Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration

Daniel Elsner
Florian Hauer
Alexander Pretschner
daniel.elsner@tum.de
florian.hauer@tum.de
alexander.pretschner@tum.de
Technical University of Munich
Munich, Germany

Silke Reimer
sre@ivu.de
IVU Traffic Technologies
Berlin, Germany

ABSTRACT

Regression test selection (RTS) and prioritization (RTP) techniques aim to reduce testing efforts and developer feedback time after a change to the code base. Using various information sources, including test traces, build dependencies, version control data, and test histories, they have been shown to be effective. However, not all of these sources are guaranteed to be available and accessible for arbitrary continuous integration (CI) environments. In contrast, metadata from version control systems (VCSs) and CI systems are readily available and inexpensive. Yet, corresponding RTP and RTS techniques are scattered across research and often only evaluated on synthetic faults or in a specific industrial context. It is cumbersome for practitioners to identify insights that apply to their context, let alone to calibrate associated parameters for maximum cost-effectiveness. This paper consolidates existing work on RTP and unsafe RTS into an actionable methodology to build and evaluate such approaches that exclusively rely on CI and VCS metadata. To investigate how these approaches from prior research compare in heterogeneous settings, we apply the methodology in a large-scale empirical study on a set of 23 projects covering 37,000 CI logs and 76,000 VCS commits. We find that these approaches significantly outperform established RTP baselines and, while still triggering 90% of the failures, we show that practitioners can expect to save on average 84% of test execution time for unsafe RTS. We also find that it can be beneficial to limit training data, features from test history work better than change-based features, and, somewhat surprisingly, simple and well-known heuristics often outperform complex machine-learned models.

CCS CONCEPTS

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

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

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464834

KEYWORDS

software testing, regression test optimization, machine learning

ACM Reference Format:
Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration. 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, 14 pages. https://doi.org/10.1145/3460319.3464834

1 INTRODUCTION

Regression test selection (RTS) aims at identifying tests that are affected by a change to the code base, when executing every test in the test suite is prohibitively expensive [25, 76]. Effective traditional RTS techniques *safely* exclude those tests that cannot fail by relying on language-specific white-box program analyses, e.g., recording test-specific execution traces through code instrumentation [26, 42, 58, 65, 66, 70, 79]. However, they are often too costly in large-scale code bases with rapid continuous integration (CI) testing [21, 45, 62], not capable of collecting test dependencies across language boundaries in multi-language software [12, 45, 55], and cannot trace third-party libraries [41]. Regression test prioritization (RTP) aims to detect faults earlier by reordering tests through "surrogates" [76]. However, traditional RTP techniques that rely on code coverage surrogates suffer similar limitations [21, 71].

To address these limitations specifically in CI environments, researchers have proposed numerous lightweight, less intrusive RTP and *unsafe* RTS techniques: They use different surrogates and machine learning (ML) models to rank tests by their likelihood to fail and, in the case of RTS, select a subset based on some cut-off criterion [8, 11, 45, 60, 71]. The underlying *ranking models* exploit different information sources. These include CI test execution logs [3, 11, 21, 71], version control system (VCS) metadata (e.g., number of changed files in commit) [37, 62], (textual) differences in code churn [8, 49, 60, 67], and project- or organization-specific information such as static build dependencies [45], flaky test detection signals [45, 60, 62], or a black-box model of the program inputs [29]. Arguably, access to the latter types of information cannot be guaranteed for arbitrary CI environments. Since CI and VCS metadata are automatically generated throughout the development process, they are generally available and inexpensive. Unsafe RTS

and RTP techniques that solely use this information are language-agnostic, easy to transfer and to evaluate, and do not require program or code access. Methodologically, they both rely on ranking tests [8, 11]. Hence, in the following, we collectively refer to techniques as CI-RTP/S, if their ranking models *only* use CI and VCS metadata.

Even though the effectiveness of CI-RTP/S is demonstrated in various studies, they have the following limitations: First, while studies show the effectiveness on single projects [3, 16, 21, 38, 43, 68] and variations across projects [37, 44, 71, 81], we find *sensitivity* to be another important attribute after consulting with our industry partner IVU Traffic Technologies[1]: How sensitive is the cost-effectiveness of CI-RTP/S to size, timeliness, and variety of data they were calibrated on? Or, since calibration can be challenging, how much cost-effectiveness is sacrificed when using a ranking model that is only semi-optimally calibrated? Existing studies that discuss related issues were carried out in a specific industrial context and it is thus unclear, if the measured sensitivity carries over to other industrial and open-source projects. Second, empirical results of CI-RTP/S cost-effectiveness are often obtained on datasets with seeded faults instead of real-world failures [8, 13] or drawn from inaccessible industrial contexts which impedes reproducibility [11, 45, 62]. A recently published dataset, RTPTorrent [50, 51], closely resembles real-world development activity. Though it has not yet been used in related studies, this dataset bears the potential to improve transparency and thus transferability of results from CI-RTP/S studies. Last, in pursuit of ever more effective CI-RTP/S, aspects of the design and evaluation of techniques are scattered across existing work. Overall, findings in the literature are neither unequivocal, nor directly comparable. It is a cumbersome task for practitioners to identify insights that apply to their context, let alone to calibrate associated parameters for maximum cost-effectiveness.

Addressing these limitations, we consolidate existing ideas from prior research into a methodology to build and evaluate approaches for RTP and unsafe RTS that exclusively rely on CI and VCS metadata. We identify an *approach* by three parameters, namely (i) *how much* (i.e., amount of training data) of (ii) *which information* (i.e., choice of features) is used to (iii) *rank tests* (i.e., choice of ranking model). Our methodology does not propose new techniques, but provides a clear, generic process that first guides practitioners from exploiting their readily available CI and VCS metadata to building candidate CI-RTP/S approaches (from prior research) for their project. Then, the subsequent comparative evaluation of candidates yields not only the most cost-effective approaches, but also gives insights on how an approach's performance changes if any of the three parameters is varied, i.e., on its cost-effectiveness sensitivity. From a practitioners view, this methodology enables simple adoption of RTP and unsafe RTS without having to manually investigate existing techniques and their applicability. We thus address the practical questions of which data to gather, how to perform feature engineering and predictive modeling, how to comparatively evaluate different candidate approaches, and how to choose the project-specific best approach.

To estimate performance trade-offs, we apply our methodology in a large-scale empirical study on real test failures from CI and VCS histories of 23 industrial and open-source projects. We then conduct rigorous statistical analyses, yielding guidelines on which candidate approaches are, empirically, the most promising ones.

In summary, our **contributions** are as follows:

- **Methodology.** We consolidate existing research into an actionable methodology to build and evaluate approaches for RTP and unsafe RTS exclusively using CI and VCS metadata.
- **Empirical Study.** First, we analyze the sensitivity of cost-effectiveness to the parameters (i) training data amount, (ii) choice of features, and (iii) choice of ranking model. Second, we estimate performance trade-offs in RTP and unsafe RTS: Cost-effectiveness fluctuates across projects underlining the need for project-specific assessment using our methodology. In the studied projects, approaches chosen by our methodology help to save on average 84% of test execution time while detecting 90% of the failures for unsafe RTS and significantly outperform established RTP baselines.
- **Guidelines.** (1) It can be beneficial to limit training data, (2) features on test history work particularly well compared to change-based features, and (3) inexpensive simple heuristics of the kind "skip test if not failed in the last ten runs" often outperform complex ML models from prior work.
- **Dataset.** To foster comparable and evidence-based studies on CI-RTP/S, we publish our dataset consisting of 23 heterogeneous software projects from industry (3) and open-source development (20)[2]. It covers more than 37,000 CI test logs with real failures and 76,000 VCS commits. The open-source projects are drawn from the recently published RTPTorrent dataset [50, 51] which embodies CI test logs that we further enrich and process for CI-RTP/S.

2 METHODOLOGY

We have motivated our goal to build approaches useful for both, unsafe RTS and RTP, that solely rely on CI and VCS metadata (CI-RTP/S). Before providing details about the methodology, we describe the problem more formally and introduce notations used throughout this paper: Let P be a program, Δ be a modification introduced to P to create P', and $\mathcal{T}$ be a test suite. For each test $T \in \mathcal{T}$, the *ranking model*, M, first predicts T's failure score by using a *set of features*, F. Second, these scores are used to rank tests in $\mathcal{T}$, yielding an intermediary $\mathcal{T}^*$, i.e., a test order as aimed at by RTP. Then, based on a *cut-off criterion* only a subset $\mathcal{T}' \subseteq \mathcal{T}^*$ is selected as part of RTS. Depending on the desired strategy (RTP or RTS), $\mathcal{T}^*$ or $\mathcal{T}'$ can be used to test P' [11, 18, 45, 65].

We identify four consecutive process steps when creating and evaluating CI-RTP/S approaches which are shown in Fig. 1: Exploiting available data sources, engineering features from the collected data, building predictive ranking models, and evaluating CI-RTP/S. Since each of these steps involves methodological subgoals, we address them in the following subsections. Notably, this schematic process is inspired by the Cross-Industry Standard Process for Data Mining (CRISP-DM) [73].

[1] *IVU Traffic Technologies* is one of the world's leading providers of public transport software solutions: https://www.ivu.com/

[2] Dataset, source code, and detailed evaluation results are part of the supplemental material available at [22].

Figure 1: Schematic process of the methodology to build and evaluate CI-RTP/S approaches.

2.1 Data Source Exploitation

Goal: Exploit *generally available* data sources to collect raw data useful for failure prediction.

Modern software projects are developed in code repositories that use some sort of VCS. While there are various flavors of VCS, such as distributed (e.g., Git) or centralized (e.g., Apache Subversion), they share the notion of a *commit*, i.e., a code revision made by a single author. It will contain at least the following information: Identifier, author, commit timestamp, commit message, and a changeset. The changeset in turn includes all the added, modified, or deleted files.

Meanwhile, regression testing is typically performed in CI environments. CI tools, such as Jenkins or Travis CI, allow users to configure CI pipelines, which are regularly executed. We refer to a CI run at timestamp t as R_t. Most pipelines contain multiple stages, including a *build stage*, where artifacts necessary to run the tests are generated (e.g., compilation), a *test stage*, where (regression) tests are executed, and a *deploy stage*, which comprises of scripts to publish and deploy tested artifacts. After the test stage in R_t, a *test log* (also called test report) is typically generated from which the following information can be extracted per test $T_{t,i}$ in the executed test suite $\mathcal{T}_t = \{T_{t,1}, \ldots, T_{t,n}\}$: Test identifier (e.g., test class name[3]), result (e.g., passed or failed), and duration. It is thus a requirement to derive test result and duration information from the CI environment. However, most testing frameworks and CI systems (or plugins) already provide structured test logs in several output formats, but they can also be parsed from raw textual CI logs, e.g., by using regular expressions [50]. Depending on the configuration, a CI run R_t may be triggered to start either after each commit, after the previous run R_{t-1} has finished, or whenever required (hardware) resources are available. The set of introduced commits between two CI runs, R_t and R_{t-1}, Δ_t, will at least contain one commit, as it would be pointless to trigger a CI run without any modification except to detect flaky tests (see Sec. 3.6.1).

Fig. 2 shows how the outlined entities *commit* and *test log* are part of the software development process. The union of all available Rs with their $\mathcal{T}$ and Δ constitutes the input for the feature engineering process described in the next section.

2.2 Feature Engineering

Goal: Craft features for failure prediction from collected raw data that capture specific *defect* hypotheses.

For a given test T, ranking models use *features* to predict a failure score. These features are numerical representations of characteristics of T. For example, considering T's failure behavior, if T failed ten times in previous test runs, the feature *failure count* will have the value 10. A *good* feature is one that improves the model's predictive performance, hence, one that captures a valid *defect* hypothesis.

[3]We follow prior RTS research by analyzing tests at class (or file) rather than module or method granularity level [27, 70, 78].

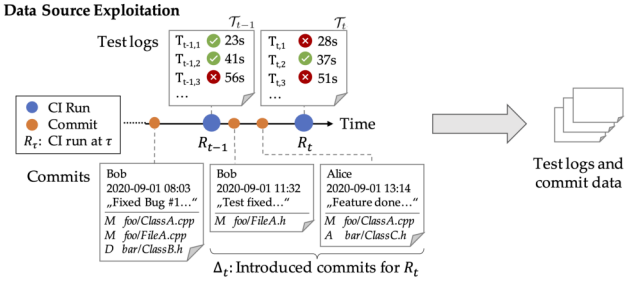


Figure 2: Exploiting VCS history and CI test logs used as input for feature engineering.

Note that we use *defect* as an umbrella term for *failures*, *faults*, or *errors*.

Feature engineering is concerned with deriving these defect hypotheses and computing respective features through a feature function ϕ. If applied to the collected raw data, i.e., to each test in all $\mathcal{T}$s with their respective Δ, we can construct a dataset, D, suitable for building and evaluating ranking models. More formally, for a given test $T_{t,i}$ in the test suite $\mathcal{T}_t$ available for R_t, we can calculate a vector of m features, $x_{t,i} = (x_{t,i,1}, \ldots, x_{t,i,m})$, from raw data using the feature function $\phi(T_{t,i}, \{\mathcal{T}_1, \ldots, \mathcal{T}_{t-1}\}, \{\Delta_1, \ldots, \Delta_t\})$. Note that this captures reality, where at timestamp t before regression testing, there are only historical test logs and commits available as well as the newly introduced set of commits Δ_t. To create D, this is done for every test suite $\mathcal{T}$ of the collected CI runs.

Hereafter, we describe 16 features for CI-RTP/S from existing work. For each feature, we state the underlying defect hypothesis, briefly explain how it is computed from outlined raw CI test logs and VCS commits, and reference prior research which already used it. Aligned with prior work, we semantically group them into four feature sets, $F = \{F_1, \ldots, F_4\}$, to increase comprehensiveness [4, 62]. The k-th feature in feature set j is denoted by $f_{j,k}$.

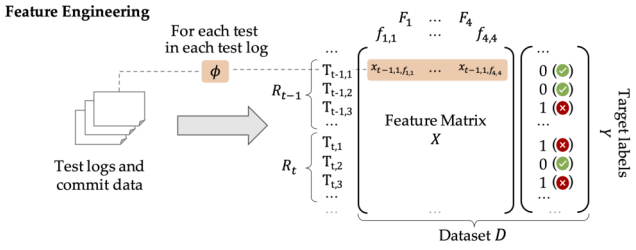


Figure 3: Structure of a dataset D for CI-RTP/S. Rows are chronologically ordered, i.e., from test executions of the earliest to the most recent CI run.

Fig. 3 illustrates how we derive D by computing F for all test executions in the test logs, e.g., test $T_{t,1}$ in CI run R_t, and labeling them with their respective test results as the ground truth, Y, e.g., 1 for *failed*. Based on the total amount of collected CI runs with test logs, $r = |R|$, D will have N rows, with $N = |\bigcup_{t=1}^{r} \mathcal{T}_t|$, and $M+1$ columns, with $M = |\bigcup_{j=1}^{4} F_j|$, including one column for $Y \in \{0,1\}^{N\times 1}$. The

feature matrix, denoted by $X \in \mathbb{R}^{N \times M}$, stacks all computed feature values for each test execution $T_{t,i}$, $x_{t,i} = (x_{t,i,1}, \ldots, x_{t,i,M})$.

Test History Features (F_1): This feature set contains features computed only from CI test logs, i.e., for a test T in $\mathcal{T}_t$ we consider its individual execution history from $\{R_1, \ldots, R_{t-1}\}$.

The first hypothesis is: Tests that previously failed will test error-prone code and are therefore likely to fail again [28, 74, 76].

- Failure count ($f_{1,1}$) [4, 45, 54, 56, 57, 60]: Total number of times T failed.
- Last failure ($f_{1,2}$) [4, 21, 34, 47]: Amount of CI runs since last failure, i.e., for R_t, if T last failed in R_τ, the value of $f_{1,2}$ will be $t - \tau$.

A *test transition* occurs if a test changes its test result between two CI runs. For instance, if a test failed in R_{t-1} and passed in the subsequent run R_t (or vice versa), the test *transitioned* in R_t.

The second hypothesis is: Tests that previously transitioned will test critical code and are therefore likely to fail.

- Transition count ($f_{1,3}$) [43]: Total number of times T transitioned.
- Last transition ($f_{1,4}$) [43]: Amount of CI runs since last transition, analogous to the feature $f_{1,2}$.

The third hypothesis is: Tests with high execution duration involve complex, time-consuming tasks (e.g., networking, file access) which are prone to fail (e.g., timeout) [62]. Additionally, on the one hand, longer running tests might cover larger parts of the program which simultaneously increases the chance of covering a fault. On the other hand, fast-running tests might reveal faults more quickly, making test execution time a potentially useful feature [13, 60].

- Average test duration ($f_{1,5}$) [13, 60]: Average test execution duration of T across previous CI runs.

(Test, File)-History Features (F_2): The idea of this feature set is to identify associations between tests and files, often referred to as *test-to-code* traceability links [72]. Therefore, we record historical co-occurrences of test failures or transitions with changed files in a contingency table [37]. For example re-consider Fig. 2, if test $T_{t,3}$ failed in R_t where the *combined changeset* {*foo/FileA.h, foo/ClassA.cpp, bar/ClassC.h*} was introduced, we increment the co-occurrence frequencies of ($T_{t,3}$, *foo/FileA.h*), ($T_{t,3}$, *foo/ClassA.cpp*), and ($T_{t,3}$, *bar/ClassC.h*). These co-occurrences are referred to as *(test, file)-failures*, the same applies to *(test, file)-transitions*.

The first hypothesis is: Files that were in the changeset when a test *failed*, are related to this test's outcome. More precisely, if certain files are in the changeset of a commit, they might imply higher failure likelihood for specific tests.

- Maximum (test, file)-failure frequency ($f_{2,1}$) [3, 37, 54, 62, 68]: Given a test T in R_t, for each file in the combined changeset, we first obtain the total (T, file)-failure frequency from $\{\mathcal{T}_1, \ldots, \mathcal{T}_{t-1}\}$. Then, we determine the maximum across all files, as the combined changeset is as risky for T as its riskiest file.
- Maximum (test, file)-failure frequency (relative) ($f_{2,2}$) [3, 37, 54, 62, 68]: The relative frequency is calculated by dividing the (T, file)-failure frequencies by the number of times T has failed so far. For example, if test T failed a hundred times before R_t, and a file was part of the combined changeset half of the times, the relative frequency will be 0.5. Again, we determine the maximum across all files. This feature allows to discriminate between systematic and arbitrary co-occurrence of a changed file and a test failure.

The second hypothesis is: Files that were in the changeset when a test *transitioned*, are related to this test's outcome.

- Maximum (test, file)-transition frequency ($f_{2,3}$): Given a test T in R_t, we take the maximum (T, file)-transition frequency across all files in the combined changeset.
- Maximum (test, file)-transition frequency (relative) ($f_{2,4}$): We take the maximum relative (T, file)-transition frequency across all files in the combined changeset.

Even though we are not aware of prior work that proposes features $f_{2,3-4}$, they build on the existing understanding of test transitions [43] and failure-based features $f_{2,1-2}$.

(Test, File)-Similarity Features (F_3): These features embody lexical similarities between *names* and *paths* of a test and files in the changeset. This similarity proxies human perceived affiliation between a file and a test [45]. The hypothesis is: Conventions lead to tests and tested files with similar names and paths [72].

- Minimum file path distance ($f_{3,1}$) [11, 72]: We use the Levenshtein distance as proposed by White et al. [72] for test-to-code traceability links. Then, we determine the minimum distance, i.e., maximum similarity, across all files in the combined changeset.
- Maximum file path token similarity ($f_{3,2}$) [43, 45]: Based on the intuition of shared directories [43], we split file paths into tokens and count common tokens among test and file path. We determine the maximum similarity across all files in the combined changeset.
- Minimum file name distance ($f_{3,3}$) [72]: Similar to $f_{3,1}$, we use the minimum Levenshtein distance between a test name and each file name in the combined changeset.

Change Features (F_4): While the features described so far directly concern predicting the outcome of a *specific* test in a CI run R_t, there are also features that express how the introduced commits (i.e., *changes*), Δ_t, affect the failure likelihood level of *all* tests.

The first hypothesis is: Changes involving a larger number of distinct authors are more likely to cause failures [43, 52].

- Distinct authors ($f_{4,1}$) [43, 45]: Number of distinct authors within Δ_t, i.e., across all commits.

The second hypothesis is: Large changes are more difficult to review and therefore more error-prone [45].

- Changeset cardinality ($f_{4,2}$) [4, 45, 62]: Number of files in the combined changeset.
- Amount of commits ($f_{4,3}$) [4, 34]: Amount of commits in Δ_t, i.e., since last CI run.

The third hypothesis is: Certain file types are more likely to cause failures than others. As we want to provide a general methodology, we rather consider the variety of file extensions in a change, than specific file types [45, 62].

- Distinct file extensions ($f_{4,4}$) [45, 62]: Number of distinct file extensions in the combined changeset.

2.3 Predictive Modeling

Goal: Select and build effective and efficient ranking models.
In the following, we explain how we use a constructed dataset D to build ranking models for CI-RTP/S. Similar to related work, we target *point-wise* ranking models [11, 45]: Given a vector of feature values, i.e., one row in X, $x_{t,i}$, the model outputs a *score*, $\hat{y}$, between 0 and 1. $\hat{y}$ can be interpreted as a test's *estimated likelihood to fail* [45] and should be close to 1 for tests which are likely to fail. $\hat{y}$ is used to relatively rank tests yielding a test order as needed for RTP. For unsafe RTS, we can derive a *cut-off value*, $\theta \in \mathbb{R}_{[0,1]}$, based on some *cut-off criterion* (see Sec. 2.4.2). It defines the decision boundary to select the test for execution, if $\hat{y} > \theta$, or skip it otherwise. As any modeling technique that learns an accurate mapping from X to Y is suitable, we apply the following set of heuristic ranking models, $M_{h,f_{j,k}}$, and supervised ML classification algorithms, M_{1-5}.

2.3.1 Heuristic Ranking Models ($M_{h,f_{j,k}}$). Heuristic ranking models are widely applied across RTP and unsafe RTS research (e.g., [11, 13, 21, 60]; partly only used as baselines). The intuition is that these models, $M_{h,f_{j,k}}$, predict a failure likelihood solely by considering a single feature $f_{j,k}$. For example, a heuristic could select only those tests that have failed within the previous n CI runs [21]. The underlying ranking model orders tests only by the *last failure* feature, $f_{1,2}$, and selects tests based on a cut-off value n.

As the ranking model ultimately has to output a score $\hat{y} \in \mathbb{R}_{[0,1]}$, each value $x \in \mathbb{R}$ of the selected $f_{j,k}$ needs to be transformed. This is done by using a *min-max-scaler*: $\hat{y} = \frac{x - min(f_{j,k})}{max(f_{j,k}) - min(f_{j,k})}$. Since this inexpensive mathematical transformation requires close to no training effort, heuristic ranking models are naturally efficient. Consequently, in the given example, n must also be transformed by the scaler for $f_{1,2}$ to obtain θ.

2.3.2 Supervised Machine Learning (M_{1-5}). We draw the following five supervised ML classification algorithms from existing work on RTP and unsafe RTS based on how frequently they were used before (at least from two distinct authors): Logistic regression [57, 62] (M_1), Multi-layer perceptron [1, 46, 62] (M_2), Linear support vector machine (SVM) [11, 62][4] (M_3), Random decision forest [2, 8] (M_4), Gradient boosted trees [45, 62] (M_5).

These models' optimal performance will most likely depend on project-specific hyper-parameter tuning. There are manual, systematic, and random search techniques for finding the best set of hyper-parameters [7]. We show a straight-forward approach without hyper-parameter tuning in our empirical study (see Sec. 3.2).

2.4 Evaluation

Goal: Measure the cost-effectiveness of CI-RTP/S approaches for a given dataset D.

2.4.1 Cost-Effectiveness. Early RTS research classifies techniques among other attributes as *safe* and *precise*, if they select all potentially fault-revealing test cases in a modified program (i.e., effectiveness) by ignoring unnecessary test cases (i.e., cost) [64]. Similarly, the average percentage of faults detected per cost (APFD$_c$) metric is usually used to evaluate the fault detection capability (i.e., effectiveness) of RTP techniques with respect to test execution cost [17, 23]. More recent studies additionally employ traditional classification performance metrics, such as accuracy, F_1 score, or Area-Under-the-ROC-Curve (i.e., effectiveness) [4, 46, 62] and measure end-to-end test run-time including analysis overhead (i.e., cost) [26, 42, 79]. Furthermore, research on Pareto efficient multi-objective regression test optimization aims to find the optimal trade-off between multiple objectives, e.g., code coverage, historical fault detection capability, or execution cost [28, 74, 76, 77].

In line with these insights from research and discussions with practitioners from IVU Traffic Technologies, we conclude: Evaluation of CI-RTP/S cost-effectiveness requires considering the trade-off between the time (i.e., cost) invested in testing and the thereby achieved level of fault detection safety (i.e., effectiveness).

There is one caveat to this understanding which is the missing fault-to-failure mapping as we do not seed faults into programs, but use observed real-world failures, i.e., failing tests. We therefore rely entirely on detecting failures rather than faults, assuming a one-to-one mapping as proposed by previous research [50, 60, 69].

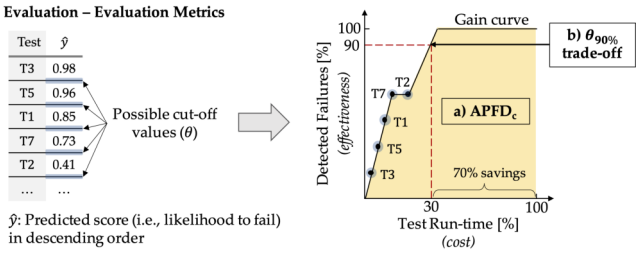


Figure 4: Process to derive the used evaluation metrics for a set of n tests with predicted failure scores ($\hat{y} \in \mathbb{R}^{1\times n}_{[0,1]}$) by using the gain curve.

2.4.2 Evaluation Metrics. Fig. 4 illustrates how evaluation metrics from prior research can be derived that reflect the cost-effectiveness trade-offs for CI-RTP/S: First, for a test suite scheduled for a CI run at timestamp t, $\mathcal{T}_t$, we obtain the failure scores, $\hat{y}$, predicted by the ranking model that is under evaluation. Second, these scores are ranked in descending order, creating a prioritized test suite $\mathcal{T}_t^*$. Third, for each possible cut-off value θ we draw a point into a coordinate system where the x-axis is the percentage of test run-time and the y-axis is the percentage of detected failures, both compared to *retest-all*, i.e., executing all tests. From these points, we can derive the following two commonly used metrics, (a) one for evaluating the total ranking as used for RTP and (b) one for unsafe RTS via some cut-off value θ:

(a) APFD$_c$. Connecting the points yields the so-called (cumulative) *gain curve* which can be further reduced to the area under the gain curve, a single aggregation measure between 0 and 1. This area is referred to as the APFD$_c$ where test costs, meaning the subscript c, are solely reflected by the test run-time [17]. Since the APFD$_c$ is an established *cost-aware* evaluation metric for RTP, we use it to assess the quality of the overall ranking model [13, 23, 60].

[4]Note that SVMs are non-probabilistic since they separate points into classes through hyperplanes. Yet, probability estimates, i.e., scores between 0 and 1, can be derived, e.g., by performing internal cross-validation [63].

(b) Cut-off Trade-offs. Yet, in the case of unsafe RTS, as opposed to RTP, reporting *only* the $APFD_c$ metric is insufficient, as we must ultimately select a subset of tests, $\mathcal{T}' \subseteq \mathcal{T}^*$. Setting the cut-off value θ depends on the acceptance criteria of a project's developers: For higher empirical safety[5], they will set the value of θ close to 0, whereas for low safety, but high time savings, it should be near 1. Therefore, to derive expected RTS cost-effectiveness trade-offs, we further measure the test run-time savings for three different empirical failure detection safety levels (90, 95, and 100%), i.e., three different cut-off values ($\theta_{90\%}$, $\theta_{95\%}$, and $\theta_{100\%}$). These levels are chosen based on the idea of empirical safety (100%), following Facebook's example (95%) [45], and using safety acceptance criteria expressed by our industry partner IVU (90%). Notably, this cut-off criterion is based on the idea of empirical safety levels [45] rather than cutting off tests based on *time constraints* [8, 71].

2.4.3 Model Training and Testing. The proposed ranking models need to be trained on a subset of D before evaluating the trained model on a *different hold-out subset* of D. We delineate this process of model training and testing in the following four steps: Defining training and test splits, model training, model testing, and randomness. Note that *testing* in this context means that the model is examined for its performance on a hold-out test dataset, i.e., on data that was not available during model training.

Training and Test Splits. There is a myriad of ways to split a dataset into training and test set. For instance, for unsafe RTS, Machalica et al. [45] use the most recent week of their three months dataset for testing. Philip et al. [62] train on one year and test on two months of data. There is no general rule, neither about how much data should be used for training and testing, nor about the ratio. Yet, in practice, we need to decide whether to use all available historical data for training or stick to more recent data, which might resemble the *current* failure behavior more accurately [3].

Fig. 5 shows the training-test-splits which we use to measure these possible influences: We divide a time-ordered dataset, D, into 5 equal-sized folds by CI runs. This is based on the widespread 5 fold split in ML research (e.g., [9]). One could also use *absolute* amounts of data for splitting, e.g., always test on one month of data, but there is no argumentation to pick one over another. Notably, we cannot perform cross-validation since there are temporal dependencies between results of CI runs: Training models on past data and testing on more recent data is realistic and most suitable in practice [62].

We obtain different training-test-splits as follows: First, to investigate the best *ratio of training to test folds*, we vary the amount of training folds while keeping the test set at the most *up-to-date* fold. The derived splits (S_{1-3}) with different ratios use 100, 75, and 50% of historical data for training. Recall that we regard the training-test-ratio, i.e., amount of training data, as the first (i) of three parameters of any CI-RTP/S approach that needs to be calibrated.

Second, to examine the sensitivity of a CI-RTP/S approach to data *timeliness*, we extend S_{1-3} by three additional training-test-splits. The intuition is that the *up-to-date* fold, which we use as a test set, might not be representative. However, if the CI-RTP/S approach works well across test sets with different timeliness, we

[5]Due to the lack of deterministic test execution traces or static dependencies, CI-RTP/S can only give *empirical*, i.e., statistical evidence-based, safety guarantees.

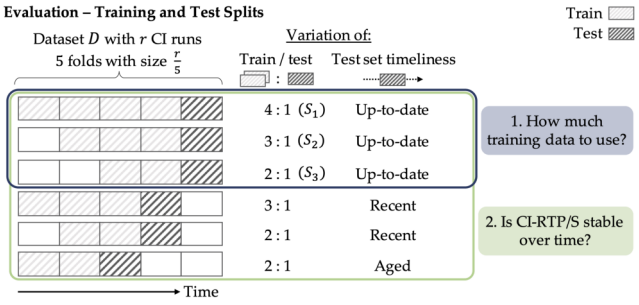


Figure 5: Splitting D into different training and test sets.

can provide better predictability of CI-RTP/S to practitioners, due to stable performance over time. Alternative test sets, i.e., with timeliness *recent* and *aged* (see Fig. 5), may also allow to derive a performance estimation in case there are no failures in the *up-to-date* fold, which disallows constructing the gain curve.

In total, this leaves us with 6 different training-test-splits which we need to evaluate. We refer to a training set as D^{train} and to a test set as D^{test} in the following.

Model Training. The heuristic ranking models, $M_{h,f_{j,k}}$, are created by fitting min-max-scalers on each of the 16 features in D^{train} as described in Sec. 2.3.1. These scalers are then used to predict the failure score for each test $T_{t,i}$ in D^{test} solely based on $f_{j,k}$ in D^{test}. All scores are clipped to be between 0 and 1 resulting in a score $\hat{y} \in \mathbb{R}_{[0,1]}$ for each test. Since a feature might follow an inverse scoring order, where a high feature value indicates a low failure score, we calculate $APFD_c$ values for $\hat{y}$ and $1 - \hat{y}$ on the training set and only use the better performing one for model testing.

Additionally, for each project, we train a ranking model for each ML algorithm and feature set as well as on the composition of all four feature sets. This results in $(4 + 1) * 5 = 25$ (feature set, model)-combinations to be evaluated per training-test-split.

Model Testing. Each created ranking model is used to predict the failure score of each test $T_{t,i}$, i.e., each row, in D^{test}. If the predicted failure scores of two tests are equal, the test with the shorter last execution duration is executed first, since the last test duration has proven to be a reasonable baseline [13, 60]. Thereby, we obtain a test ranking for each CI run in D^{test}. We follow prior research [11, 50, 60] by reporting our evaluation metrics averaged across all CI runs in D^{test} that contained failures: *Avg. $APFD_c$* (RTP) and *avg. test time savings* (RTS).

Randomness. Several ML algorithms involve randomization. We repeat the experiments with 30 different random seeds to reduce the impact of randomness [23, 71]. Results in our empirical study (see Sec. 3.3) report the mean of evaluation metrics.

3 EMPIRICAL STUDY

We perform an empirical study to evaluate CI-RTP/S approaches built and calibrated using our methodology; and to derive evidence-based guidelines and cost-effectiveness expectations for practitioners. Therefore, we strive to answer the following research questions (RQs):

- **RQ**$_1$: How sensitive is the cost-effectiveness of CI-RTP/S to different parameterizations regarding amount of training data, choice of features, and ranking model?
- **RQ**$_2$: How do CI-RTP/S approaches built with our methodology compare against baseline RTP and RTS techniques in terms of cost-effectiveness?

3.1 Study Subjects

Table 1 lists the selected 23 software projects from industry (3) and open-source development (20).

3.1.1 Industrial Projects. These are provided by our industry partner IVU Traffic Technologies, each counting several millions of source lines of code (SLOC). One project is primarily written in C/C++ (P_1), two in Java (P_{2-3}). Additionally, web-based and native graphical user interface (GUI) clients are part of these code bases which are programmed in different domain specific languages (DSLs) or other general purpose programming languages (GPLs) such as JavaScript. Their test suites, besides unit testing, involve integration- as well as system-level testing often performed across project boundaries. Developers commit their changes directly to the main VCS development line, where the company-internal Jenkins CI system collects commits and triggers a new *retest-all* CI run once the previous run has finished. Once a test run is finished, a Jenkins plugin aggregates all XUnit test results into a structured test report in XML or JSON format[6].

3.1.2 Open-source Projects. These are part of a recently published dataset for RTP, RTPTorrent (Mattis et al., Zenodo, CC BY 4.0[7]), that aims to deliver a representative sample of all Java projects on GitHub [50, 51]. We discovered that most of them (14/20) additionally use more GPLs other than Java (e.g., C++, Python) or DSLs (e.g., SQL, YAML). As RTPTorrent is yet missing some required links between CI runs and respective VCS commits, we used the underlying massive TravisTorrent [6] CI dataset to extend RTPTorrent. If there are multiple VCS branches that are tested in the CI system, we use historical data from *all* of these branches. The same applies for multiple sub-stages in a test stage, where different sub-stages, e.g., for different compiler versions, might report the same failures. In the worst case, this leads to an over- or undersampling of failures. We still keep these data to not waste potentially valuable information.

3.1.3 Datasets. We argue that this set of projects resembles reality, where RTP and RTS techniques have to cope with multi-language software of varying size as well as test-levels, i.e., unit-, integration, or system-level [12, 80]. More than 37,000 CI test logs and 76,000 VCS commits were analyzed. We publish the resulting 23 datasets, $D_{P_{1-23}}$, as part of our supplemental material.

3.2 Experimental Setup

Recall that we identify CI-RTP/S approaches as triples of the parameters (i) training data amount (i.e., training-test-ratio), (ii) features, and (iii) ranking model. All studied settings for these parameters described in Sec. 2 are summarized in Table 2. Due to the combinatorial explosion of assessed study subjects, random seeds, training-test-splits, features (or feature sets), and ranking models, the experiments were run on a highly parallelized cluster infrastructure. The measured total CPU time was more than 50,000 hours.

Since we aim to constitute a generic example of applying our methodology, we do not perform elaborate hyper-parameter grid search for ML algorithms. Instead, we follow Chen et al. [13] and stick to the default model hyper-parameters provided in the *scikit-learn* package [59], but use the *LightGBM* package for a more lightweight implementation of gradient boosting [35]. Notably, before model training each feature is normalized.

In our supplemental material, we provide the source code necessary to reproduce our results from the created 23 datasets, $D_{P_{1-23}}$.

3.3 Results

In the following, we discuss the empirical results and address the RQs. Detailed results are provided with the supplemental material.

3.3.1 RQ$_1$: Cost-Effectiveness Sensitivity Analysis. We aim to analyze how sensitive the cost-effectiveness of the CI-RTP/S approaches, built with our methodology, is to the parameters (i)-(iii) and find calibrations that are empirically superior to others (see Table 2). When comparing different parameter settings, we use the APFD$_c$ as calculated for each CI run in a project's test dataset D^{test} and average it over these runs to obtain the avg. APFD$_c$. This metric is then considered across projects for sensitivity analysis. We run a one-way analysis of variance (ANOVA) for each parameter to investigate its individual influence on the avg. APFD$_c$. Therefore, we vary its value, while having the other two parameters at their project-specific best setting. Our approach for sensitivity analysis of parameterization follows related work on RTS and RTP [8, 60].

(i) Training Data Amount. To check how much training data is beneficial for CI-RTP/S, we study whether there are significant differences in the means of the avg. APFD$_c$ across projects for S_{1-3}. To choose the appropriate statistical test for the ANOVA, we first perform the Shapiro-Wilk test with Bonferroni correction to check for normality which cannot be rejected with a minimal p-value of 0.045 (significance level $\alpha = 0.05$ corrected by $|S| = 3$ is $\alpha_{norm} = 0.017$). We then use Bartlett's test for homoscedasticity, which is also not rejected at a p-value of 0.977. Hence, with normal and homoscedastic data we can perform a repeated measures ANOVA. We fail to reject the null hypothesis (p-value = 0.717), indicating no significant difference between the mean values of S_1, S_2, and S_3. Among them, S_2 shows the highest mean avg. APFD$_c$ of 0.896.

Prior research has trained ranking models only on *faulty* CI runs [11], that is runs that contain at least one test failure, or by using *all* available CI runs [45]. To check if there are any significant differences, we repeated all experiments a second time, but this time we only trained on those CI runs in the training set that contained failures. Using the same procedure as before, we find that differences in means are insignificant (p-value = 0.284) by using a paired t-test, which is suitable as data is normally distributed (p-value = 0.059) and we have two populations, that is *only faulty* and *all* CI runs.

Overall, we can summarize that using less training data does not harm cost-effectiveness of CI-RTP/S. Even limiting ourselves to only using faulty CI runs did not negatively impact the avg. APFD$_c$.

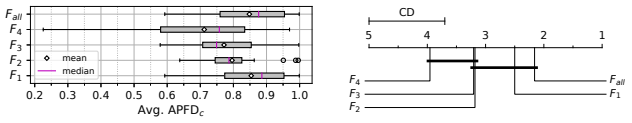
[6] Jenkins JUnit Plugin: https://plugins.jenkins.io/junit/
[7] CC Attribution 4.0 International: https://creativecommons.org/licenses/by/4.0/

Table 1: Study subject statistics

P_{ID}	Project	# SLOC	Time period [days]	# Commits	# CI runs	# Failing CI runs	# Test runs	# Tests	# Failures	Avg. test stage duration [sec]	Avg./median # failures per failing test stage
P_1	IVU_Cpp	≫ 1M	267	8,632	3,996	2,841	3,608.4K	1,240	25,973	5,454	9.1/3.0
P_2	IVU_Java_1	≫ 1M	313	7,747	943	876	178.7K	279	14,568	65,557	16.6/6.0
P_3	IVU_Java_2	≫ 1M	699	7,965	3,209	1,521	3,526.3K	1,278	7,603	5,330	5.0/2.0
P_4	jcabi-github	64K	872	1,050	809	205	398.1K	201	740	778	2.2/2.0
P_5	jade4j	10K	1,539	400	358	59	35.9K	46	1,323	4	13.8/19.0
P_6	optiq	243K	395	560	458	38	55.3K	63	110	2,168	1.6/1.0
P_7	buck	562K	307	2,517	846	339	586.1K	864	1,511	1,650	4.5/1.0
P_8	jetty.project	346K	63	237	192	174	63.9K	787	415	508	1.3/1.0
P_9	jsprit	59K	368	326	267	14	91.8K	107	123	23	2.4/1.0
P_{10}	LittleProxy	13K	1,580	353	271	62	11.0K	50	172	134	2.8/2.0
P_{11}	dynjs	57K	1,163	517	385	25	68.5K	83	496	15	12.1/1.0
P_{12}	sling	673K	213	13,376	1,403	812	268.1K	304	1,158	420	1.4/1.0
P_{13}	HikariCP	13K	661	1,787	1,575	125	44.0K	23	383	58	3.1/1.0
P_{14}	wicket-bootstrap	42K	1,245	1,150	904	342	41.4K	91	9,007	8	26.3/29.0
P_{15}	okhttp	69K	1,423	3,518	3,412	744	236.5K	266	939	108	1.2/1.0
P_{16}	titan	59K	747	621	384	157	43.3K	107	551	2,366	2.2/2.0
P_{17}	deeplearning4j	138K	727	1,071	982	566	14.6K	174	908	477	1.6/1.0
P_{18}	cloudify	132K	909	6,048	4,973	496	283.6K	116	602	92	1.2/1.0
P_{19}	graylog2-server	127K	1,381	5,414	3,891	165	798.5K	250	403	792	1.6/1.0
P_{20}	Achilles	54K	1,114	904	642	23	139.9K	627	162	139	6.0/2.0
P_{21}	DSpace	384K	1,043	2,489	1,929	82	122.1K	83	1,697	130	20.7/35.0
P_{22}	sonarqube	661K	532	7,899	4,286	488	6,696.0K	3,122	2,156	334	3.5/1.0
P_{23}	jOOQ	351K	961	1,525	1,318	403	81.5K	51	573	13	1.1/1.0

Table 2: Parameters of CI-RTP/S approaches: (i) Training data amount S, (ii) features F, (iii) ranking models M

S_1	100% of available historical data
S_2	75% of available historical data
S_3	50% of available historical data
F_1	Failure count ($f_{1,1}$), Last failure ($f_{1,2}$), Transition count ($f_{1,3}$) Last transition ($f_{1,4}$), Avg. test duration ($f_{1,5}$)
F_2	Max. (test, file)-failure freq. ($f_{2,1}$), Max. (test, file)-failure freq. (rel.) ($f_{2,2}$), Max. (test, file)-transition freq. ($f_{2,3}$), Max. (test, file)-transition freq. (rel.) ($f_{2,4}$),
F_3	Min. file path distance ($f_{3,1}$), Max. file path token similarity ($f_{3,2}$), Min. file name distance ($f_{3,3}$)
F_4	Distinct authors ($f_{4,1}$), Changeset cardinality ($f_{4,2}$), Amount of commits ($f_{4,3}$), Distinct file extensions ($f_{4,4}$)
M_1	Logistic regression
M_2	Multi-layer perceptron
M_3	Linear SVM
M_4	Random decision forest
M_5	Gradient boosted trees
$M_{h,f_{j,k}}$	Heuristic ranking models



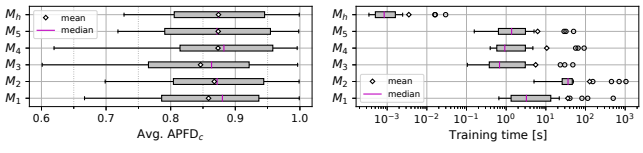
(a) Avg. APFD$_c$ across projects **(b) Pairwise comparison**

Figure 6: Sensitivity analysis of *features* F (ii)

(ii) Features. In Sec. 2.2 we described how features are grouped into feature sets F_{1-4} to increase comprehensiveness. This allows us to study their individual cost-effectiveness and empirical differences among them. Fig. 6a shows the distribution of the avg. APFD$_c$ for each feature set F_{1-4} and F_{all}. Again, we assume normality after conducting a Shapiro-Wilk test with Bonferroni correction which yields a minimal observed p-value of 0.023 ($\alpha_{norm} = 0.01$). Bartlett's test for homoscedasticity is further rejected at p-value 0.02, assuming heteroscedasticity. Thus, we use the non-parametric ANOVA Friedman test to check for differences in the means of avg. APFD$_c$. It is rejected at p-value 0.002 indicating that there are significant differences which we further explore with the *post hoc* Nemenyi test as proposed by Demšar [14]: It compares all feature sets pairwise based on the absolute differences of their avg. rankings. For α a *critical difference (CD)* is determined; if the avg. ranking difference is greater than CD, the null hypothesis that they have equal performance is rejected. Fig. 6b visualizes these pairwise differences in the avg. ranks through a CD diagram: If two feature sets are connected by a horizontal bar, they are not significantly different to each other [32]. The diagram indicates that F_4 is significantly worse than F_{all} (best in pairwise comparison) and F_1 (best among F_{1-4}), questioning the usefulness of features in F_4. Further, we find that even though differences in avg. ranks are not significant, F_1 and F_{all} have higher mean avg. APFD$_c$ than other feature sets by at least 0.05. This emphasizes the usefulness of features from test history (F_1).

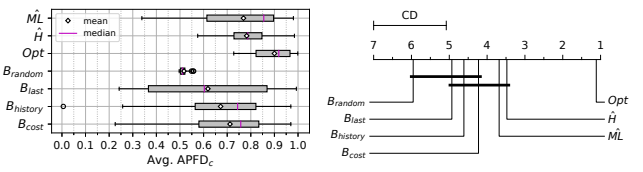
As heuristic ranking models do not rely on entire feature sets but only on single features, we conduct the sensitivity analysis for all single features $f_{j,k}$ as well, i.e., their respective heuristic ranking models $M_{h,f_{j,k}}$. The ANOVA shows that there are no significant differences between features (Shapiro-Wilk test rejected at p-value 0.001; Friedman test not rejected at p-value 0.063). However, from the five features with highest mean rank and median avg. APFD$_c$, three are from feature set F_1 ($f_{1,4}$, $f_{1,1}$, $f_{1,5}$), and two from F_2 ($f_{2,1}$, $f_{2,2}$), with the best one being $f_{2,1}$, i.e., max. (test,file)-failure frequency. Again, this emphasizes that features using test history correlate with better cost-effectiveness. There seems to be some combination of features that leads to superiority of F_1 over F_4. This observation motivates the use of more elaborate statistical feature selection techniques in the future (see Sec. 3.6).

(iii) Ranking Model. To investigate the cost-effectiveness of each ranking model M, we perform ANOVA twice, with the response variables avg. APFD$_c$ and *training time*, respectively. We consider the latter as a reasonable proxy for model efficiency as our experimental results show that model inference time is negligibly small (see experiment results in supplemental material). Fig. 7 shows the distributions of the avg. APFD$_c$ and the training times of ranking



(a) Avg. APFD$_c$ across projects **(b) Training time across projects**

Figure 7: Sensitivity analysis of *ranking models* M (iii)



(a) Avg. APFD$_c$ across projects **(b) Pairwise comparison**

Figure 8: Comparison of RTP cost-effectiveness

models across all projects. For the first ANOVA, we perform a repeated measures ANOVA, as the data is normal (p-value = 0.087) and homoscedastic (p-value = 0.953). Since the null hypothesis is not rejected at p-value 0.066, we assume that there are no statistically significant differences in the mean avg. APFD$_c$. M_5 (gradient boosted trees) and M_h have the highest mean avg. APFD$_c$ (both 0.874). The second ANOVA shows significant differences in training time: We reject the Friedman test at p-value <0.001 (non-parametric ANOVA due to non-normality at p-value <0.001). Without further inspection, it is obvious that M_h (as expected) is far more efficient than the ML algorithms as its training procedure is simply a mathematical transformation. Yet, despite its simplicity, the cost-effectiveness of M_h is still comparable.

While prior research also finds M_5 to be particularly effective [45], interestingly, these findings rather suggest focusing on existing simple heuristic ranking models (M_h) instead of investing the effort in training complex ML models from prior research.

> *RQ$_1$: We find that CI-RTP/S cost-effectiveness is sensitive to the choice of features, but is not significantly impacted by the amount of training data or the ranking model. We empirically determine that the best approaches contain features from test history and use heuristic ranking models.*

3.3.2 RQ$_2$: Comparative RTP and RTS Performance. We aim to provide estimations on the cost-effectiveness of CI-RTP/S approaches for RTP and unsafe RTS. We have motivated in Sec. 1 that it is of particular interest for practitioners to know how much cost-effectiveness is sacrificed if using only a semi-optimally calibrated approach. Thus, we compare the following CI-RTP/S approaches including four baselines for RTP and unsafe RTS.

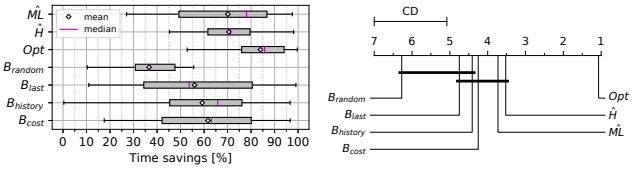
- $\hat{\boldsymbol{ML}}$: Empirically best ML ranking model from RQ$_1$, i.e., M_5 (gradient boosted trees) with F_{all} on S_2 (75% training data)
- $\hat{\boldsymbol{H}}$: Empirically best heuristic ranking model from RQ$_1$, i.e., $M_{h,f_{2,1}}$ (max. (test,file)-failure freq.) on S_2 (75% training data)
- ***Opt***: Always uses optimally calibrated approach from our methodology for each project; implies high effort in practice, as all combinations of parameter settings are computed.
- $\boldsymbol{B_{random}}$: Baseline ranking tests in random order [11, 17, 20]
- $\boldsymbol{B_{last}}$: Baseline ranking tests in ascending order by the time since the last failure (i.e., $f_{1,2}$) [21, 71]
- $\boldsymbol{B_{history}}$: Baseline ranking tests in descending order by the amount of historical failures (i.e., $f_{1,1}$) [3, 60]
- $\boldsymbol{B_{cost}}$: *Cost-only* baseline ranking tests in ascending order by their last execution time [13, 60]

The distribution of their avg. APFD$_c$ across projects is shown in Fig. 8a, reflecting the cost-effectiveness for RTP. To compare the median values, we perform the non-parametric ANOVA Friedman test assuming non-normality at minimal p-value of 0.001 (α_{norm} = 0.007). It is rejected at p-value <0.001 indicating that there are significant differences. Again, we use the *post hoc* Nemenyi test for pairwise comparison of avg. ranks and report the results in Fig. 8b. We can see that *Opt* significantly outperforms all other approaches, which is also reflected by the highest median avg. APFD$_c$ of 0.919 with the lowest median absolute deviation of 0.1. Although not statistically significant, the medians of the semi-optimally calibrated approaches $\hat{ML}$ (0.855) and $\hat{H}$ (0.787) are also better than all baselines. Notably, the spread of the avg. APFD$_c$ for $\hat{H}$ with respect to median absolute deviation is considerably smaller (0.177) than the one for $\hat{ML}$ (0.239). Similar to prior research [60], B_{cost} seems to be the best performing baseline (median: 0.757), yet not statistically significant across studied projects. Moreover, all approaches and baselines (partly significantly) outperform random ordering (B_{random}), which is also in line with previous results (see Fig. 8b) [11, 18, 19, 71]. Interestingly, we found that for 5 and 11 projects, respectively, the baselines $B_{history}$ and B_{last} performed worse than $M_{h,f_{1,1}}$ and $M_{h,f_{1,2}}$ which use the same features, but followed an inverse scoring order in these projects.

Regarding RTS, Fig. 9a shows the distribution of the avg. time savings for RTS across projects when setting the cut-off value to $\theta_{90\%}$, i.e., 90% empirical failure detection safety. The ANOVA, reported in Fig. 9b, has similar results (Shapiro-Wilk: p-value 0.029; Bartlett: p-value <0.001; Friedman: p-value <0.001), which is not surprising, as we generally expect good RTP to correlate with good RTS approaches. Though the project-specific best approach used for *Opt* is not necessarily the same for RTS and RTP: For $\theta_{90\%}$, 19 out of 23 projects have the same best project-specific approach.

Overall, using *Opt* we are able to save 84% of testing time on average across projects. However, even with the semi-optimally calibrated approaches, $\hat{H}$ and $\hat{ML}$, savings of on average >70% are achieved. While the baselines have relatively high average savings as well (up to 61.7%), they suffer a large spread across projects. For $\theta_{95\%}$ and $\theta_{100\%}$, we find 83.1 ± 13.8% and 82.8 ± 14.4% average test time savings with *Opt*, respectively. $\hat{H}$ and $\hat{ML}$ achieve average savings of 69.8±10.2% and 69.7±14.2% for $\theta_{95\%}$ and 69.4±10.5% and 69.4 ± 14.3% for $\theta_{100\%}$. The overall conclusions regarding relative performance of the seven compared approaches (including four baselines) remain similar. All numbers and figures for $\theta_{95\%}$ and $\theta_{100\%}$ are part of the provided supplemental material.

Finally, we investigate to what extent the ranking performance of *Opt*, $\hat{H}$, and $\hat{ML}$ (as they have been calibrated on the *up-to-date*



(a) Avg. savings across projects **(b) Pairwise comparison**

Figure 9: Comparison of RTS cost-effectiveness for $\theta_{90\%}$

Table 3: Time stability measured by avg. APFD$_c$ across 6 different training-test-splits ($\mu \pm \sigma$)

P_{ID}	*Opt*		$\hat{H}$	$\hat{ML}$
P_1	(F_{all}, M_4)	0.97 ± 0.00	0.74 ± 0.02	0.94 ± 0.02
P_2	$(f_{1,2}, M_h)$	0.90 ± 0.03	0.80 ± 0.02	0.88 ± 0.03
P_3	(F_{all}, M_4)	0.95 ± 0.01	0.77 ± 0.03	0.81 ± 0.06
P_4	(F_1, M_1)	0.84 ± 0.07	0.81 ± 0.08	0.64 ± 0.15
P_5	$(f_{3,3}, M_h)$	0.87 ± 0.16	0.72 ± 0.07	0.65 ± 0.22
P_6	$(f_{1,4}, M_h)$	0.64 ± 0.11	0.70 ± 0.11	0.65 ± 0.23
P_7	$(f_{1,5}, M_h)$	0.98 ± 0.01	0.91 ± 0.02	0.87 ± 0.10
P_8	(F_{all}, M_1)	0.86 ± 0.10	0.75 ± 0.12	0.81 ± 0.08
P_9	(F_{all}, M_1)	0.70 ± 0.30	0.81 ± 0.09	0.72 ± 0.25
P_{10}	(F_2, M_4)	0.63 ± 0.10	0.63 ± 0.10	0.42 ± 0.10
P_{11}	$(f_{3,3}, M_h)$	0.80 ± 0.21	0.83 ± 0.17	0.46 ± 0.30
P_{12}	(F_{all}, M_5)	0.86 ± 0.09	0.76 ± 0.12	0.86 ± 0.09
P_{13}	(F_4, M_5)	0.79 ± 0.05	0.69 ± 0.06	0.56 ± 0.08
P_{14}	$(f_{1,2}, M_h)$	0.79 ± 0.04	0.80 ± 0.02	0.72 ± 0.08
P_{15}	(F_1, M_2)	0.82 ± 0.05	0.81 ± 0.02	0.77 ± 0.09
P_{16}	(F_1, M_1)	0.69 ± 0.09	0.70 ± 0.06	0.68 ± 0.04
P_{17}	$(f_{1,1}, M_h)$	0.79 ± 0.11	0.80 ± 0.07	0.76 ± 0.10
P_{18}	(F_{all}, M_5)	0.85 ± 0.10	0.61 ± 0.10	0.85 ± 0.10
P_{19}	$(f_{3,3}, M_h)$	0.90 ± 0.17	0.90 ± 0.18	0.81 ± 0.15
P_{20}	$(f_{3,1}, M_h)$	0.90 ± 0.07	0.84 ± 0.08	0.67 ± 0.30
P_{21}	(F_1, M_5)	0.70 ± 0.14	0.75 ± 0.04	0.56 ± 0.14
P_{22}	(F_2, M_1)	0.67 ± 0.06	0.72 ± 0.03	0.53 ± 0.06
P_{23}	(F_{all}, M_4)	0.96 ± 0.00	0.86 ± 0.03	0.92 ± 0.00
Avg.		0.82 ± 0.07	0.77 ± 0.04	0.72 ± 0.08

test set) is stable over different test sets in time (see Fig. 5). Table 3 lists the mean and standard deviation of their avg. APFD$_c$ for each project, if applied across all 6 available training-test-splits. While *Opt* is always the best approach on the *up-to-date* test set, it is not necessarily the best one averaged across all training-test-splits. If there were two approaches performing equally well on the *up-to-date* test set in terms of their avg. APFD$_c$, hence being candidates for *Opt*, we decide in favor of the one with smaller standard deviation of the APFD$_c$. The cost-effectiveness oscillates considerably over time with an average σ of 0.07 for *Opt*, σ of 0.08 $\hat{ML}$, and 0.04 for $\hat{H}$. Hence, we conclude that re-adaptation intervals should be kept short, as optimal calibration of CI-RTP/S fluctuates over time.

> *RQ$_2$: We find that CI-RTP/S approaches outperform established baselines and save on average 84% of test run-time while retaining 90% of empirical failure detection safety. However, CI-RTP/S is unstable over time, thus requiring regular adaptation.*

3.4 Guidelines and Expectations for Practice

In summary, we derive the following practical implications from the findings of our empirical study:

(1) CI-RTP/S does not need large amounts of training data per se. It suffices to use the most recent or even only faulty CI runs. This speeds up re-adaptation and decreases required storage.
(2) Features from test history are frequently performing well, yet, in our experiments, adding VCS metadata can increase cost-effectiveness: F_{all} has been the best feature set.
(3) Rather naïve, inexpensive heuristic ranking models often outperform sophisticated ML algorithms.
(4) Calibrating the project-specific *optimal* (*Opt*) CI-RTP/S approach gives significant cost-effectiveness benefits over semi-optimally calibrated approaches or baseline models.
(5) CI-RTP/S approaches are not stable over time. Frequently re-adapting CI-RTP/S to more recent development is advisable.
(6) Unsafe RTS, even if solely based on metadata from CI and VCS, can achieve considerable test run-time savings (on average 84% while detecting 90% of failures across projects from our study).

3.5 Application in Industry

Besides the empirical results on the performance expectations of CI-RTP/S reported above, we share some initial experiences and challenges from deploying our methodology at IVU Traffic Technologies who supported and partially sponsored this research.

We implemented our methodology as a web service that is deployed in the company's infrastructure and integrated with their Jenkins CI system. As described in Sec. 3.1.1, the existing main Jenkins pipelines continuously execute all regression tests. Depending on the project this is either done in random order, to detect and prevent test order dependencies (see [39]), or by the alphabetical naming order of tests. In addition to the existing pipeline, we created a parallel RTS pipeline for project P_2 from our empirical study: This pipeline first queries the web service with the introduced changeset since the last CI run and a desired empirical safety level (the default is 90%) and then only executes the subset of tests retrieved from the web service. The reason why we choose this parallel setup for now, is to build up trust in the RTS mechanism among developers. They can directly compare results from the existing (safe) *retest-all* to the RTS pipeline, which makes test time savings transparent.

In this industry setting, we decided to *only* include *heuristic* ranking models ($M_{h,f_{j,k}}$) for the following reasons: First, as we have shown empirically, these heuristics often outperform complex ML algorithms and require low training effort in both time and computation resources. They are also non-randomized, which eliminates the need for costly repeated experiments. Second, they are easily interpreted by developers, who are not necessarily experts in predictive modeling, which might increase overall acceptance of the used models. As our guidelines from the empirical study suggest, the web service re-adapts all ranking models every night including the new data from the last day which are fetched from Jenkins. For the subsequent day, the web service will then only use the best performing model to rank tests. We follow the assumption by Facebook [45] that model performance on our test dataset is a good approximation of the model's general performance on unseen data. Still, to regularly check this assumption, we store trained models and inspect in hindsight how they performed on the following day, i.e., if the empirical safety level carried over from the test set.

Our parallel RTS pipeline has been in use in project P_2 for six weeks. This project contains a large fraction of relatively *long running* Java tests that operate across language boundaries (Java and C++) and often have long test setup times for database schemas.

During the considered time period, the RTS pipeline executed 366 CI runs, where 176 included at least one failure. Across those failing CI runs, the realized test time savings were on average 19.8% with 93.4% of failures being detected. We observe that the empirical safety level was only notably violated (below 90%) when the code base underwent major refactorings which were accompanied by a sudden increase in failures. However, the test time savings are considerably smaller than what we achieved for P_2 during the empirical study and for the open-source projects. There are (at least) two possible reasons for this observation: First, we found that the setup times often significantly impact the test execution time and created database schemas are cached and re-used for subsequent tests. Hence, even if certain tests are excluded, the considerable impact of the test setup time will still be prevalent if *any* of the tests requiring that setup is executed. Second, if multiple tests failed in a CI run, we observed an increase in the number of selected tests on the following days. We expect this to be a consequence of our decision to rely on heuristic ranking models only: $M_{h,f_{1,2}}$ has been the best for P_2 and thus, if multiple tests fail once, they will be selected for execution throughout the subsequent days which negatively affects the test time savings.

Even though these initially realized time savings are smaller than in the empirical study, IVU engineers are positive about the achieved results and will deploy parallel RTS pipelines for more projects. They expect test setup times to have significantly less impact in their other projects. In fact, the test setup caching mechanism is very project-specific and little prior research exists on test dependency aware RTP and RTS [39]. Thus, while our empirical study of open-source projects establishes comparability, context-specific practical challenges can impact CI-RTP/S cost-effectiveness and we will further investigate how to address them at IVU.

In the next step, we aim to reduce the frequency of *retest-all* cycles. Instead of parallel execution, the RTS pipeline will be the default CI pipeline. Re-executing all tests in certain intervals will still be required as the thereby obtained test outcome data are necessary for re-adapting ranking models.

3.6 Threats to Validity

3.6.1 External Validity. The main threats to external validity concern the representativeness of results. We address them by studying a heterogeneous set of real-world projects, but cannot, by nature of empirical studies, easily generalize our findings beyond our dataset. Since this is a known limitation of ML models as they always depend on the dataset quality, we followed established data mining and ML practices for splitting data, repeating randomized experiments, and training and testing models to mitigate these threats.

The investigated time periods of CI and VCS history might contain irregular development behavior, e.g., unusual maintenance activities. Therefore, we create multiple training-test-splits and investigate time stability of performances.

Since we rely on test execution time as reported in the CI logs, there is a threat from fluctuations within one CI test run due to irregular workload on the build machine. Tests are usually run in isolation inside CI environments to reduce such side-effects, but it might still affect the concrete values of reported evaluation metrics. Similar to prior work [60], we address this threat by our large dataset of CI runs and by conducting rigorous statistical analyses to ensure that findings are significant across projects.

Furthermore, we deliberately exclude an automated feature selection process or model hyper-parameter tuning. While this might limit the performance of ML algorithms compared to more sensible tuning, it enables us to perform fine-grained sensitivity analyses. We deem the investigation of automated feature selection techniques as an important future task to prune our current feature sets. In addition, while we focus on point-wise ranking models, there are recent studies on other approaches such as reinforcement learning (RL), which are beyond the scope of this work [8, 71].

As described in Sec. 2.4.1, we rely on a one-to-one failure-to-fault mapping similar to previous research [50, 60, 69]. While this assumption might distort results since faults often cause multiple failures, prior research on RTP shows that different mappings still lead to similar overall conclusions [60].

Finally, the presence of flaky tests may impact the effectiveness of CI-RTP/S. Existing research is not unequivocal regarding the expected effect of flaky tests: While at Facebook [45], the presence of flaky tests does not preclude the applicability of CI-RTP/S, Peng et al. [60] see substantial impact for some RTP techniques. We argue that flaky test detection requires special efforts and can be performed on top of our methodology. Due to resource constraints, we cannot re-run more than 37,000 CI test histories multiple times to *deflake* each dataset as proposed by prior work [45]. At IVU, flaky tests are currently not documented, yet developers are encouraged to fix such tests immediately when they behave non-deterministically.

3.6.2 Internal Validity. We identify the integrity of exploited data sources as well as the correctness of the implemented feature engineering and evaluation analysis as the main internal threat. To address this, we wrote run-time assertions and unit tests that discover invalid data and check feature computations. Furthermore, we manually checked results for their validity with IVU engineers.

4 RELATED WORK

Several RTP and unsafe RTS techniques have been proposed which incorporate other information than traditional white-box program analyses to predict test failures and rank tests. Throughout this paper, we have referenced existing research that we have consolidated into our methodology. While we focus on techniques that solely rely on CI and VCS metadata (CI-RTP/S), there is also significant related work which uses other additional information that is non-guaranteed in CI settings.

Studies on techniques that use such additional information *beyond* CI and VCS metadata have shown their effectiveness in specific contexts (i.e., single projects or organizations) [1, 2, 4, 11, 45, 47, 48, 54, 57, 62] as well as across multiple projects [5, 8, 13, 29, 33, 36, 40, 46, 49, 53, 56, 57, 60, 67]. We consider the following to be most relevant for our work: Machalica et al. [45] report a reduction of testing infrastructure cost by 50% and test executions by >66% at Facebook while retaining 95% empirical failure detection safety. They train a failure prediction ML model on features from CI and VCS metadata as well as static build dependencies and project identifiers. Busjaeger and Xie [11] train a linear SVM on black- and white-box (i.e., code coverage) features obtained from Salesforce's code repository and CI system. By ranking and selecting tests, they

achieve a trade-off of executing 3% of all tests to detect 75% of the failures. Chen et al. [13] train ML models to predict the effectiveness of RTP techniques by using features from test coverage and testing time. They find that there are no universally optimal RTP techniques across projects which supports findings in the related field of defect prediction [82]. Bertolino et al. [8] compare ten ML algorithms for ranking tests to provide guidelines on when to choose RL over supervised ML or vice versa. They use features from white-box code and dependency analysis as well as test history, and evaluate algorithms' performance on six open-source subjects with artificial faults. Henard et al. [29] provide an experimental comparison of 10 white-box and 10 black-box RTP techniques from prior research. In their study on five C programs from the software infrastructure repository [15] with seeded faults, they find that black-box techniques based on combinatorial interaction testing [10, 61] and test input diversity [24, 30, 31] perform comparably well to white-box approaches. While these black-box techniques are also applicable if there is no source code access, they require either a program's test inputs or a model thereof, which goes beyond CI and VCS metadata. Yoo and Harman [74, 75, 77] introduce the concept of Pareto efficient multi-objective regression test optimization to account for trade-offs between test criteria for different types of testing (e.g., structural and functional). Defect hypotheses and associated features in our methodology are drawn from their work (see Sec. 2.2), but we cannot directly apply their techniques due to the lack of required coverage information. Similarly, we cannot directly apply the unsafe RTS approach by Kim and Porter [36], who were among the first to create statistical ranking models for tests based on past-fault coverage, i.e., tests' history, and function coverage. Peng et al. [60] empirically study information retrieval (IR) techniques for RTP as first proposed by Saha et al. [67]. Their hybrid technique combines features from textual program changes and, similar to our work, test execution time and test failure history. It outperforms coverage-based RTP techniques and the baselines B_{cost} and $B_{history}$ (see Sec. 3.3.2) on a real-world dataset, and they argue for the "necessity [...] to better balance textual, cost, and historical information for more powerful test prioritization" [60]. We deem our approach with advanced feature engineering and predictive modeling to be one step in that direction, albeit excluding white-box textual analysis. Notably, analyzing textual code changes is possible, if CI and VCS metadata contain code *diffs* and tests' source code is accessible. However, going beyond the scope of our methodology, effective IR techniques further employ programming language-specific analysis (e.g., building an abstract syntax tree) or computationally expensive topic modeling [49, 60, 67].

Similarly, effectiveness for specific contexts [3, 16, 21, 38, 43, 68] and across projects [37, 44, 71, 81] has been studied for CI-RTP/S. However, none of the studies that investigate multiple projects uses VCS metadata; they exclusively rely on historical test execution information (i.e., CI logs). Using both kinds of data, as done in our study, indicates that features from CI logs alone are indeed powerful, but including VCS features can further increase cost-effectiveness. We are not aware of related work that uses *both* kinds of data and individually measures associated features' cost-effectiveness across projects. We consider the following papers to be most relevant for our work: Elbaum et al. [21] were the first to apply CI-RTP/S at industry scale: They used the simple heuristic B_{last} (see Sec. 3.3.2) to cost-effectively prioritize and select tests in Google's pre- and post-submit testing process. Spieker et al. [71] use RL for CI-RTP/S. On ABB's and Google's CI test history, their approach achieves competitive performance to simple RTP heuristics (e.g., B_{last} from Sec. 3.3.2). They formulate unsafe RTS as a time-constrained RTP problem, where the cut-off value is determined by the available test time. In contrast, we define cut-off values by empirical failure detection safety levels.

Finally, we have alluded to why sensitivity of cost-effectiveness to size, timeliness, and variety of data is important for CI-RTP/S. Prior work has investigated how much historical data can be beneficial [3, 8, 71]. Research conducted at Microsoft further showed fluctuating cost-effectiveness over time [33]. The impact of data variety has been studied by analyzing performance of predictive features [4, 8, 11, 45]. However, these studies either focus on specific industrial contexts or use more information than only CI and VCS metadata. In summary, we are not aware of prior work on CI-RTP/S approaches that studies how sensitive their cost-effectiveness is to associated parameters, performs *cost-aware* evaluation on real-world failures from industrial and open-source projects, and derives empirical guidelines for calibrating CI-RTP/S in practice.

5 CONCLUSION

Unsafe RTS and RTP techniques that exclusively rely on CI and VCS metadata (CI-RTP/S) are attractive alternatives to traditional, more intrusive techniques: They are inexpensive, language-agnostic, easy to transfer—no program or code access is necessary—, and have been shown to work well in different contexts. However, aspects of their design and evaluation are scattered across research, leaving practitioners to identify insights that apply to their context. Besides, adequately calibrating these techniques often requires high effort and experience with predictive modeling. Still, empirical calibration guidelines are not available. Instead of proposing new techniques, we consolidate existing RTP and unsafe RTS research into a methodology for building and evaluating CI-RTP/S approaches.

In our empirical study, we show that (1) limiting the training data to the most recent or even only faulty CI test runs often suffices, (2) features on test history work particularly well, and (3) naïve heuristics often outperform complex ML models from prior work. Practitioners can use these empirical guidelines to reduce the amount of effort for selecting and calibrating the best CI-RTP/S approaches for their project. Across studied projects, the approaches chosen by our methodology significantly outperform established RTP baselines. On average, practitioners can thereby expect to save 84% of the testing time while still detecting 90% of the failures when selecting tests. If CI and VCS metadata are available, the methodology is universally applicable, allowing practitioners to comfortably build and calibrate cost-effective RTP and RTS approaches.

ACKNOWLEDGMENTS

We thank Maria Graber, René Dammer, Markus Schnappinger, and the anonymous reviewers who provided helpful feedback to improve this paper. This work was partially funded by IVU Traffic Technologies and the German Federal Ministry of Education and Research (BMBF), grant "SOFIE, 01IS18012A".

REFERENCES

[1] Khaled Walid Al-Sabbagh, Miroslaw Staron, Regina Hebig, and Wilhelm Meding. 2019. Predicting Test Case Verdicts Using Textual Analysis of Committed Code Churns. In *Joint Proceedings of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*, Vol. 2476. 138–153.

[2] Khaled Walid Al-Sabbagh, Miroslaw Staron, Miroslaw Ochodek, Regina Hebig, and Wilhelm Meding. 2020. Selective Regression Testing based on Big Data: Comparing Feature Extraction Techniques. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*. 322–329. https://doi.org/10.1109/ICSTW50294.2020.00058

[3] Jeff Anderson, Saeed Salem, and Hyunsook Do. 2014. Improving the effectiveness of test suite through mining historical data. In *Processings of the Working Conference on Mining Software Repositories*. 142–151. https://doi.org/10.1145/2597073.2597084

[4] Jeff Anderson, Saeed Salem, and Hyunsook Do. 2015. Striving for Failure: An Industrial Case Study about Test Failure Prediction. In *Proceedings of the International Conference on Software Engineering*. 49–58. https://doi.org/10.1109/ICSE.2015.134

[5] Maral Azizi and Hyunsook Do. 2018. ReTEST: A Cost Effective Test Case Selection Technique for Modern Software Development. In *Proceedings of the International Symposium on Software Reliability Engineering*. 144–154. https://doi.org/10.1109/issre.2018.00025

[6] 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 International Conference on Mining Software Repositories*. 447–450. https://doi.org/10.1109/msr.2017.24

[7] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, 1 (2012), 281–305.

[8] Antonia Bertolino, Antonio Guerriero, Roberto Pietrantuono, Stefano Russo, Breno Miranda, and Roberto Pietran-Tuono. 2020. Learning-to-Rank vs Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration. In *Proceedings of the International Conference on Software Engineering*. 1–12. https://doi.org/10.1145/3377811.3380369

[9] Leo Breiman and Philip Spector. 1992. Submodel Selection and Evaluation in Regression. The X-Random Case. *International Statistical Review / Revue Internationale de Statistique* 60, 3 (1992), 291–319. https://doi.org/10.2307/1403680

[10] Renée C. Bryce and Charles J. Colbourn. 2006. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology* 48, 10 (2006), 960–970. https://doi.org/10.1016/j.infsof.2006.03.004

[11] Benjamin Busjaeger and Tao Xie. 2016. Learning for test prioritization: An industrial case study. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 975–980. https://doi.org/10.1145/2950290.2983954

[12] Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression test selection across JVM boundaries. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 809–820. https://doi.org/10.1145/3106237.3106297

[13] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing test prioritization via test distribution analysis. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 656–667. https://doi.org/10.1145/3236024.3236053

[14] Janez Demšar. 2006. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research* 7 (2006), 1–30.

[15] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10, 4 (2005), 405–435. https://doi.org/10.1007/s10664-005-3861-2

[16] Edward Dunn Ekelund and Emelie Engstrom. 2015. Efficient regression testing based on test history: An industrial evaluation. In *Proceedings of the International Conference on Software Maintenance and Evolution*. 449–457. https://doi.org/10.1109/icsm.2015.7332496

[17] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. 2001. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the International Conference on Software Engineering*. 329–338. https://doi.org/10.1109/icse.2001.919106

[18] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2000. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis*. 101–112. https://doi.org/10.1145/347324.348910

[19] 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

[20] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. 2004. Selecting a cost-effective test case prioritization technique. *Software Quality Journal* 12, 3 (2004), 185–210. https://doi.org/10.1023/b:sqjo.0000034708.84524.22

[21] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 235–245. https://doi.org/10.1145/2635868.2635910

[22] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Supplemental Material for: "Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration". https://doi.org/10.6084/m9.figshare.13656443

[23] Michael G. Epitropakis, Shin Yoo, Mark Harman, and Edmund K. Burke. 2015. Empirical evaluation of Pareto efficient multi-objective regression test case prioritisation. In *Proceedings of the International Symposium on Software Testing and Analysis*. 234–245. https://doi.org/10.1145/2771783.2771788

[24] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. 2016. Test Set Diameter: Quantifying the Diversity of Sets of Test Cases. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. 223–233. https://doi.org/10.1109/ICST.2016.33

[25] Kurt F. Fischer. 1977. A test case selection method for the validation of software maintenance modifications. In *Proceedings of International Computer Software and Applications Conference*. 421–426.

[26] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In *Proceedings of the International Conference on Software Engineering*. 713–716. https://doi.org/10.1109/icse.2015.230

[27] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the International Symposium on Software Testing and Analysis*. 211–222. https://doi.org/10.1145/2771783.2771784

[28] Mark Harman. 2011. Making the case for MORTO: Multi objective regression test optimization. In *Proceedings of the International Conference on Software Testing, Verification, and Validation Workshops*. 111–114.

[29] 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 International Conference on Software Engineering*. 523–534. https://doi.org/10.1145/2884781.2884791

[30] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering* 40, 7 (2014), 650–670. https://doi.org/10.1109/TSE.2014.2327020

[31] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 2013. Assessing software product line testing via model-based mutation: An application to similarity testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*. 188–197. https://doi.org/10.1109/ICSTW.2013.30

[32] Steffen Herbold. 2020. Autorank: A Python package for automated ranking of classifiers. *Journal of Open Source Software* 5, 48 (2020), 2173–2173. https://doi.org/10.21105/joss.02173

[33] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The art of testing less without sacrificing quality. In *Proceedings of the International Conference on Software Engineering*. 483–493. https://doi.org/10.1109/icse.2015.66

[34] Xianhao Jin and Francisco Servant. 2020. A cost-efficient approach to building in continuous integration. In *Proceedings of the International Conference on Software Engineering*. 13–25. https://doi.org/10.1145/3377811.3380437

[35] 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 *Proceedings of the International Conference on Neural Information Processing Systems*. 3149–3157.

[36] Jung Min Kim and Adam Porter. 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the International Conference on Software Engineering*. 119–129. https://doi.org/10.1145/581339.581357

[37] Eric Knauss, Miroslaw Staron, Wilhelm Meding, Ola Soder, Agneta Nilsson, and Magnus Castell. 2015. Supporting Continuous Integration by Code-Churn Based Test Selection. In *Proceedings of the International Workshop on Rapid Continuous Software Engineering*. 19–25. https://doi.org/10.1109/rcose.2015.11

[38] Jung Hyun Kwon and In Young Ko. 2018. Cost-Effective Regression Testing Using Bloom Filters in Continuous Integration Development Environments. In *Proceedings of the Asia-Pacific Software Engineering Conference*. 160–168. https://doi.org/10.1109/apsec.2017.22

[39] Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D. Ernst, and Tao Xie. 2020. Dependent-Test-Aware Regression Testing Techniques. In *Proceedings of the International Symposium on Software Testing and Analysis*. 298–311. https://doi.org/10.1145/3395363.3397364

[40] Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. 2012. Prioritizing test cases with string distances. In *Automated Software Engineering*, Vol. 19. 65–95. https://doi.org/10.1007/s10515-011-0093-0

[41] 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 International Symposium on Foundations of Software Engineering*. 583–594. https://doi.org/10.1145/2950290.2950361

[42] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic regression test selection. In *Proceedings of the International Conference on Automated Software Engineering*. 949–954. https://doi.org/10.1109/ase.2017.8115710

[43] 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 International Conference on Software Engineering: Software Engineering in Practice*. 101–110. https://doi.org/10.1109/icse-seip.2019.00019

[44] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining prioritization: Continuous prioritization for continuous integration. In *Proceedings of the International Conference on Software Engineering*. 688–698. https://doi.org/10.1145/3180155.3180213

[45] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive Test Selection. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 91–100. https://doi.org/10.1109/ICSE-SEIP.2019.00018

[46] Dusica Marijan, Arnaud Gotlieb, and Abhijeet Sapkota. 2020. Neural Network Classification for Improving Continuous Regression Testing. In *Proceedings of the International Conference On Artificial Intelligence Testing*. 123–124. https://doi.org/10.1109/AITEST49225.2020.00025

[47] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test case prioritization for continuous regression testing: An industrial case study. In *Proceedings of the International Conference on Software Maintenance*. 540–543. https://doi.org/10.1109/icsm.2013.91

[48] Dusica Marijan and Marius Liaaen. 2018. Practical selective regression testing with effective redundancy in interleaved tests. In *Proceedings of the International Conference on Software Engineering*. 153–162. https://doi.org/10.1145/3183519.3183532

[49] Toni Mattis and Robert Hirschfeld. 2020. Lightweight Lexical Test Prioritization for Immediate Feedback. *The Art, Science, and Engineering of Programming* 4, 3 (2020), 12:1–12:32. https://doi.org/10.22152/programming-journal.org/2020/4/12

[50] Toni Mattis, Patrick Rein, Falco Dürsch, and Robert Hirschfeld. 2020. RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization. In *Proceedings of the Conference on Mining Software Repositories*. 385–396. https://doi.org/10.1145/3379597.3387458

[51] Toni Mattis, Patrick Rein, Falco Dürsch, and Robert Hirschfeld. 2020. RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization. https://doi.org/10.5281/zenodo.3610998

[52] 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 International Conference on Software Engineering: Software Engineering in Practice*. 233–242. https://doi.org/10.1109/icse-seip.2017.16

[53] Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. 2018. FAST approaches to scalable similarity-based test case prioritization. In *Proceedings of the International Conference on Software Engineering*. 222–232. https://doi.org/10.1145/3180155.3180210

[54] Armin Najafi, Weiyi Shang, and Peter C. Rigby. 2019. Improving Test Effectiveness Using Test Executions History: An Industrial Experience Report. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 213–222. https://doi.org/10.1109/icse-seip.2019.00031

[55] Agastya Nanda, Senthil Mani, Saurabh Sinha, Mary Jean Harrold, and Alessandro Orso. 2011. Regression testing in the presence of non-code changes. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*. 21–30. https://doi.org/10.1109/icst.2011.60

[56] Tanzeem Bin Noor and Hadi Hemmati. 2016. A similarity-based approach for test case prioritization using historical failure data. In *Proceedings of the International Symposium on Software Reliability Engineering*. 58–68. https://doi.org/10.1109/issre.2015.7381799

[57] Tanzeem Bin Noor and Hadi Hemmati. 2017. Studying test case failure prediction for test case prioritization. In *Proceedings of the International Conference on Predictive Models and Data Analytics in Software Engineering*. 2–11. https://doi.org/10.1145/3127005.3127006

[58] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 241–251. https://doi.org/10.1145/1029894.1029928

[59] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[60] Qianyang Peng, August Shi, and Lingming Zhang. 2020. Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis*. 324–336. https://doi.org/10.1145/3395363.3397383

[61] Justyna Petke, Shin Yoo, Myra B. Cohen, and Mark Harman. 2013. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 26–36. https://doi.org/10.1145/2491411.2491436

[62] Adithya Abraham Philip, Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Nachiappan Nagppan. 2019. FastLane: Test Minimization for Rapidly Deployed Large-Scale Online Services. In *Proceedings of the International Conference on Software Engineering*. 408–418. https://doi.org/10.1109/icse.2019.00054

[63] John Platt. 1999. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers* 10, 3 (1999), 61–74.

[64] Gregg Rothermel and Mary Jean Harrold. 1994. A Framework for Evaluating Regression Test Selection Techniques. In *Proceedings of the International Conference on Software Engineering*. 201–210. https://doi.org/10.1109/ICSE.1994.296779

[65] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 6, 2 (1997), 173–210. https://doi.org/10.1145/248233.248262

[66] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. 2000. Regression test selection for C++ software. *Software Testing, Verification and Reliability* 10, 2 (2000), 77–109.

[67] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. 2015. An information retrieval approach for regression test prioritization based on program changes. In *Proceedings of the International Conference on Software Engineering*. 268–279. https://doi.org/10.1109/icse.2015.47

[68] Mark Sherriff, Mike Lake, and Laurie Williams. 2007. Prioritization of regression tests using singular value decomposition with empirical change records. In *Proceedings of the International Symposium on Software Reliability Engineering*. 81–90. https://doi.org/10.1109/issre.2007.25

[69] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating Test-Suite Reduction in Real Software Evolution. In *Proceedings of the International Symposium on Software Testing and Analysis*. 84–94. https://doi.org/10.1145/3213846.3213875

[70] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *Proceedings of the International Symposium on Software Reliability Engineering*. 228–238. https://doi.org/10.1109/issre.2019.00031

[71] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the International Symposium on Software Testing and Analysis*. 12–22. https://doi.org/10.1145/3092703.3092709

[72] Robert White, Jens Krinke, and Raymond Tan. 2020. Establishing Multilevel Test-to-Code Traceability Links. In *Proceedings of the International Conference on Software Engineering*. 861–872. https://doi.org/10.1145/3377811.3380921

[73] Rüdiger Wirth and Jochen Hipp. 2000. CRISP-DM : Towards a Standard Process Model for Data Mining. In *Proceedings of the International Conference on the Practical Application of Knowledge Discovery and Data Mining*. 29–39.

[74] Shin Yoo and Mark Harman. 2007. Pareto efficient multi-objective test case selection. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM Press, 140–150. https://doi.org/10.1145/1273463.1273483

[75] Shin Yoo and Mark Harman. 2010. Using hybrid algorithm for Pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software* 83, 4 (2010), 689–701. https://doi.org/10.1016/j.jss.2009.11.706

[76] 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/stv.430

[77] Shin Yoo, Robert Nilsson, and Mark Harman. 2011. Faster Fault Finding at Google Using Multi Objective Regression Test Optimisation. In *Proceedings of the International Symposium on the Foundations of Software Engineering*.

[78] Tingting Yu and Ting Wang. 2018. A Study of Regression Test Selection in Continuous Integration Environments. In *Proceedings of the International Symposium on Software Reliability Engineering*. 135–143. https://doi.org/10.1109/ISSRE.2018.00024

[79] Lingming Zhang. 2018. Hybrid regression test selection. In *Proceedings of the International Conference on Software Engineering*. 199–209. https://doi.org/10.1145/3180155.3180198

[80] Hua Zhong, Lingming Zhang, and Sarfraz Khurshid. 2019. TestSage: Regression test selection for large-scale web service testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. 430–440. https://doi.org/10.1109/icst.2019.00052

[81] Yuecai Zhu, Emad Shihab, and Peter C. Rigby. 2018. Test re-prioritization in continuous testing environments. In *Proceedings of the International Conference on Software Maintenance and Evolution*. 69–79. https://doi.org/10.1109/icsme.2018.00016

[82] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 91–100. https://doi.org/10.1145/1595696.1595713