollars just for the computation (not counting the cost of developers who maintain or use TAP)" [17].
- The human cost of maintaining the hardware and software infrastructure for test suite execution. For example, a study reports that about 5% of Facebook engineers are dedicated to the task of developing, improving, and maintaining continuous integration tools [47]. Continuous test suite failure prediction cannot entirely eliminate this cost, but is likely to reduce it because fewer test suite executions require a less sophisticated infrastructure for scheduling and performing tests.
- The human cost of developers who wait for the outcome of a test suite run, e.g., because a code change appears in the main branch of a shared repository only after passing the test suite. Studies on open-source projects report a mean time to schedule and perform a test suite execution of 34 minutes and 20 minutes, respectively [2]. At Google, "unacceptably large delays of up to 9 hours" are not uncommon [34].

3.1.2 Cost of Delayed Detection of a Failure-Inducing Code Change. Another important cost is that of detecting a code change that causes the test suite to fail only later than if one would trigger the test suite immediately after the code change. We represent the overall cost of detecting a single test failure late as a parameter d. This cost is influenced by at least two factors:

- The increased difficulty to localize and fix the root cause of a test suite failure, which may take longer when developers have forgotten the specific context of a code change.
- The negative impact on other developers who are indirectly affected by some not yet diagnosed misbehavior.

A cost estimate about bugs missed during regression testing assumes a cost of 1.5 to 22 person-hours, depending on the severity of the bug [5]. The cost relevant for our model is likely lower, as test suite failures are not completely missed but only found later. Reports of costs experienced during several large software projects state that finding a bug later than immediately during coding can increase the time needed to fix the bug by factors between 2 and 100 [51]. To bound the cost d in practice, developers can run the test suite in regular intervals, and always before releasing the software, to make sure all known test failures are identified and handled.

3.1.3 Test Suite Failure Rate. The third parameter of our cost model is the rate at which the test suite fails on average. We refer to this rate as f. In a dataset gathered from 242 open-source projects (Section 4.1), f is 4.21%.

By focusing on these three input parameters, our model ignores some other costs. For example, we ignore the cost of introducing continuous test suite failure prediction into the continuous integration infrastructure. The reason is that this is a one-time effort for a larger organization or a centralized continuous integration

service. We also ignore the cost of model training and inference, because it is negligible compared to the cost of executing the test suite (Section 5.1).

3.2 Strategies

We use the cost model to reason about five strategies for answering the question if and when to run the test suite during continuous integration. These strategies run the test suite (i) when suggested by our prediction model, (ii) for each code change, (iii) for no code change at all, (iv) periodically based on a fixed schedule, and (v) based on a random decision. The following describes each strategy and what costs it imposes. The description is based on the usual terminology for true positives (TP), false negatives (FN), etc., where TP (TN) means the test suite is correctly predicted to fail (pass), and FP (FN) means the test suite is predicted to fail (pass) but actually passes (fails). The values TN, FN, etc. are the absolute numbers of code changes that are true negatives, false negatives, etc. according to the prediction model. We refer to the overall number of all code changes as $N = TP + TN + FP + FN$. The failure rate is $f = \frac{FN+TP}{N}$.

3.2.1 MODEL Strategy. The MODEL strategy uses the classification model introduced in Section 2 to decide whether to run the test suite for a code change. Specifically, if the classifier predicts the test suite to pass, then running the test suite is skipped. In contrast, if the classifier predicts a failure, then the test suite is executed to enable the developers to debug the test failure as soon as possible. Intuitively, the MODEL strategy tries to balance the two costs r and d.

The overall cost of this strategy is composed of two parts. On the one hand, there is the cost of running the test suite for each predicted failure, i.e., $FP + TP$ times. On the other hand, there is the cost of detecting test failures late whenever the model predicts "Pass" but the test suite would actually fail, which happens FN times.

$$cost_{MODEL} = r \cdot (FP + TP) + d \cdot FN \tag{1}$$

3.2.2 ALL Strategy. This strategy runs the test suite for each code change and is the strategy currently adopted by the majority of continuous integration practitioners. The strategy avoids the cost d but always imposes the cost r.

$$cost_{ALL} = r \cdot N \tag{2}$$

3.2.3 NEVER Strategy. This strategy does not run the test suite for any code change. In practice, using the NEVER strategy boils down to no testing during continuous integration, but only, e.g., before each release. This strategy always imposes the cost d while avoiding the cost r.

$$cost_{NEVER} = d \cdot (FN + TP) \tag{3}$$

3.2.4 PERIOD Strategy. A simple way to balance the costs r and d is to invoke the test suite periodically every k code changes. In this strategy, the cost r occurs inversely proportional to k because a larger k means that the test suite runs less often. The cost d is N times the probability that a failure does not coincide with the periodic test suite execution, i.e., $N \cdot f \cdot (1 - \frac{1}{k})$. Hence, the overall cost of this strategy is:

$$cost_{PERIOD} = r \cdot \lceil \frac{N}{k} \rceil + d \cdot N \cdot f \cdot (1 - \frac{1}{k}) \tag{4}$$

3.2.5 RANDOM Strategy. As a trivial baseline for a classifier that predicts whether to run the test suite for a given code change, we consider a random decision. The decision follows the distribution of failing and passing test suite executions, i.e., it predicts "Fail" with probability f and "Pass" with probability $1 - f$. A well trained classifier should outperform this random decision, and we include it here as a lower bound on what our approach could achieve.

The overall cost of RANDOM is computed similarly to MODEL. The difference is that the predictions are random, and hence, the numbers of true positives, false positives, etc. differ.

$$cost_{RANDOM} = r \cdot (FP_{rand} + TP_{rand}) + d \cdot FN_{rand} \tag{5}$$

Whether the predictions match the actual test suite outcomes depends only on the failure rate of the test suite: $FP_{rand} = N \cdot f \cdot (1 - f)$ because the probability that the random model produces a false positive is the product of the probability of predicting a failure, f, and the probability of seeing an actual success, $1 - f$. In a similar vein, $TP_{rand} = N \cdot f^2$ and $FN_{rand} = N \cdot (1 - f) \cdot f$.

3.3 Comparing Strategies with Boundary Conditions

Expressing the five strategies in our cost model allows for comparing strategies with each other. We compute boundary conditions that express for which fractions $\frac{d}{r}$ one model has less overall cost than another model. For example, the MODEL strategy is beneficial over the ALL strategy when $cost_{MODEL} - cost_{ALL} < 0$. Using Equations 1 and 2, this is equivalent to:

$$\frac{d}{r} < \frac{FN + TN}{FN} \tag{6}$$

Given this boundary condition, one can instantiate the cost model with concrete values to check which strategy is best for the costs r and d and the FN, TN, and FN values in a specific project or organization.

Similar, we can compute the boundary condition for MODEL outperforming the other strategies. We find that MODEL outperforms NEVER when:

$$\frac{d}{r} > \frac{FP + TP}{TP} \tag{7}$$

The MODEL strategy outperforms PERIOD when:

$$\frac{d}{r} \begin{cases} < \frac{\lceil N/k \rceil - FP - TP}{FN - N \cdot f \cdot (1 - 1/k)} & \text{if } FN - N \cdot f \cdot (1 - 1/k) > 0 \\ > \frac{\lceil N/k \rceil - FP - TP}{FN - N \cdot f \cdot (1 - 1/k)} & \text{if } FN - N \cdot f \cdot (1 - 1/k) < 0 \end{cases} \tag{8}$$

Finally, MODEL outperforms RANDOM when:

$$\frac{d}{r} \begin{cases} < \frac{N \cdot f - FP - TP}{FN - N \cdot f \cdot (1 - f)} & \text{if } FN - N \cdot f \cdot (1 - f) > 0 \\ > \frac{N \cdot f - FP - TP}{FN - N \cdot f \cdot (1 - f)} & \text{if } FN - N \cdot f \cdot (1 - f) < 0 \end{cases} \tag{9}$$

The above boundary conditions compare one strategy with another. By combining the boundary conditions of one strategy S compared to all other strategies, we compute the *optimality interval* of S. The interval defines the lowest and highest possible value of $\frac{d}{r}$ where S imposes less overall costs than any other strategy. When instantiating our cost model with concrete values for d and r, one can hence tell whether a strategy is beneficial. Section 5.3 instantiates the cost model based on a real-world dataset to assess under what cost parameters d and r the MODEL strategy is beneficial.

4 EXPERIMENTAL SETUP

4.1 Data Collection

We apply the predictive model (Section 2) and our cost model (Section 3) to a dataset gathered from 242 open-source projects. The workflow for gathering this dataset consists of three steps: 1) Travis CI information collection, 2) GitHub information collection, and 3) information integration.

We initially considered using existing datasets like TravisTorrent [3] and RTPTorrent [31], which are the most related datasets we are aware of. Because our dataset targets continuous test suite failure prediction we perform a more stringent filtering of builds to include. For example, we remove duplicated builds triggered by the same commits (around 50% builds removed) and only count first failures (around 40% builds removed), whereas neither TravisTorrent nor RTPTorrent perform such filtering. After our filtering process only around 2k data samples remain from the TravisTorrent dataset, which, given the imbalanced nature of the data, is insufficient for training an effective model. Instead, our dataset contains 15k samples, i.e., it is an order of magnitude larger than the best available dataset.

1) Travis CI information collection. We target Travis CI as our data source because it is among the most popular continuous integration services [17]. In Travis CI, a project includes many *builds*, and each build is triggered to run a series of tasks to determine whether the build passes or fails. A build consists of several *jobs*, which are usually executed in parallel, targeting the same software under different configurations, e.g., programming language versions and third-party package versions. If any job fails, the overall build will fail. A job consists of several *phases*, which typically include installation, building, testing, and deploying phases.

We consider only Travis CI projects with more than 100 builds to ensure that developers and organizations are experienced in continuous integration. We then check the build status and filter out builds that are skipped or cancelled. Next, we check each job status from job logs and exclude jobs that do not run tests. For each job that runs tests, if there are failed test cases, the test result for the job will be a failure. Otherwise, the result will be a pass. We also collect some information on testing, including test time and the number of test cases. After that, we label each build as "Pass" if all jobs of the build have passing test results, and as "Fail" otherwise. Finally, we remove data samples where several builds targeting the same commit are executed with different results to avoid flaky builds/tests [29].

2) GitHub information collection. We target GitHub as our data source for code changes because it is well-integrated with Travis CI. We focus on projects developed in Java, but our method could also be applied to other programming languages. Then, we identify the code changes on all branches that trigger the Travis CI builds, as well as the commits that constitute the code changes. By inspecting the commit logs and diff logs, we further exclude code changes that constitute a merge commit. A merge commit could be huge, and the intention could be scattered, which makes further analysis difficult. Finally, we retrieve the source files that are touched by the code change, excluding code changes that do not affect any code file.

3) Information integration. One challenge when integrating the data from Travis CI and GitHub is that some commits recorded to trigger a build in Travis CI can no longer be found in local Git logs. There are several possible reasons, e.g., the branch including the recorded trigger commits has been removed and can no longer be accessed, or the commits have been re-organized into the master branch and the commit ID has been updated. To resolve such missing trigger commits, we search for a valid commit that is committed on the same day, by the same author, and that affects the same files. If for the trigger commit in the Travis CI records we cannot find the corresponding GitHub commit, we ignore the build. Another challenge is that the same code change content, even in different branches, can be triggered in Travis CI by different developers. We identify such cases and remove all duplicates.

Table 3: Statistics of the dataset

Projects	242	Avg. changed lines	147
Period	07/2016–01/2020	Avg. test cases	12,041
Commits	23,720	Avg. build time	31 min
Samples	14,999	Avg. test time	21 min
Test suite failures	631 (4.21%)	Total failed test cases	15,874

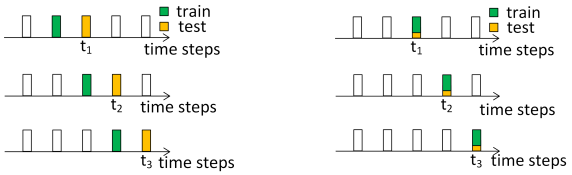
Statistics of the generated dataset are shown in Table 3. Our dataset includes 14,999 data samples, 631 of which are labeled as failure. That is, our dataset is imbalanced, with a test failure rate of 4.21%, which confirms earlier reports that most test cases pass [1]. The failure rate is much smaller than the 10.3% in Beller et al.'s study [2], because we only count first test failures (Section 2.4). Comparing individual projects with each other shows that the test suite failure rate varies. For example, Apache Druid has almost 24% failing test suite runs, whereas many other projects remain below 1%.[3] As our approach takes the test suite failure rate f as a parameter, it can be adapted to a specific project. Although there is an option to tolerate test failures and execute all test cases, it is only adopted by few developers (less than 1%). The "Failed test cases" include both failed tests, e.g., an assertion violation, and tests with errors, e.g., an uncaught exception. We count a test case multiple times if it is executed multiple times. From the statistics, we can also see that about two-thirds of the overall build time is spent on testing.

4.2 Training and Prediction

Data preprocessing. To address the imbalance of our dataset, we apply random oversampling, a class balancing technique that randomly duplicates samples from the minority class until the training set is balanced, on the training set. As is common with heterogeneous numeric features, we standardize all features so they have zero mean and unit variance.

Dataset split and model validation. We use three setups for splitting the dataset into training and validation data. On the one hand, we use 5-fold cross-validation with 10 repetitions of each fold, called the CV setup. We use stratified sampling to compute each fold. The results reported in the CV setup are averages over the $5 \cdot 10 = 50$ experiments. On the other hand, we also evaluate the approach with two time-based data splits, called the TIME1 and TIME2 setups, illustrated in Figure 3. For both time-based setups, we order all data

[3] All per-project statistics and results are available in the supplementary material.



(a) Setup TIME1: Use data from adjacent time steps for training and testing.

(b) Setup TIME2: Use the first 80% of the data of a time step for training and the remaining 20% for testing.

Figure 3: Time-based data splitting. The green boxes represent training data samples, and the orange boxes represent test data samples.

points by time and group them into 15 time steps that each have the same amount of failure samples. In the TIME1 setup, we use the data at time step i to train the model, and then validate with the data at time step $i+1$. In the TIME2 setup, we use the first 80% of the data at a time step for training, and then validate with the remaining 20% of the data at the same time step. The advantage of the time-based setups is to be closer to how our approach would be used in practice, where a model trained at some point is used for a period of time. However, the relatively small training and validation datasets of TIME1 and TIME2 cause some fluctuation in the results. We hence also use CV, which allows us to draw statistically significant conclusions. We use all three setups to evaluate the effectiveness of our approach (Section 5.1), and the CV setup to evaluate the features (Sections 5.2) and to apply our cost model (Section 5.3).

Classification models. We use several off-the-shelf classification models: decision trees, naive Bayes, support vector machine, logistic regression, random forest, and multi layer perceptron, all implemented in scikit-learn [41], and LightGBM, using the implementation of the authors [21].

4.3 Evaluation Metrics

To measure the prediction results, we use four metrics that have been used and shown to be important in related work [20, 53]: AUC, F-measure, G-measure, and MCC. AUC is the area under the receiver operating curve (ROC). It ranges from 0 to 1, where 0.5 means a random prediction, and 1 means a perfect prediction. We include this metric because it is robust against data imbalance and different thresholds. F-measure is the harmonic mean of precision and recall. It ranges from 0 to 1, and a higher F-measure means more accurate prediction. Precision is defined as $precision = \frac{TP}{TP+TP}$, while recall (True Positive Rate, TPR) is defined as $recall = TPR = \frac{TP}{TP+FN}$. F-measure is calculated as: $F\text{-}measure = \frac{2 \cdot precision \cdot recall}{precision+recall}$. G-measure is the harmonic mean of TPR and true negative rate $TNR = \frac{TN}{TN+FN}$. It aims to measure the influence of mislabeled positive and negative samples. G-measure ranges from 0 to 1, where higher is better. G-measure is calculated as: $G\text{-}measure = \frac{2 \cdot TPR \cdot TNR}{TPR+TNR}$. MCC, or Matthews Correlation Coefficient, describes the correlation between true values and predicted values. It ranges from -1 to 1, where 1 means perfect prediction, 0 means random prediction, and -1 means that all predictions failed. MCC is calculated as: $MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$. Both G-measure and MCC are robust against data imbalance and sensitive at different thresholds.

Table 4: Effectiveness of different classification models. The best performance of the models is highlighted in bold.

	AUC	F-measure	G-measure	MCC
Decision tree	0.576	0.203	0.333	0.169
Naive Bayes	0.696	0.253	0.349	0.231
Support Vector Machine	0.768	0.267	0.427	0.234
Logistic Regression	0.769	0.271	0.442	0.239
Random Forest	0.813	0.346	0.517	0.318
Multi-layer Perceptron	0.694	0.209	0.399	0.171
LightGBM	**0.836**	**0.386**	**0.561**	**0.359**

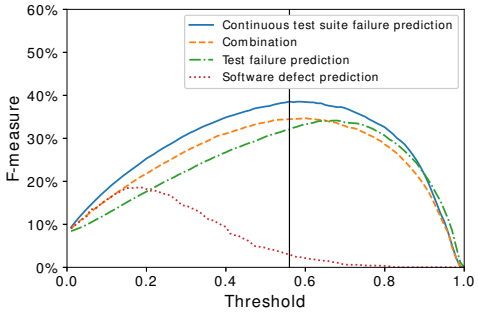


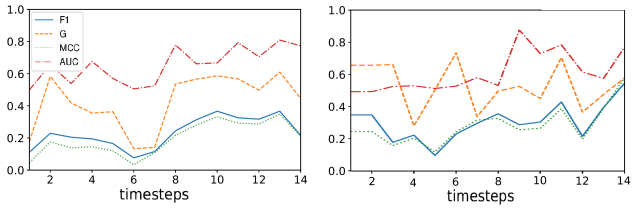
Figure 4: The effect of different classification thresholds and different features sets on prediction performance for the LightGBM model. The vertical line represents the default threshold.

5 RESULTS AND DISCUSSION

5.1 RQ1: How Effective are the Prediction Models?

5.1.1 Cross-Validation. Table 4 shows the effectiveness of different classification models measured with the CV setup (Section 4.2). LightGBM performs best according to all metrics, and we hence use this model as the default in the remainder of the evaluation. The results show that continuous test suite failure prediction is effective. From the view of evaluation metrics that are robust against data imbalance, the model performs well, with an AUC of 0.836. The F-measure of 0.386 is relatively low compared to related work on defect prediction (e.g., 0.43 in [20]) and test case failure prediction (0.522 in [1]). The main reason is that our data is even more imbalanced than theirs (12.8% and 10%, respectively), and that the F-measure is highly impacted by data imbalance. However, our model clearly outperforms a trivial classifier predicting all samples as failure, which would get an F-measure of 0.081.

We also explore a suitable classification threshold for the model (Figure 4). If the predicted probability is less than the threshold, the prediction result is "Pass"; otherwise, the result is "Fail". From the figure, we can see that the F-measure of the LightGBM model



(a) Setup TIME1. (b) Setup TIME2.

Figure 5: Prediction effectiveness using time-ordered data points.

is maximized at a threshold of 0.56, which shows that adjusting thresholds can achieve better prediction results.

> The test suite failure prediction model is effective. The best studied model, LightGBM, achieves an AUC of 0.836.

5.1.2 Time-Ordered Dataset. Figure 5 shows the effectiveness of our model with the two time-based splits into training data and validation data (TIME1 and TIME2, Section 4.2). We observe that the prediction performance is roughly on par with the results from cross-validation. For some time steps, the time-based splits slightly outperforms the earlier results, e.g., getting an F-measure of 0.546 for a specific time step. In general, the time-based splits yield slightly lower effectiveness, though, which can be attributed to the fact that the two time-based splitting methods result in less training data than cross-validation. To validate this hypothesis, we reduce the training dataset to random samples of 1,000 and 2,000 data points, which roughly matches the training set sizes for setups TIME1 and TIME2, respectively. Training with 1,000 data points achieves an AUC of 0.712 and an F1 of 0.215, and a dataset of 2,000 data points achieves an AUC of 0.721 and an F1 of 0.308.

> With time-based data splitting, the model is still effective, but provides slightly worse predictions due to the smaller size of the training data set.

5.2 RQ2: How Effective are the Features?

5.2.1 RQ2a: Comparison with Features Used in Prior Work. Because this is the first work on skipping all tests for a give code change, it is not clear a priori which features are most suitable. Table 5 compares our full feature set with different subsets: (i) only those features used in software defect prediction, (ii) only those features used in test case failure prediction, and (iii) the combination of (i) and (ii).[4] We choose the best threshold setting for each feature set, i.e., 0.18 for just-in-time defect prediction, 0.68 for test case failure prediction, 0.6 for their combination, and 0.56 for continuous test suite failure prediction. The results show that the full feature set is most effective. In particular, we find that our approach improves the AUC compared to each of the two existing baselines by 13.9% and 2.9%, respectively. To check if this improvement is statistically significant, we perform a Friedman test with the Holm correction method on the different train-validate splits during cross-validation.

[4]See Table 2 for what features belong to what prior work.

Table 5: Features known from prior work vs. full feature set from Table 2.

	AUC	F-measure	G-measure	MCC
Just-in-time defect prediction	0.697	0.186	0.340	0.147
Test case failure prediction	0.807	0.342	0.506	0.313
Combination	0.817	0.347	0.514	0.318
Continuous test suite failure prediction	**0.836**	**0.386**	**0.561**	**0.359**

With a confidence interval of 0.05, our feature set is significantly better. A direct comparison of the results also shows that our features outperform others in most cases for F-measure (47/50) and AUC (43/50).

Although the experiments are not designed for a direct comparison with build outcome prediction, we can roughly compare our results with Jin et al.'s results [19], and find that our F-measure is almost 30% higher. We attribute this difference to the fact that predicting the outcome of a test suite after a code change is an easier problem than predicting the outcome of a build, leading to a more effective model.

> Simply reusing features from related domains yields a less effective model than our full feature set.

5.2.2 RQ2b: Most Effective Features. To better understand how important individual features are for the model, we investigate the most and the least influential features. To this end, we rank the features according to their importance for training the LightGBM model, which yields the following:

- Ten most important features: TF10, TC, TP10, TF, REXP, SEXP, EXP, Awareness, CL, TFR20
- Ten least important features: ND, NClass, TU, NUC, NExc, NFIX, FIX, NS, Refactor, GC

The top-10 features focus on two categories: test and experience. Experience features are pervasively used in just-in-time defect prediction, and they also work well in continuous test suite failure prediction. Test features represent the historical test results of the builds, which are effective in test case failure prediction.

The least important features are mainly from the diffusion and the purpose categories. Refactor and GC only target a very small fraction of samples, and hence, their overall importance is low.

> Information about the developer experience, previous test results, and abundance of test cases is most important for an effective prediction model.

5.3 RQ3: (When) Is the Model Cost-Saving?

We instantiate the theoretical cost model (Section 3) with the failure rate and the confusion matrix values in our real-world dataset (averaged over cross-validation), i.e., with $f = 4.21\%$, $FP = 82.6$, $TN = 2791.0$, $TP = 49.9$, and $FN = 76.3$. Doing so enables us to assess if and when test suite failure prediction is overall cost-saving.

5.3.1 RQ3a: Comparison with Other Strategies. Instantiating the cost model yields a boundary condition for each of the five modeled strategies (Section 3.3). The boundary condition refers to $\frac{d}{r}$, i.e., the

Table 6: Boundary conditions for when the MODEL strategy outperforms the other strategies. ✓ means MODEL always outperforms the other strategies.

	RANDOM	ALL	NEVER	PERIOD			
				K=5	K=10	K=20	K=40
Boundary condition	$\frac{d}{r}$>0.14	$\frac{d}{r}$<37.58	$\frac{d}{r}$>2.47	✓	✓	✓	$\frac{d}{r}$>1.31

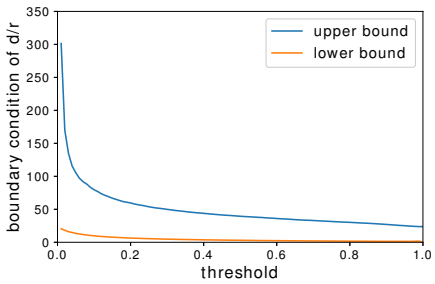


Figure 6: The upper bound and lower bound of boundary condition $\frac{d}{r}$ for MODEL strategy under different thresholds.

ratio of the cost of running a test suite and the cost of not immediately seeing a test failure. Table 6 shows the boundary conditions of each strategy compared to our MODEL strategy. For example, comparing our model with never running the test suite shows that $cost_{MODEL} < cost_{NEVER}$ when $\frac{d}{r}$>2.47. For PERIOD with k=5, 10, and 20, the boundary condition is ✓ to show that these strategies are always worse than MODEL because the boundary condition is always true.

By computing the optimality intervals of all strategies, we find three strategies to provide the overall lowest cost, depending on the cost parameters r and d:

- NEVER is optimal for: $\frac{d}{r} \in (0, 2.47)$
- MODEL is optimal for: $\frac{d}{r} \in (2.47, 37.58)$
- ALL is optimal for: $\frac{d}{r} \in (37.58, +\infty)$

These intervals show that our predictive model can be cost-saving for a large range of scenarios. For example, suppose an organization where the cost r of running a test suite as part of continuous integration is 2 person-hours and the cost of finding out late about a test suite failure is 20 person-hours, which are values compatible with existing estimates and reports [5, 34, 51]. In this organization $\frac{d}{r} = 10$, i.e., MODEL is the best available strategy.

Instantiating the theoretical cost model with real-world data shows that the predictive model is cost-effective if $2.47 < \frac{d}{r} < 37.58$. The MODEL, NEVER, and ALL strategies outperform the other strategies under different optimality intervals.

5.3.2 RQ3b: Influence of Classification Threshold on Cost Model. By default, the classification threshold is 0.56, which we pick because it maximizes F-measure (see RQ1). However, the threshold may not be the best threshold for the cost model. We enumerate thresholds with a step size of 0.01, calculate the boundary conditions of MODEL versus other strategies for each threshold, and then determine the lower and upper bounds. The ratio of executed test suites is 50.8%, 4.4%, and 0.34% at a threshold of 0.05, 0.56, and 0.95, respectively. Figure 6 shows how the optimality interval of the MODEL strategy changes depending on the threshold.

The figure allows for two observations. First, there is a relatively large range of $\frac{d}{r}$ ratios where MODEL is the best strategy, irrespective of the threshold. This differs from Herbold's observations in software defect prediction [16]. The reason is that there is always a tradeoff between the two kinds of costs that our model considers. Second, one can adapt the classification threshold to make MODEL beneficial for a given $\frac{d}{r}$ ratio. Up to a ratio of 301, our model outperforms all other strategies when choosing an appropriate threshold. For $\frac{d}{r} > 301$, i.e., a scenario where missing a test failure is extremely costly, the best threshold is 0, which is equivalent to always executing the test suite, i.e., the ALL strategy.

The optimality interval of our model is robust w.r.t. the classification threshold. By adjusting the threshold, our model can outperform the other strategies in scenarios up to a $\frac{d}{r} = 301$.

5.3.3 RQ3c: Other Costs. Beyond the costs represented in the cost model, deploying our idea in a realistic setting imposes some additional computational costs. One is for gathering the features of code changes during the evolution of a project. This effort can be greatly reduced by caching and updating project information locally for each code change. During feature generation, the most time-consuming part is the AST-based analysis, which we find to take less than a second per code change. The computational costs of training a model, which would likely be done at regular intervals in practice, is up to several minutes, and the cost of a prediction is within seconds. All computation times are measured on a standard laptop. Overall, these costs are negligible compared to the cost of regularly running large test suites.

6 THREATS TO VALIDITY

We see four threats to the validity of our results. First, flaky tests may introduce noise into our dataset. We mitigate this threat by removing test suite runs labeled as flaky in the Travis CI test reports, which affects a small percentage of all test suite runs (0.067%). We further check for code changes that trigger multiple test suite runs (8.6% of all code changes) and remove those that yield different test results (<10% of the 8.6%). Despite this filtering, there may still be flaky tests in our dataset, e.g., because the test suite is triggered only once and thus not identified as flaky. Second, our results may not generalize to different or larger datasets, or to programs written in languages other than Java. To partially address this threat, we gather our own dataset, which is an order of magnitude larger than the largest available dataset [3]. Third, even though our work is partially motivated by the immense testing efforts performed at large-scale organizations, our evaluation focuses on open-source projects, which have a comparably small scale. We hence do not claim our empirical results to hold for large-scale organizations with a centralized continuous integration infrastructure, but leave applying our idea in such environments for future work. Finally,

our cost model is a theoretical abstraction of real costs, which (as every model) may not accurately represent reality. Instead, we see the cost model as a way to reason about the advantages of different continuous testing strategies.

7 RELATED WORK

7.1 Reducing the Cost of Regression Testing

One popular way to reduce the time required for executing regression tests is test selection [10, 12, 24, 39, 48]. A related approach is test case prioritization, which aims at executed those tests first that are the most likely to reveal faults [7, 15, 33, 40]. Both kinds of approaches have been studied on industrial systems [38] and on evolving code repositories [26, 28]. Leong et al. [25] propose a simulation framework for studying test selection and prioritization, and apply it at Google. Their results underline the importance of flaky tests, which our approach ignores and which we try to filter in our dataset.

Test case failure prediction also aims to identify test cases that are more likely to fail. Anderson et al. [1] predict each test case as passing or failing before these tests are executed. Machalica et al. [30] use change-level, target-level, and other features to predict test case results, and they also consider the effects of test flakiness. We adopt some of the features from test case failure prediction, but also show that our extended feature set is even more effective. Our newly added code change features are likely to improve stability, as reported also by Lu et al. [28]. Our work shares with all the above work the goal of reducing the overall testing effort, and future work should explore how to combine our work with prior techniques. E.g., our work could decide whether to run the test suite at all, and if this decision is positive, existing work could optimize which test cases to run.

Similar to this work, some prior work also aims at optimizing which tests to execute at the level of test suites. Google's test automation platform [34] schedules test targets to run, where "target" means a set of related test cases or test scripts. Their approach decides which test targets to eventually execute based on dependencies, but heuristically postpones test executions to reduce the overall test execution effort. Test suite selection [8, 49] tries to identify the most failure-prone subset out of possibly many test suites associated with a code base. They use three features: the last time a test suite has triggered a failure, the last time a test suite was executed, and whether the test suite is newly created. While these features partially overlap with ours, our analysis of most effective features (Section 5.2.2) shows that there are also other important features. Another approach is to combine test selection at several levels of granularity, e.g., by first selecting tests at the module level and by then fine-tuning the selection at the class level [49].

Saff and Ernst [46] propose a form of continuous testing aimed at reducing the time between introducing a mistake and finding it via testing, by giving the illusion to developers that regression tests are continuously running in the background. The approach uses otherwise idle computational resources on the developer's machine, whereas we target a continuous integration system, where tests are running on some centralized computing infrastructure whenever a developer commits code.

7.2 Just-in-Time Defect Prediction

Just-in-time defect prediction aims to predict if a code change introduces defects into the code base. Unlike test failures, software defects are usually not identified in the first place. Kamei et al. [20] is among the pioneer researchers in just-in-time defect prediction. They proposed change measures in diffusion, size, purpose, history, and experience categories. Many researchers follow their work [18, 27, 32, 53] and make further improvements. Tabassum et al. [53] studies the impact of training set size on prediction performance. McIntosh et al. [32] consider the "moving target" effect in a longitude just-in-time defect prediction study. We adopt features from just-in-time defect prediction, which are then used in continuous test suite failure prediction. Our results show that simply reusing these features is less effective than our extended feature set, and we also investigate the influence of a time-ordered dataset [53].

7.3 Build Outcome Prediction

Continuous build outcome prediction [14] aims to predict whether a build will pass. Zheng et al. [55] proposes a semi-supervised online prediction method, and Rausch et al. [44] find that the recent build history is the strongest influencing factor. Jin and Servant [19] and Chen et al. [4] propose build outcome prediction models that distinguish between "first failures" and subsequent failures. Our dataset also focuses on "first failures", i.e., test suite failures that are preceded by a passing test suite. The key difference to prior work is that our approach focuses on test suite failures, instead of all kinds of build failures, which include various other reasons, e.g., compilation errors, configuration problems, missing dependencies, and outages of external services. Compared to build outcome prediction, the test suite failure prediction problems is less complex and easier to characterize by features, making it a fruitful target for accurate prediction.

7.4 Cost Models

Herbold [16] proposes a cost model for software defect prediction, which considers, e.g., quality assurance costs, defect costs, initialization costs, and execution costs. Our model is similar in spirit to [16], but is based on a set of assumptions and cost parameters suitable for continuous test suite failure prediction instead of defect prediction. Another line of work is on cost estimations for regression test selection [13, 45], which checks whether the cost necessary to select subsets of the tests is lower than the savings obtained by running the reduced test suite. There are other domains where the idea of cost models is adopted, e.g., modeling the costs of releases [42] and fraud detection [52]. This paper presents the first cost model for continuous test suite failure prediction.

8 CONCLUSION

This paper introduces the problem of continuous test suite failure prediction, presents an effective prediction model that addresses the problem, and evaluates the prediction model with a novel dataset gathered from 204 real-world projects. The prediction model is based on nine categories of features, which focus on the code change, the past behavior of the test suite, and the development history. Our experiments show these features to yield an effective classification model. Moreover, we find that our approach improves

over baselines that use features proposed for just-in-time defect prediction and for predicting whether to run individual test cases by 13.9% and 2.9%, respectively. We also present a theoretical cost model, which reasons about whether using our approach is beneficial depending on the cost of running a test suite and the cost of not immediately detecting a test failure. Beyond the evaluation metrics, the cost model helps decide when continuous test suite failure prediction is worthwhile based on estimates of both costs for a specific project or organization. We envision continuous test suite failure prediction to be useful both for large-scale organizations with a centralized continuous integration infrastructure and for continuous integration platforms that offer their services to open-source and other projects.

ACKNOWLEDGMENTS

This work has been supported by the China Sponsorship Council, by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and Perf4JS projects.

REFERENCES

[1] Jeff Anderson, Saeed Salem, and Hyunsook Do. 2015. Striving for failure: an industrial case study about test failure prediction. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 49–58. https://doi.org/10.1109/ICSE.2015.134

[2] M. Beller, G. Gousios, and A. Zaidman. 2017. Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 356–367. https://doi.org/10.1109/MSR.2017.62

[3] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Travistorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 447–450. https://doi.org/10.1109/MSR.2017.24

[4] 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 *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 42–53.

[5] Hyunsook Do and Gregg Rothermel. 2006. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. 141–151. https://doi.org/10.1145/1181775.1181793

[6] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. 2002. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering* 28, 2 (2002), 159–182. https://doi.org/10.1109/32.988497

[7] Sebastian G. Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. 2004. Selecting a Cost-Effective Test Case Prioritization Technique. *Softw. Qual. J.* 12, 3 (2004), 185–210. https://doi.org/10.1023/B:SQJO.0000034708.84524.22

[8] Sebastian G. 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, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 235–245. https://doi.org/10.1145/2635868.2635910

[9] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982

[10] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, Michal Young and Tao Xie (Eds.). ACM, 211–222. https://doi.org/10.1145/2771783.2771784

[11] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. 2010. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 495–504. https://doi.org/10.1145/1806799.1806871

[12] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression Test Selection for Java Software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001*, Linda M. Northrop and John M. Vlissides (Eds.). ACM, 312–326. https://doi.org/10.1145/504282.504305

[13] Mary Jean Harrold, David S. Rosenblum, Gregg Rothermel, and Elaine J. Weyuker. 2001. Empirical Studies of a Prediction Model for Regression Test Selection. *IEEE Trans. Software Eng.* 27, 3 (2001), 248–263. https://doi.org/10.1109/32.910860

[14] 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 (ASE '06)*. IEEE Computer Society, USA, 189–198. https://doi.org/10.1109/ASE.2006.72

[15] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing white-box and black-box test prioritization. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 523–534. https://doi.org/10.1145/2884781.2884791

[16] Steffen Herbold. 2019. On the costs and profit of software defect prediction. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2957794

[17] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 426–437. https://doi.org/10.1145/2970276.2970358

[18] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 34–45. https://doi.org/10.1109/MSR.2019.00016

[19] Xianhao Jin and Francisco Servant. 2020. A Cost-efficient Approach to Building in Continuous Integration. In *International Conference on Software Engineering*. 13–25.

[20] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773. https://doi.org/10.1109/TSE.2012.70

[21] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in neural information processing systems*. 3146–3154. https://doi.org/10.1109/MSR.2019.00016

[22] Kenji Kira and Larry A. Rendell. 1992. A Practical Approach to Feature Selection. In *Machine Learning Proceedings 1992*, Derek Sleeman and Peter Edwards (Eds.). Morgan Kaufmann, San Francisco (CA), 249–256. https://doi.org/10.1016/B978-1-55860-247-2.50037-1

[23] A Güneş Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu. 2008. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering* 35, 2 (2008), 293–304. https://doi.org/10.1109/TSE.2008.90

[24] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 583–594. https://doi.org/10.1145/2950290.2950361

[25] Claire Leong, Abhayendra Singh, Mike Papadakis, Yves Le Traon, and John Micco. 2019. Assessing transition-based test selection algorithms at Google. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, Helen Sharp and Mike Whalen (Eds.). IEEE / ACM, 101–110. https://doi.org/10.1109/ICSE-SEIP.2019.00019

[26] 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. https://doi.org/10.1145/3180155.3180213

[27] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. 2017. Code churn: A neglected metric in effort-aware just-in-time defect prediction. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 11–19. https://doi.org/10.1109/ESEM.2017.8

[28] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How does regression test prioritization perform in real-world software evolution?. In *Proceedings of the 38th International Conference on Software Engineering*. 535–546. https://doi.org/10.1145/2884781.2884874

[29] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 643–653. https://doi.org/10.1145/2635868.2635920

[30] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. In *Proceedings of the 41st International Conference on*

Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019, Helen Sharp and Mike Whalen (Eds.). IEEE / ACM, 91–100. https://doi.org/10.1109/ICSE-SEIP.2019.00018

[31] Toni Mattis, Patrick Rein, Falco Dürsch, and Robert Hirschfeld. 2020. RTPTorrent: An open-source dataset for evaluating regression test prioritization. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 385–396. https://doi.org/10.1145/3379597.3387458

[32] Shane McIntosh and Yasutaka Kamei. 2018. Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction. *IEEE Trans. Software Eng.* 44, 5 (2018), 412–428. https://doi.org/10.1109/TSE.2017.2693980

[33] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. 2012. A Static Approach to Prioritizing JUnit Test Cases. *IEEE Trans. Software Eng.* 38, 6 (2012), 1258–1275. https://doi.org/10.1109/TSE.2011.106

[34] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 233–242. https://doi.org/10.1109/ICSE-SEIP.2017.16

[35] Audris Mockus and David M Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5, 2 (2000), 169–180. https://doi.org/10.1002/bltj.2229

[36] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*. 284–292. https://doi.org/10.1109/ICSE.2005.1553571

[37] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*. 452–461. https://doi.org/10.1145/1134285.1134349

[38] Daniel Di Nardo, Nadia Alshahwan, Lionel C. Briand, and Yvan Labiche. 2015. Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system. *Softw. Test. Verification Reliab.* 25, 4 (2015), 371–396. https://doi.org/10.1002/stvr.1572

[39] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004*, Richard N. Taylor and Matthew B. Dwyer (Eds.). ACM, 241–251. https://doi.org/10.1145/1029894.1029928

[40] David Paterson, José Campos, Rui Abreu, Gregory M Kapfhammer, Gordon Fraser, and Phil McMinn. 2019. An Empirical Study on the Use of Defect Prediction for Test Case Prioritization. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 346–357. https://doi.org/10.1109/ICST.2019.00041

[41] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[42] Hoang Pham. 2003. Software reliability and cost models: Perspectives, comparison, and practice. *European Journal of Operational Research* 149, 3 (2003), 475–489. https://doi.org/10.1016/S0377-2217(02)00498-8

[43] Ranjith Purushothaman and Dewayne E Perry. 2005. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering* 31, 6 (2005), 511–526. https://doi.org/10.1109/TSE.2005.74

[44] T. Rausch, W. Hummer, P. Leitner, and S. Schulte. 2017. An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 345–355. https://doi.org/10.1109/MSR.2017.54

[45] David S. Rosenblum and Elaine J. Weyuker. 1996. Predicting the Cost-Effectiveness of Regression Testing Strategies. In *SIGSOFT '96, Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering, San Francisco, California, USA, October 16-18, 1996*, David Garlan (Ed.). ACM, 118–126. https://doi.org/10.1145/239098.239118

[46] David Saff and Michael D. Ernst. 2003. Reducing wasted development time via continuous testing. In *14th International Symposium on Software Reliability Engineering (ISSRE 2003), 17-20 November 2003, Denver, CO, USA*. IEEE Computer Society, 281–292. https://doi.org/10.1109/ISSRE.2003.1251050

[47] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. 2016. Continuous deployment at Facebook and OANDA. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 21–30. https://doi.org/10.1145/2889160.2889223

[48] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing trade-offs in test-suite reduction. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 246–256. https://doi.org/10.1145/2635868.2635921

[49] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and improving regression test selection in continuous integration. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 228–238. https://doi.org/10.1109/ISSRE.2019.00031

[50] Emad Shihab, Ahmed E Hassan, Bram Adams, and Zhen Ming Jiang. 2012. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11. https://doi.org/10.1145/2393596.2393670

[51] Forrest Shull, Vic Basili, Barry Boehm, A Winsor Brown, Patricia Costa, Mikael Lindvall, Daniel Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. 2002. What we have learned about fighting defects. In *Proceedings eighth IEEE symposium on software metrics*. IEEE, 249–258. https://doi.org/10.1109/METRIC.2002.1011343

[52] Salvatore J Stolfo, Wei Fan, Wenke Lee, Andreas Prodromidis, and Philip K Chan. 2000. Cost-based modeling for fraud and intrusion detection: Results from the JAM project. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, Vol. 2. IEEE, 130–144. https://doi.org/10.1109/DISCEX.2000.821515

[53] Sadia Tabassum, Leandro L Minku, Danyi Feng, George G Cabral, and Liyan Song. 2020. An investigation of cross-project learning in online just-in-time software defect prediction. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 554–565. https://doi.org/10.1145/3377811.3380403

[54] 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 2015 10th Joint Meeting on Foundations of Software Engineering*. 805–816. https://doi.org/10.1145/2786805.2786850

[55] Zheng Xie and Ming Li. 2018. Cutting the Software Building Efforts in Continuous Integration by Semi-Supervised Online AUC Optimization. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 2875–2881. https://doi.org/10.24963/ijcai.2018/399

[56] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stvr.430