Predictive Test Optimization Without Historical Failure Data

Maximilian Jungwirth*¶, Raphael Nömmer§†, Andreas Stahlbauer†, Sven Apel§, and Gordon Fraser¶
*BMW Group, Munich, Germany
†CQSE GmbH, Passau, Germany
§Saarland University, Saarbrücken, Germany
¶University of Passau, Passau, Germany
maximilian.jungwirth@bmw.de, noemmer@cqse.eu, stahlbauer@cqse.eu, apel@cs.uni-saarland.de, gordon.fraser@uni-passau.de

***Abstract*—As software systems grow in size and complexity, their test suites become increasingly expensive to execute, delaying feedback and wasting resources. State-of-the-art predictive test optimization methods use machine learning to select tests relevant to a change, typically trained on historical failures. This depends on failure histories and is susceptible to label noise from flaky tests. We introduce the novel method of co-evolution labeling for predictive test optimization, deriving test relevance from tests and code changing together in the version history. These data are widely available and less noisy. We train and evaluate models with failure and co-evolution labels on three large projects—from industry and open source; each with thousands of tests and more than 1M LOC. Co-evolution labeling nearly matches the failure detection capabilities of failure-based models, with an overall *Average Percentage of Faults Detected* of 0.88, while being more resistant to label noise and requiring no test result history. Co-evolution labeling offers a practical, immediately deployable path to faster pipelines and reliable predictive test optimization.**

***Index Terms*—Test Optimization; Test Selection; Test Prioritization; Machine Learning; Predictive; Regression Testing**

I. Introduction

As software systems grow in size and complexity, so do their test suites [1], [2]. Executing these large test suites is time-consuming and delays developer feedback, diminishing the benefits of testing [3], [4]. Researchers and practitioners have devised techniques such as *test selection* [5] and *test prioritization* [6] that utilize per-test coverage data, control flow data, and call graphs to effectively reduce test suite execution time or the time to first failure [4].

While these data provide an excellent source for test optimization [7], [8], they can be difficult to collect and maintain in industrial settings due to large-scale systems, distributed architectures spanning multiple languages and communication protocols, or diverse testing technologies. Predictive test optimization techniques have therefore gained popularity [1], [9], [10], [11], [12]. These techniques leverage machine learning on historical test failure data to predict future failures, delivering state-of-the-art performance while using information that is easier to collect [13], [14], [1], [9].

However, relying on test failure data for model training has two major drawbacks. **Cold Start:** Historical test results are often unavailable or span too short a period, requiring a costly data-collection phase. **Label Noise:** Flaky tests corrupt the training signal and degrade prediction performance [1], [15].

To address both limitations, we propose *test and code co-evolution* as an alternative labeling source: when developers modify code, they frequently edit or add related tests in the same commit [16], [17], [18], [19], creating implicit links between code and tests in the version history. Co-evolution information is universally available in version control systems, eliminating the cold-start. Because it is derived statically from commit history, flaky tests do not impact the training signal.

We hypothesize that analyzing test and code co-evolution reveals which tests are relevant to a given code change, providing a solid base for predictive test optimization. This contrasts with existing literature, which utilizes failing tests as the sole proxy for test relevance [1], [11], [12]. We explore co-evolution as a label source for predicting test relevance. Our evaluation on three large industry and open-source projects shows that co-evolution labels are effective for predictive test optimization, with an *APFD* of 0.88, while being more robust against label noise. Our key contributions are:

Approach: A novel strategy for predictive test optimization that leverages test and code co-evolution to determine test relevance, eliminating the need for historical test results and enhancing robustness against test flakiness.

Evaluation: An empirical evaluation of the proposed approach on three large projects (>1M LOC each, two from industry), on its performance in test optimization, robustness against label noise (test flakiness and accidental co-evolution), and versatility on different types of tests.

Guidance: A practitioner's guide for leveraging co-evolution labels for predictive test optimization to overcome existing limitations, for example, limited and noisy data.

Data: A replication package containing the code and data used in our evaluation, including co-evolution and failure data from one open-source and one industry project, available at: https://github.com/test-optimization-research/Predictive-Test-Optimization-Without-Historic-Failure-Data

Maximilian Jungwirth and Raphael Nömmer contributed equally.

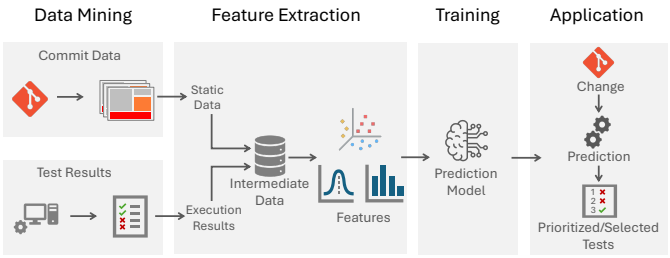

Fig. 1. Overview of the test optimization approach.

II. Predictive Test Optimization

Test optimization is an umbrella term that encompasses, among others, test selection and test prioritization, both of which aim to reduce testing effort while maintaining fault detection capability [20]. Test prioritization aims to order tests so that likely failing tests run earlier, while test selection chooses a subset of tests relevant to recent changes. As our approach can be used for either one, we refer to it as a test optimization approach. Traditional test optimization relies on static information like build graphs and class dependencies, or dynamic information like code coverage, but these techniques either lack sufficient accuracy or become infeasible in large systems with multiple languages, distributed components, and diverse communication protocols [1], [12]. Therefore, predictive test optimization techniques that leverage machine learning have gained popularity [1], [9], [10], [11], [12]. Their core idea is to predict a "relevance score" for each test given a change set, enabling practitioners to prioritize tests (faster time to first failure) or select a subset (reduced test runtime). The model learns this score by establishing a relationship between code changes and relevant (potentially failing) tests during training.

A. Predictive Test Optimization Workflow

Figure 1 illustrates the four phases of our workflow:

1) Data Mining: We continuously collect version control and Continuous Integration (CI) data. For failure-based training, flaky test results have to be filtered to reduce label noise. For co-evolution-based training, high-quality labels result from guidelines and practices that promote writing and updating tests alongside code changes.

2) Feature Extraction: We parse code, extract keywords, and map execution results to test implementations for feature computation. Labels ("failed", "passed", or "co-evolved") are stored alongside feature vectors.

3) Batch-wise Training: We train models on feature vectors within a window of commits to limit computational costs and retrain on demand or after a set number of commits.

4) On-Request Test Optimization: For each incoming change, the deployed model scores every existing test, enabling practitioners to prioritize execution order or select a relevant subset.

B. Test Failures (TeFa) as Labels

Current predictive test optimization approaches primarily learn from historical test failures, predicting tests most likely to fail for a given change to reveal regression faults [1], [9], [10], [11], [12]. We abbreviate this label type and its models as *TeFa*. These labels rely on dynamic CI execution data, categorizing tests as "failed" or "not-failed" (passed/skipped).

Gathering sufficient test failures to train a model that reliably predicts future failures is a substantial challenge for predictive test optimization. In many projects, historical test failures are simply unavailable: test results might not be archived because they are not relevant in everyday workflows. Others store test results but retain them for only one to two weeks, which is insufficient to train a robust model. Additional data sources, such as local test runs, are typically inaccessible.

While it is possible to begin archiving test results when introducing a failure-based approach, it takes substantial time to collect sufficient data to produce results. In industrial practice, this "cold start" period makes it challenging to convince stakeholders to adopt test optimization.

Beyond the difficulties of data acquisition, the quality of historical test failure data is often suboptimal. A primary issue is flaky tests; tests that do not reliably produce the same results due to non-determinism in the source or test code, unreachable external services, or other factors, for example, tests that only fail in February of a leap year [21]. Flakiness is a frequently occurring problem, for example, up to 84% of pass-to-fail transitions at Google stem from flaky tests [22], Chrome reports 99% [15], and in our own analysis of the open source reference management software JabRef,[1] over 90% of 1065 test failures across 50 months turned out to be flaky. Identifying and filtering flaky results before model training requires significant upfront effort: rerunning them can cost tens of thousands of dollars [23], and manual triaging takes minutes per failure [24], yet failing to do so degrades prediction performance [1].

These factors necessitate substantial upfront and continuous effort to maintain predictive test optimization using failure-based labels, motivating the search for a different label source.

C. Test Code Co-Evolution (CoEvo) as Labels

To overcome these challenges, we propose *test and code co-evolution*, abbreviated as *CoEvo*, as an alternative label source for training predictive test optimization models. A co-evolution label is positive for a test if and only if it was changed in a commit alongside (therefore related) production code. This means that if only a test or only production code was changed in a commit, the co-evolution label created for any test is negative. We omit such commits as they provide no valuable data for training. To identify test and production code, we rely on heuristics based on file name pattern matching, for example, files containing `test`, or residing in designated test directories. This practical approach scales to repositories with millions of lines of code without requiring language-specific tooling.

Co-evolution data is derived directly from version control history, bypassing dynamic execution requirements and the cold start problem. This is a prevalent practice: our analysis of our study objects (*BMW*, *Teamscale*, *Gitlab*) and major open-source projects like Spring Boot and TensorFlow revealed high

[1] https://jabref.org

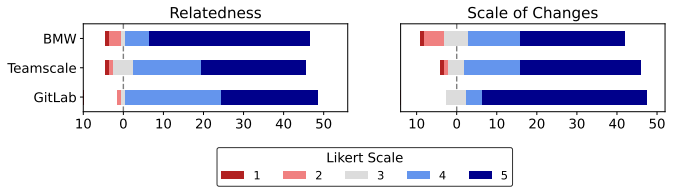

Fig. 2. We categorized the changes of 50 commits per project regarding relatedness of changed code and tests in the left chart, ranging from 1 "very unrelated" to 5 "closely related" as well as scale of changes in the right chart from 1 "very many topics addressed" to 5 "only a single topic addressed" in a commit.

co-evolution rates between 25% and 75%. Given their mature version histories, these projects provide thousands of commits to train models on co-evolution even in the worst case (25%).

Typical software development organizations are driven by bug, feature, and maintenance tickets. Each ticket is addressed in isolation and corresponding commits, merges, and merge requests are created. In these units, both tests and code are introduced and modified. Our hypothesis is that co-evolution of tests and code frequently implies a semantic connection between them. By looking at sufficient samples of changed tests and code we thus expect that a model can learn a coupling between tests and related code.

A resulting classifier based on test and code co-evolution labels thus predicts whether a change to the system under test is related to a test. If related code is changed in the future, it is more likely that a coupled test will fail than if this code has never been changed alongside a test. Software development methodologies such as Test-Driven Development (TDD) and Behavior-Driven Development (BDD) further encourage the creation and modification of tests alongside the implementation of new features or changes in existing code [18], [19], thus providing a good foundation for co-evolution labels. Since co-evolution data are purely static information, flaky test executions have no impact. However, there may be coincidental co-evolution, for example, when multiple unrelated parts of the system are changed at once. Global refactorings, dependency updates, and other non-functional changes produce coincidental co-evolution too, but we expect the rate of this to be lower than flakiness, due to established development practices.

To support this claim we randomly sampled 50 commits with co-evolution for each study object (cf. Section III-A) and manually analyzed them. We rated the commits based on two factors that constitute accidental co-evolution: Scale and Relatedness. Scale describes how many files with different topics were changed in a single commit. If a commit changes many files addressing different parts of the system, there are likely many changes that are irrelevant for a test, causing noise in the data. Relatedness describes how closely related tests and code were in a commit. If the code changes are not relevant for the modified tests, the model will not be able to learn meaningful patterns. Each factor was rated on a Likert scale from 1 to 5 [25]—details on the scale in Fig. 2. For *BMW* and *Teamscale* the rating was done by experts involved in the projects. The results of our manual investigation are shown in Fig. 2. With an overall 62% of commits being rated with a score of 5 for both factors and 88% being 4 or above, the level of accidental co-evolution in our sample set is significantly lower (5% with score 2 or below) than the numbers reported for test flakiness in large projects [15], [26].

D. Features for Predictive Test Optimization

Feature selection for a machine learning technique is vital, as it provides the basis for a model's ability to learn its prediction task. This section discusses the features we used for machine learning as well as the rationale for including them. We based our selection on the original work on predictive test selection [1]. Compared to a newer study on predictive test selection [9], we use a smaller feature set for two reasons: (1) work council rules and data privacy regulations restrict developer-metric features, and (2) coverage-based features add substantial setup complexity. We decided against using static analysis features that are language-dependent, as our approach should be largely language agnostic and our study objects use different languages and technologies. Furthermore, their higher-level tests cross language borders via, for example, REST interfaces, which are difficult to track for dependency analysis. We instead add several change-based features, the details of which we explain in the following sections, to better facilitate co-evolution-based training. We differentiate between two types of features: history features, which consist of data from a test's history, and change features, which consider the changes to a test and code in a specific commit.

1) Test History Features: This group considers aspects such as the frequency of file changes or test failures in a specific time period. We use the following test-history-based features:

Change History (*TeFa* & *CoEvo*): This family of features represents the number of times a code file (test or production) was modified in different time frames. We consider several intervals based on prior work [1] and on the development practices at our industry partners (sprints span between 2–4 weeks): 7, 14, 28 and 56 days, as well as the total number of times a file was changed within the part of the project history used for training. For functionality that is under active development, spanning multiple commits or pipeline runs, it is more likely that previous test failures will recur.
Historical Failure Rates (only *TeFa*): This metric is also based on the time intervals introduced in prior work [1]: the last 7, 14, 28 and 56 days, as well as the overall number of failures. It represents the frequency of failures of a test within these periods. Failure frequency has been shown to be an effective heuristic by itself for test selection [13]. However, it cannot identify tests that newly fail.

2) Change Features: This group of features contains values derived from the changes made in the current commit, that is, the commit for which the model attempts to predict failing or co-evolving tests. Change-based features consist of two groups: numerical features and textual features.

a) Numerical Change Features: Numerical refers to features which are represented as single numerical values without advanced text processing:

File Cardinality (*TeFa* & *CoEvo*): The number of modified files in a change set in a commit/merge-request. Commit sizes might affect the expected number of failing tests.

Number of Common Path Tokens (*TeFa* & *CoEvo*): The number of shared tokens (split on the path separator) between the file paths in a commit and a test, relative to the root of the repository. This feature represents a lexical distance between code and test [1]. The rationale is that files and tests in similar directories are more likely to be related; it reflects that the organization and naming of tests is typically similar to that of the system's implementation. This number is accumulated for all changed code files.

Number of Changed Lines (*TeFa* & *CoEvo*): The total number of added lines and removed lines in a given commit, treated as two separate features, to provide the model with a notion of commit size.

b) Textual Change Features: These features concern either the similarity between tests or clustering of tests and file contents to categorize similar tests or commits. Textual features are derived from text processing techniques including clustering and information retrieval. We apply these techniques to map strings and string similarities to different categories or scores that we can use for training our prediction model.

***Clustering*:** To map textual fragments like file names, code snippets, and identifiers to categories suitable for training, we first create term frequency-inverse document frequency (TF-IDF) [27] vectors from the test files. As the cosine angle between two of these vectors can be used as a similarity measure between two documents, we can then group them using HDBScan clustering [28]. The resulting clustering produces the same identifier for similar documents. We select TF-IDF over the *Bag of Words* approach used in previous work [12] because TF-IDF corrects for frequent terms appearing in many documents; due to code syntax requirements, we expect many such terms. Our reason for choosing HDBScan, which determines the number of clusters autonomously over a simpler option like K-means, which is used in related work [29], [30], is that a reasonable number of clusters is not decidable across different projects. We use the cluster identifiers to identify similar documents.

***Okapi BM25 (BM25) Scores*:** BM25 is a ranking function that estimates the relevance of a set of documents for a search query [31]. It is based on TF-IDF scores but adds term frequency saturation to reduce the gain that long documents—or in our case, long test files—receive from containing many terms. We use BM25 scores with the changed source code as the query and test code as the documents. Information retrieval approaches have been shown to be effective for prioritizing tests by themselves [32], which is why we include this information in our feature set. To correct for differences in query lengths, we additionally include a normalized version of the scores. We calculate the normalized version by dividing the score by the sum of the IDF weights of the terms in the query. This corrects for the inherently higher scores of longer queries while considering the difference in the weights of terms.

We use the techniques on our group of textual change features:

Similarity between Test File Names and Changed File Names (*TeFa* & *CoEvo*): For this feature group, we compute BM25 similarity scores between the test file name and the changed file names in a commit, explicitly capturing whether a test file name is similar to modified files.

Similarity between Changed Code and Test Code (*TeFa* & *CoEvo*): For this set of features, we compute a BM25 similarity score, prepared on the last state of the training set. We split the similarity into two separate features, added and removed similarity calculating the similarity of a test with the added code and removed code of a commit. We hypothesize that related tests share many common tokens with the code they test, therefore, we capture this feature.

Changed File Name Cluster (*TeFa* & *CoEvo*): We compute this feature by combining the file name tokens of the modified files and returning the identifier of their cluster.

Test Name Cluster (*TeFa* & *CoEvo*): The cluster identifier of the test name captures patterns of similar test names.

Changed Test File Names Cluster (only *TeFa*): Names of modified tests are extracted, combined, and a cluster identifier is computed in order to identify related tests for *TeFa*. To avoid leakage in the prediction task, this feature is not used for models based on *CoEvo*.

III. Evaluation

To evaluate the performance of predictive test optimization based on test and code co-evolution, we compare it to test-failure labeling and several non-learning baselines. Our main goal is to understand how the labeling strategy influences the prediction performance for changes and tests found in large real-world software projects. For the evaluation, we refer to a test as a test file or test target, as it is common in industry to perform optimization at this granularity level [12], [11].

We evaluate our approach in the form of a field experiment [33] and address the following research questions:

RQ1 Performance: How effective and efficient is predictive test optimization based on co-evolution?

RQ2 Robustness: How robust is predictive test optimization based on co-evolution against label noise?

RQ3 Versatility: How is predictive test optimization based on co-evolution influenced by a different type of tests?

A. Study Objects

We run our experiments on two closed source industry projects, *BMW* and *Teamscale*. We selected these projects based on our domain knowledge and access to their development and testing processes. To further validate our approach and improve verifiability and generalizability we added *Gitlab* [34], as an open source project.[2] All three projects have test suites of

[2]https://gitlab.com/gitlab-org/gitlab

TABLE I
CHARACTERISTICS OF THE STUDY OBJECTS.

	SLoC	Tests	Pipeline History	Pipeline Failures	Test Failures	Commit History	Commits
BMW	∼7.8 M	∼6,000	90 d	∼450	613	90 d	∼34 k
Teamscale	∼1.7 M	∼2,500	500 d	∼4,000	∼2.5 k	500 d	∼37 k
Gitlab [34]	∼4.9 M	∼17,000	60 d	∼10,000	∼5 k	60 d	∼80 k

substantial size that take several hours when run in sequence, which makes test optimization a worthwhile consideration for all of them. We had access to validated test failures, confirmed by delayed validation runs [34]. This ensures the reliability of our failure data and strengthens the validity of our evaluation. We had access to the complete version history for all study objects, but restricted it to the same time span for each project in which failure data were available to ensure a fair comparison. Table I summarizes the central characteristics of the study objects and the following paragraphs give more detail:

BMW (Autonomous Driving Software): BMW is a closed source autonomous driving project [35] with approximately 7.8 million lines of code, primarily written in C++ and Python. We evaluate a pre-merge pipeline job with 6,091 tests over 90 days, covering ∼34,000 commits, 457 failed pipeline runs, and 613 validated test failures [36]. For RQ1 and RQ2, we omit acceptance tests; RQ3 includes them, yielding 1,425 failures for training and evaluation.

Teamscale (Software Quality Platform): Teamscale is a closed source software quality platform developed in Java and TypeScript. It has approximately 1.7 million lines of code and 2,495 tests. Over the longer evaluation of 500 days, it produced ∼37,000 commits and ∼4,000 pipeline runs yielded ∼2,700 validated test failures.

Gitlab (Open Source DevOps Platform): Gitlab is an open source DevOps platform, primarily developed in Ruby and JavaScript, with ∼4.9 million lines of code and the largest test suite (17,435 tests). Over 60 days, ∼80,000 commits and ∼10,000 pipeline runs yielded ∼5,400 validated test failures. We use a dataset created in the context of test flakiness research [34] for *Gitlab*.

B. Experimental Setup

Implementation: We use Extreme Gradient Boosted Decision Trees (XGBoost) to predict the tests to run for a given change set, due to their low-cost inference, high performance with coarse-grained features and robustness to overfitting [37], [11]. Furthermore, these models have been used in prior work and in production on predictive test optimization [1], [38], [11]. For training, we use an 80%/10%/10% train/val/test split on the available pipeline runs in temporal order. As non-co-evolving (passing) tests outweigh co-evolving (failing) tests, we randomly undersample the majority class (non-co-evolving/passing tests) to balance the training data. By evaluating ratios between 10:1 and 1:4 as well as no sampling, we determined that a 1:1 ratio of non-co-evolving/passing to co-evolving/failing tests yields the best results among the investigated ratios. We further use Bayesian optimization per project and labeling strategy to tune XGBoost hyperparameters: max_depth, learning_rate, subsample, colsample_bytree, min_child_weight, gamma, reg_alpha, and reg_lambda. The search space for each hyperparameter was defined based on the XGBoost documentation [37]—see replication package. We run 35 iterations of optimizations on a time-series split with 10 folds, which simulates production deployment (predict future outcomes based on the past). Note that using failure-based labels as proxy for test relevance can favor *TeFa*; we retain this choice to mirror independent labeling (from failure records) for *CoEvo*. Each model outputs a probability per test (failure or co-evolution likelihood), which we use to rank tests and select within the given budget.

Baselines: We compare *CoEvo* to six baselines: *TeFa* (failure-labeled), *TeFa (wo FR)* (*TeFa* without historical failure-rate features) to isolate the effect of repeated test failures, *IR* (information retrieval using non-normalized BM25 similarity on added/removed code tokens; see Section II-D) as picking tests that are textually similar to changed code has been shown to be an effective test optimization tool by itself [32], *ReFa* (recently failed tests) as simply repeating recently failed tests can provide good results [13], *CoEvolved* (select only historically co-evolved tests) which tests whether learning from co-evolution outperforms using co-evolution links directly, and *Random* which represents a no-knowledge baseline.

Evaluation Setup: We evaluate effectiveness using per-project shared test sets across all labeling strategies and baselines to ensure comparability. We define relevance as failing tests [1] (additionally verified by delayed reruns). For each selection budget (fraction of the suite), we report the percentage of relevant (failing) tests selected. We also report *APFD*, which summarizes ranking and selection in a single metric [6], [39]. Since *APFD* requires a failure–fault mapping, we adopt a one-to-one mapping with equal severity per failing test, as done in previous research [6], [12], [1]. As we filter flaky tests in the test sets, flakiness has no impact on this mapping.

Since we compare $k = 7$ approaches on paired samples (each pipeline run yields one *APFD* value per approach), we first apply the Friedman test as a non-parametric omnibus test for repeated measures to detect whether at least one approach differs significantly [40]. If the Friedman test rejects the null hypothesis, we conduct post-hoc pairwise Wilcoxon signed-rank tests—appropriate for paired, non-normally distributed data—at $\alpha = 0.05$ and apply Bonferroni correction over all $\binom{7}{2} = 21$ comparisons per project to control the error rate. We quantify effect sizes with Vargha-Delaney's $\hat{A}_{12}$ [41].

C. Threats to Validity

Internal Validity: Our evaluation depends on label accuracy for training and testing. All models—*CoEvo*, *TeFa*, *TeFa (wo FR)*—are susceptible to label noise; enforced reviews and ticket-based workflows at our partners reduce expected noise for *CoEvo*. For *Teamscale* and *Gitlab*, we used validated datasets [34] with extra reruns to filter inconsistent failures; we applied the same validation strategy for *BMW*.

TABLE II

PRECISION, RECALL, F1-SCORE OF ALL MODEL TYPES (*CoEvo*, *TeFa*, *TeFa (wo FR)*) ON THEIR CORRESPONDING LABEL TYPE FOR EACH PROJECT.

	BMW			*Teamscale*			*Gitlab*		
Model	P	R	F	P	R	F	P	R	F
CoEvo	97	93	95	99	80	88	96	87	91
TeFa	91	93	92	96	43	60	99	68	73
TeFa (wo FR)	92	90	80	83	66	73	90	62	73

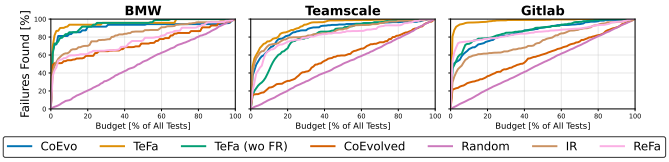

Fig. 3. Percentage of failed tests selected per budget for each study object.

External Validity: We analyzed *CoEvo* on three major projects spanning open source and enterprise contexts, multiple languages (C++, Java, Ruby, JavaScript, TypeScript), and domains (autonomous driving, code quality, DevOps). While this diversity supports generality, broader studies across additional projects and settings would further increase external validity.

Construct Validity: We mitigate construct threats by using complementary metrics—for example, failures-found per budget for selection, *APFD* for ranking—and using statistical significance tests and effect size measures to quantify differences.

IV. RESULTS

A. RQ1: Performance

RQ1 evaluates the effectiveness and efficiency of *CoEvo* by measuring how well it prioritizes and selects failing tests across budgets and its overall ranking quality (*APFD*), compared to *TeFa* and non-learning baselines. To provide a viable alternative for practitioners when failure histories are unavailable or incomplete, we need to establish performance relative to existing approaches to determine whether *CoEvo* delivers comparable performance while remaining simpler to deploy.

To see if the models can predict their respective labeling tasks, we first investigated precision, recall, and F1 for each model on its label type. As our models output probabilities of co-evolution or failure, we apply a 0.5 threshold to obtain binary predictions. We tested different thresholds in the interval [0.0, 1.0] with increments of 0.05. A value of 0.5 performed best according to the F1-score across projects and label types. Table II shows that *CoEvo* achieves the highest F1 across projects, indicating its strong ability to identify co-evolving tests accurately. *TeFa* performs well on *BMW* and *Gitlab* but has a lower recall on *Teamscale*. *TeFa (wo FR)* performs worst, particularly in recall, highlighting the dependency of *TeFa* on frequency data of recent test failures.

To evaluate *CoEvo* for test optimization, we assess its effectiveness and efficiency by (1) the percentage of failing tests found at each selection budget (fraction of the suite) and (2) overall ranking quality using *APFD*, comparing against

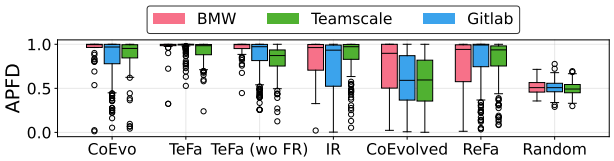

Fig. 4. *APFD* boxplot for all projects.

TABLE III

FRIEDMAN TEST AND PAIRWISE WILCOXON SIGNED-RANK TESTS (BONFERRONI-CORRECTED) WITH VARGHA-DELANEY $\hat{A}_{12}$ FOR *APFD* ACROSS ALL PROJECTS.

	BMW			*Teamscale*			*Gitlab*		
Friedman Test ($k=7$)	$p<0.01$			$p<0.01$			$p<0.01$		
Pairwise Comparison	$p_{corr.}$	$\hat{A}_{12}$	mag.	$p_{corr.}$	$\hat{A}_{12}$	mag.	$p_{corr.}$	$\hat{A}_{12}$	mag.
CoEvo vs *TeFa*	1.00	0.49	negl.	0.13	0.44	negl.	<0.01	0.30	medi.
CoEvo vs *CoEvolved*	0.08	0.59	smal.	<0.01	0.78	larg.	<0.01	0.71	medi.
CoEvo vs *ReFa*	0.04	0.67	medi.	0.85	0.58	smal.	1.00	0.44	negl.
CoEvo vs *Random*	<0.01	0.95	larg.	<0.01	0.95	larg.	<0.01	0.91	larg.
CoEvo vs *IR*	1.00	0.60	smal.	1.00	0.48	negl.	<0.01	0.60	smal.
CoEvo vs *TeFa (wo FR)*	1.00	0.49	negl.	<0.01	0.65	smal.	0.20	0.50	negl.
TeFa vs *CoEvolved*	0.01	0.59	smal.	<0.01	0.81	larg.	<0.01	0.82	larg.
TeFa vs *ReFa*	<0.01	0.71	medi.	<0.01	0.67	medi.	<0.01	0.61	smal.
TeFa vs *Random*	<0.01	0.96	larg.	<0.01	0.99	larg.	<0.01	1.00	larg.
TeFa vs *IR*	0.04	0.63	smal.	0.64	0.53	negl.	<0.01	0.79	larg.
TeFa vs *TeFa (wo FR)*	1.00	0.50	negl.	<0.01	0.73	medi.	<0.01	0.71	medi.
CoEvolved vs *ReFa*	1.00	0.52	negl.	<0.01	0.28	medi.	<0.01	0.29	medi.
CoEvolved vs *Random*	<0.01	0.72	medi.	0.04	0.60	smal.	<0.01	0.59	smal.
CoEvolved vs *IR*	1.00	0.44	negl.	<0.01	0.23	larg.	<0.01	0.37	smal.
CoEvolved vs *TeFa (wo FR)*	0.04	0.39	smal.	<0.01	0.28	medi.	<0.01	0.27	medi.
ReFa vs *Random*	<0.01	0.81	larg.	<0.01	0.86	larg.	<0.01	0.87	larg.
ReFa vs *IR*	1.00	0.43	negl.	1.00	0.39	smal.	0.06	0.63	smal.
ReFa vs *TeFa (wo FR)*	<0.01	0.32	medi.	1.00	0.58	smal.	1.00	0.56	negl.
Random vs *IR*	<0.01	0.12	larg.	<0.01	0.09	larg.	<0.01	0.24	larg.
Random vs *TeFa (wo FR)*	<0.01	0.02	larg.	<0.01	0.07	larg.	<0.01	0.08	larg.
IR vs *TeFa (wo FR)*	0.65	0.40	smal.	0.23	0.65	smal.	<0.01	0.39	smal.

TeFa and all baselines. Figure 3 reports the fraction of failed tests found across budgets (0–100%) per project. *TeFa* achieves the highest effectiveness, especially at low budgets, prioritizing failures early. *CoEvo* and *TeFa (wo FR)* perform similarly, close to *TeFa*, and surpass *IR* and *ReFa*—for *BMW* at all budgets, for *Teamscale* and *Gitlab* above 20%. *Random* finds the fewest failures at all budgets. All trained models outperform non-learning baselines across budgets, confirming the value of predictive optimization. The gaps between *TeFa* and *CoEvo* or *TeFa (wo FR)* shrink with larger budgets.

The per-project *APFD* in Fig. 4 summarizes the patterns of Fig. 3. *TeFa* yields the highest *APFD* (0.96 for *BMW*, 0.93 for *Teamscale*, 0.98 for *Gitlab*), prioritizing failures early. *CoEvo* (0.91, 0.88, 0.86) and *TeFa (wo FR)* (0.95, 0.82, 0.88) closely follow. Co-evolution thus delivers competitive effectiveness without prior failures. Heuristic and random baselines lag behind all predictive approaches on all projects: *IR* (0.83, 0.86, 0.75), *ReFa* (0.76, 0.82, 0.85), *CoEvolved* (0.73, 0.58, 0.60), and *Random* (0.51, 0.50, 0.51).

The Friedman test confirms significant differences among all seven approaches for every project ($p<0.01$; Table III). Bonferroni-corrected pairwise Wilcoxon signed-rank tests with Vargha-Delaney $\hat{A}_{12}$ effect sizes detail these differences. *CoEvo* significantly outperforms *Random* on all projects with large effects ($\hat{A}_{12}=0.91$–0.95, all $p<0.01$) and *CoEvolved* on *Teamscale* and *Gitlab* with large to medium effects ($\hat{A}_{12}=0.78$ and 0.71, both $p<0.01$). *CoEvo* also significantly outperforms *ReFa* on *BMW* with a medium effect ($\hat{A}_{12}=0.67$, $p=0.04$) and

IR on *Gitlab* with a small effect ($\hat{A}_{12} = 0.60$, $p < 0.01$). Against *TeFa*, *CoEvo* shows no significant difference on *BMW* ($p = 1.00$, $\hat{A}_{12} = 0.49$) and *Teamscale* ($p = 0.13$, $\hat{A}_{12} = 0.44$), both with negligible effects; only on *Gitlab* does *TeFa* significantly outperform *CoEvo* with a medium effect ($\hat{A}_{12} = 0.30$, $p < 0.01$). The comparison with *TeFa (wo FR)* yields negligible effects on *BMW* ($\hat{A}_{12} = 0.49$, $p = 1.00$) and *Gitlab* ($\hat{A}_{12} = 0.50$, $p = 0.20$) and a small effect on *Teamscale* ($\hat{A}_{12} = 0.65$, $p < 0.01$), indicating that historical fail-rate features are the main differentiator giving *TeFa* its edge over *CoEvo*.

To test this hypothesis, we analyzed whether *CoEvo*'s gains coincide with sparse failure history. In cases where *CoEvo* achieved higher *APFD* than *TeFa*, 66% of the involved tests had no prior failures, limiting *TeFa*'s ability to learn from history. Across the full test set, 39% of failing tests had no recorded prior failure history. This contrast indicates that *CoEvo* gains most when failure histories are sparse.

To better understand the shortcomings of *CoEvo*, we manually analyzed commits for which it performs worse than *Random*, that is, *APFD* < 0.5. For *BMW* this occurred for two commits, for *Teamscale* for six commits and *Gitlab* for 25 commits. In both commits of *BMW*, the failing test covered a large part of the code base and was never changed during the analysis period (only five copyright header changes throughout the lifetime of the project). Because these tests are broad in scope and rarely change, the training data expose too few discriminative features to capture a specific test and code relationship. Consequently, the model receives insufficient and infrequent signals to form accurate predictions, which results in low predicted relevance and the test not being selected.

For *Teamscale* we identified three different reasons for the bad performance in these commits. Three of the six commits had test failures that were caused by changes from prior commits. *Teamscale* runs a pipeline every time code is pushed to the version control, however, multiple commits without a push, or a merge from another branch cause the model to only see the changes from the last commit which did not cause the failing test. This is a technical limitation of our selection implementation and should be addressed in the future by looking at changes since the last test run. For two of the commits, changes to tests caused those tests to fail. These are difficult for the *CoEvo* model to find since changes to tests are removed from the training set, as the model aims to predict this. This could be addressed by prioritizing any changed tests as they arguably should be run in any case. The last commit has changes that directly cause the test failure. In that case, the model failed to detect the very indirect connection between code changes and the failing test.

Gitlab has 25 commits with *APFD* smaller than 0.5, all of which are virtual commits in a merge pipeline, designed to run tests on the already merged code state before introducing the changes on the main branch [42], [43], [44]. In these cases, there are test failures that were introduced on the main branch which also fail in the virtual commit and are unrelated to the changes of the merge request. In our dataset, tests failing on the main branch cause all of these pipeline failures. As there are no relevant changes for the tests, the model is not able to predict these failures. On the other hand, predicting these failures is not a priority as they do not uncover issues in the code changes.

In summary, *CoEvo* is effective and closely matches approaches relying on historical failure data and outperforms the non-learning baselines. While *TeFa* achieves the highest scores, *CoEvo* performs nearly as well, with only small differences in *APFD* scores and selected failed tests per budget. Including failure rates tends to prioritize tests that fail repeatedly, which can inflate the performance number of test prioritization techniques as shown in prior work [15]. Our further investigations showed the superior performance of *CoEvo* if test failure histories are sparse. Furthermore, our manual analysis revealed that shortcomings of *CoEvo* can be (in most cases) attributed to implementation details. In combination with the performance of *CoEvo* being up to par with *TeFa (wo FR)*, this shows that predictive test optimization based on co-evolution can achieve competitive results compared to the existing failure-based approach. Notably, *CoEvo* should be directly integrable into existing CI environments, as it only requires access to the version control system (which CI environments typically have), whereas *TeFa* needs parsing and storage of test results.

Performance. Predictive test optimization models based on co-evolution are nearly as performant as those based on historical failure data, with small differences in fault detection and clearly outperforming non-learning baselines, making it an obvious choice when failure data are unavailable, incomplete, or unreliable.

B. RQ2: Robustness

RQ2 assesses the robustness of *CoEvo* compared to *TeFa* under label noise—for example, test flakiness and accidental co-evolution—and quantifies how performance degrades as noise increases. As real CI environments exhibit noise, optimization that is sensitive to noise can be misleading and costly.

To evaluate the robustness of *CoEvo*, we analyze its performance across different noise levels compared to *TeFa*. We introduce synthetic noise into the training by randomly flipping the training labels according to the noise level and label type—as done in previous work [45]. We accept that our synthetic noise may not capture all real-world noise characteristics, but some root causes of flaky tests follow a uniform distribution—for example, random number generation. We adjust the failure and modification rates according to the flipped labels. To account for randomness we repeated each experiment ten times—including sampling and training—and report the average results. For each noise level, we analyze the distribution of *APFD* scores against the non-altered base data using the Wilcoxon signed-rank test and the Vargha-Delaney $\hat{A}_{12}$ measure to assess the stability of the approaches.

To quantify robustness, Fig. 5 shows boxplots of *APFD* for each approach and project at increasing noise levels. The Friedman test detects significant differences among noise levels

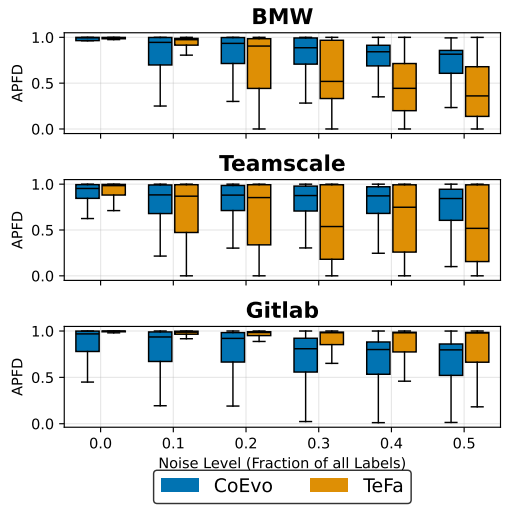

Fig. 5. *APFD* per project at different noise levels

for both *CoEvo* and *TeFa* on all three projects ($p < 0.01$). Bonferroni-corrected pairwise Wilcoxon tests compare each noise level against the unaltered baseline (noise $= 0.00$; see Table IV). For *BMW*, *CoEvo* shows no significant degradation at noise 0.10 ($p = 0.31$) and 0.20 ($p = 0.09$), with small to medium effects ($\hat{A}_{12} = 0.35$ and 0.31); degradation becomes significant from 0.30 onward ($p \leq 0.03$, $\hat{A}_{12} = 0.12$–0.25, large). In contrast, *TeFa* degrades significantly at every noise level ($p < 0.01$), consistently with large effects ($\hat{A}_{12} = 0.10$–0.20). For *Teamscale*, both *CoEvo* and *TeFa* show significant degradation at all noise levels ($p < 0.01$), but *CoEvo* exhibits only small effects up to noise 0.40 ($\hat{A}_{12} = 0.35$–0.40) and reaches a medium effect only at 0.50 ($\hat{A}_{12} = 0.28$), whereas *TeFa* shows large effects throughout ($\hat{A}_{12} = 0.22$–0.26). For *Gitlab*, *CoEvo* does not degrade significantly at noise 0.10 ($p = 0.06$, $\hat{A}_{12} = 0.37$, small) and shows small effects at 0.20 ($\hat{A}_{12} = 0.35$), transitioning to large effects from 0.30 onward ($\hat{A}_{12} = 0.24$–0.26). *TeFa* again shows significant large-effect degradation at every noise level ($\hat{A}_{12} = 0.16$–0.24, all $p < 0.01$). Overall, *CoEvo* resists label noise better than *TeFa*: it tolerates low noise without significant loss on two of three projects and consistently maintains higher $\hat{A}_{12}$ values (smaller degradation) than *TeFa* at every noise level. In summary, *CoEvo* demonstrates notable resistance to label noise, as for *BMW* and *Teamscale* it starts outperforming *TeFa* at all noise levels greater than 0.10 on average *APFD*-values with differences ranging from $+0.07$ to $+0.27$ and $+0.12$ to $+0.20$ respectively.

To understand the differences in sensitivity to label noise, we analyze the feature importance using SHAP-values [46], which quantify the impact of the individual features on the prediction results. Table V shows the mean SHAP-values for each feature across all projects and models. Generally, *CoEvo* prioritizes similarity and change history, while *TeFa* places more weight on test-file relationships and code modifications. For example, for *BMW* and *Teamscale*, *CoEvo* relies heavily on features like *Number of Common Path Tokens*, *File Cardinality*, and *Change History (7d)*, while *TeFa* focuses on features like *Test Name vs. Changed File Names (norm)* or *Changed Code vs. Test Code (added)*. The features influencing *CoEvo* reflect stable structural and historical aspects of the codebase, making *CoEvo*'s predictions less sensitive to short-term fluctuations and thus more robust. For *Gitlab*, both *CoEvo* and *TeFa* prioritize features related to recent code changes and test-file relationships, such as *Number of Changed Lines (added)* and *Changed Test File Names*. This overlap in influential features means that *CoEvo* is about as stable as *TeFa* for this project, since both models respond similarly to changes.

The interpretation of our synthetic label noise depends on the model: Real label noise in test failure data is caused by test flakiness, whereas noise in the co-evolution data represents co-evolution of unrelated code and tests. For example, experiments reveal that flaky failures reach up to 59% on *Teamscale* and 32% on *Gitlab* [34]. If we assume only test flakiness as noise source, then *CoEvo* would quickly outperform *TeFa* since it is not affected by it. For example, for *Teamscale*, *CoEvo* outperforms *TeFa* already for noise (flakiness) levels greater than 0.10, and reaches up to 0.34 improvement for higher levels. Similarly, for *BMW* the *APFD* values can be up to 0.41 higher for *CoEvo* when there is test flakiness.

TABLE IV
POST-HOC WILCOXON TEST RESULTS OF LABEL NOISE EVALUATION.

		CoEvo			*TeFa*		
Project	Noise	p	$\hat{A}_{12}$	Magnitude	p	$\hat{A}_{12}$	Magnitude
BMW	0.10	0.31	0.35	small	< 0.01	0.20	large
	0.20	0.09	0.31	medium	< 0.01	0.14	large
	0.30	0.03	0.25	large	< 0.01	0.11	large
	0.40	< 0.01	0.16	large	< 0.01	0.11	large
	0.50	< 0.01	0.12	large	< 0.01	0.10	large
Teamscale	0.10	< 0.01	0.40	small	< 0.01	0.26	large
	0.20	< 0.01	0.39	small	< 0.01	0.25	large
	0.30	< 0.01	0.37	small	< 0.01	0.23	large
	0.40	< 0.01	0.35	small	< 0.01	0.23	large
	0.50	< 0.01	0.28	medium	< 0.01	0.22	large
Gitlab	0.10	0.06	0.37	small	< 0.01	0.24	large
	0.20	< 0.01	0.35	small	< 0.01	0.20	large
	0.30	< 0.01	0.26	large	< 0.01	0.18	large
	0.40	< 0.01	0.25	large	< 0.01	0.17	large
	0.50	< 0.01	0.24	large	< 0.01	0.16	large

TABLE V
SHAP VALUES FOR EACH FEATURE, PROJECT, AND MODEL.

	BMW		*Teamscale*		*Gitlab*	
Feature	*CoEvo*	*TeFa*	*CoEvo*	*TeFa*	*CoEvo*	*TeFa*
File Cardinality	1.4	0.17	0.72	0.098	0.20	0.051
Number of Common Path Tokens	1.5	0.20	0.52	0.18	0.68	0.037
Changed File Package Cluster	0.17	0.034	0.062	0.13	0.20	0.12
Changed File Name Cluster	0.34	0.081	0.16	0.092	0.14	0.092
Number of Changed Lines (added)	0.11	0.32	0.14	0.041	0.82	0.24
Number of Changed Lines (deleted)	0.074	0.20	0.44	0.16	0.24	0.082
Test Name Cluster	0.050	0.000	0.12	0.18	0.11	0.12
Test Name vs. Changed File Names	0.044	0.41	0.19	0.27	0.24	0.14
Changed Code vs. Test Code (added)	0.032	0.22	0.31	0.33	0.18	0.048
Changed Code vs. Test Code (removed)	0.076	0.031	0.046	0.13	0.15	0.087
Test Name vs. Changed File Names (norm)	0.75	1.7	0.83	0.16	0.46	0.11
Changed Code vs. Test Code (added, norm)	0.12	0.066	0.52	0.042	0.60	0.38
Changed Code vs. Test Code (removed, norm)	0.055	0.0080	0.28	0.10	0.29	0.11
Changed Test File Names	-	0.53	-	0.022	-	1.1
Change History (7d)	1.0	0.042	1.1	0.19	0.65	0.21
Change History (14d)	0.24	0.000	0.21	0.061	0.15	0.13
Change History (28d)	0.062	0.17	0.18	0.20	0.15	0.050
Change History (56d)	0.011	0.000	0.13	0.038	0.18	0.40
Change History (total)	0.065	0.13	0.094	0.15	0.078	0.21
Historical Failure Rates (7d)	-	0.000	-	0.21	-	0.41
Historical Failure Rates (14d)	-	0.020	-	0.052	-	0.63
Historical Failure Rates (28d)	-	0.037	-	0.15	-	0.17
Historical Failure Rates (56d)	-	0.060	-	0.14	-	0.032
Historical Failure Rates (total)	-	0.24	-	0.64	-	0.37

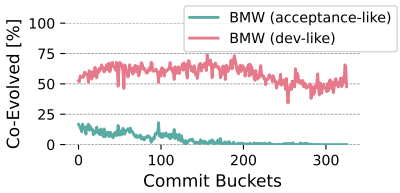

Fig. 6. Fraction of commits with co-evolved unit- and acceptance tests for *BMW* grouped into 1000-commit buckets over the lifespan of the project.

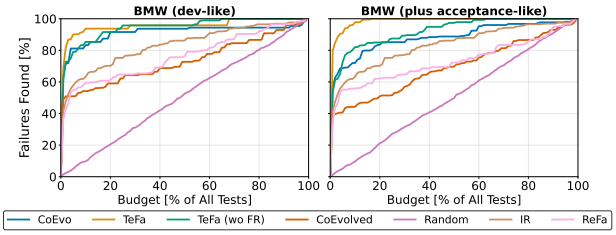

Fig. 7. Failures found per budget of *CoEvo* on only dev and dev plus acceptance-like tests at *BMW* compared to the other baselines.

Robustness. Predictive test optimization models based on co-evolution are more resistant to label noise than models based on failure labeling. They are not affected by test flakiness, making them suitable for environments with high flakiness.

C. RQ3: Versatility

Test suites may contain heterogeneous tests with different evolution dynamics. This raises the question whether the *CoEvo* approach generalizes across organizations and what tests provide co-evolution signals that are strong enough to be useful. RQ3 therefore investigates how *CoEvo* performs across test levels and types, and whether effectiveness depends on test granularity and coupling with code changes. To answer this RQ, we specifically focus on *BMW*, which develops safety-critical software with strict requirements enforced by invariant acceptance tests, which must be met to ship software to cars. These tests are mostly written once and rarely changed, as they are kept consistent with (safety) requirements throughout the project's development [47]. This can be seen in Fig. 6, which shows that acceptance tests co-evolve substantially less frequently with changes in the codebase compared to development tests (dev tests), which denote tests that are created and updated alongside the codebase and include commonly used test levels like unit, integration and system tests.

Figure 7 presents the percentage of failures found at varying budget levels for only dev tests and dev plus acceptance-like tests at *BMW*. The plot shows that *TeFa* consistently achieves the highest percentage of detected failures across all budgets, followed by *TeFa (wo FR)*. Both failure-based models outperform *CoEvo*, although *CoEvo* still outperforms *Random*, *ReFa*, *IR*, and *CoEvolved*.

Figure 8 confirms these observed differences. The Friedman test detects significant differences among all seven approaches for both configurations ($p < 0.01$). Bonferroni-corrected pairwise Wilcoxon tests reveal how adding acceptance tests shifts the comparisons. On dev tests only, *CoEvo* and

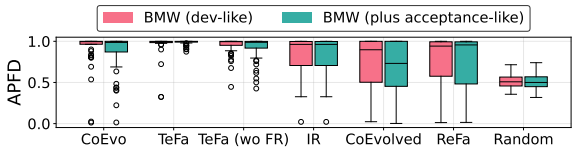

Fig. 8. Boxplot of *APFD*-values from *CoEvo* and all other baselines on only dev and dev plus acceptance-like tests for *BMW*.

TeFa show no significant difference ($\hat{A}_{12} = 0.49$, $p = 1.00$), consistent with the RQ1 results. When acceptance tests are included, *TeFa* significantly outperforms *CoEvo* with a small effect ($\hat{A}_{12} = 0.41$, $p < 0.01$). Similarly, *TeFa* and *TeFa (wo FR)* show no significant difference on dev tests only ($\hat{A}_{12} = 0.50$, $p = 1.00$), but *TeFa* significantly outperforms *TeFa (wo FR)* when acceptance tests are added ($\hat{A}_{12} = 0.64$, $p < 0.01$). *CoEvo* and *TeFa (wo FR)* show no significant difference in either configuration ($\hat{A}_{12} = 0.49$ and 0.51, both $p > 0.2$). *CoEvo* significantly outperforms *Random* in both configurations with large effects ($\hat{A}_{12} = 0.95$ and 0.91, both $p < 0.01$).

This indicates that (acceptance) tests that are less tightly coupled to code changes lead *CoEvo* to perform less effectively. Additionally, at *BMW*, acceptance tests are more likely to be run in post-merge pipelines because their broad scope leads to longer runtimes. As a result, such tests do not have to pass for changes to be merged, and developers are not required to co-evolve these tests. This limits the effectiveness of *CoEvo*, which relies on historical co-evolution data to predict relevant tests. *CoEvo* still provides an improvement over all heuristic and random approaches, in this challenging context.

Versatility. Co-evolution-based predictive test optimization is more effective for dev tests than for acceptance tests, due to their lower co-evolution frequency. *CoEvo* still consistently outperforms heuristic and random baselines.

V. CoEvo for Practitioners: A Phased Strategy for Immediate Deployment and Incremental Benefits

Predictive test optimization techniques address the problem of long-running test suites and have gained popularity in recent years. Practitioners require low setup and maintenance costs to adopt such techniques. Current predictive approaches rely on historical failure data that organizations must accumulate and validate before the models become effective [1], [11], [9].

Practitioners can deploy *CoEvo* immediately, as it only requires version history, which is available in almost every modern software project. Extracting *CoEvo* labels from the version history and training a model takes less than a single day on consumer hardware for our large-scale study objects, substantially less than a single execution of their entire test suites on such hardware. If test results are not available, practitioners need to either rerun their test suite to obtain them or wait until they are collected in the normal course of development. In contrast, *CoEvo* can be deployed immediately (potentially within one day) without any additional data collection, providing immediate benefits in test optimization and faster feedback cycles. Without any additional hard-to-obtain data, *CoEvo*

TABLE VI
CROSS-PROJECT *APFD* OF *CoEvo* TRAINED ON ONE PROJECT AND APPLIED TO ANOTHER.

Trained on ↓ / Applied to →	*BMW*	*Teamscale*	*Gitlab*
BMW	0.91 (±0.00)	0.75 (−0.13)	0.75 (−0.11)
Teamscale	0.90 (−0.01)	0.88 (±0.00)	0.80 (−0.06)
Gitlab	0.90 (−0.01)	0.85 (−0.03)	0.86 (±0.00)

outperforms all non-learning baselines (Section IV-A). To obtain fast feedback without missing important test failures, organizations can still run the full test suite periodically, for example, running predictive test optimization based on *CoEvo* during the week and the full test suite on the weekend.

While *CoEvo* runs in production, practitioners can begin collecting and validating failure records to further improve performance. This step requires engineers to design and implement tooling that parses and stores test results and to accumulate sufficient history. We tested how many weeks of filtered failure data *TeFa* requires reaching parity with *CoEvo*, by incrementally adding weeks of data, retraining, and evaluating performance. For *BMW* and *Teamscale TeFa* requires five weeks of filtered data, and one week on *Gitlab*.

Besides the time investment, practitioners must address potential flakiness before using failure data in a predictive model [22], [15], [34], as we discuss in Section II-B. Our noise-robustness evaluation (Section IV-B) shows that label noise from flaky tests significantly degrades *TeFa*'s performance. Because *CoEvo* draws its labels from version history rather than test outcomes, it remains robust under flakiness and can even assist the filtering process: a test that the model predicts as irrelevant to a change yet still fails likely signals flakiness.

Once validated failure data exist and practitioners have trained a failure-based model, they can form an ensemble that matches or exceeds the *TeFa* model by taking the maximum prediction score from both models for each test. This ensemble achieves an average *APFD* of 0.98 (+0.02) for *BMW*, 0.96 (±0.00) for *Gitlab*, and 0.93 (±0.00) for *Teamscale*.

Additionally, practitioners can transfer *CoEvo* models across projects with little degradation (−0.01 to −0.13 *APFD*; Table VI). However, project-specific models deliver superior performance with little additional effort, since projects that require test optimization typically have a sufficiently long version history. Obtaining training data and training a model takes less than a day, which is less than a single execution of the entire test suite of our study objects.

VI. RELATED WORK

Predictive Test Optimization: Researchers have extensively studied (regression) test optimization [48], [4], [5], applying techniques such as genetic algorithms [49], using test coverage [50], and historical, as well as dynamic analysis [51], [52], [53]; different ML-based techniques have been applied [10]. We discuss techniques related to predictive test optimization, as we introduce a new labeling strategy. Reinforcement learning appears promising for CI systems [54], [55], [56], [12], [10]. However, recent studies show that supervised learning (SL) outperforms it for test optimization in both open source [57] and industrial [58] contexts. We therefore adopt supervised learning. SL-based test optimization spans diverse techniques and inputs [59], [9], [60], [61], [62] and is deployed in production [1], [11], [38]. Building on work by Facebook [1], we use similar features, but replace failure history—often unavailable or unreliable [15], [24], [34]—with test and code co-evolution. This enables predictive selection where failure data are sparse. Hybrid approaches (using ML on top of static analysis) have been proposed [63], [11], [12], which our method supports. As simple heuristics can outperform complex ML [13], we compare against non-learning baselines.

Uses of Co-Evolution: Co-evolution is widely applied: patterns of test co-evolution enable automatic test generation [64], [65]. Given a change, researchers also predict co-evolving artifacts [66], clones [67], build configurations [68], and tests [69]. It supports traceability from tests to code [70], [71] and to requirements [72]; feature design [73]; fault localization [74], [75]; and modularity analysis via co-change structures [76], [77], [78]. The most related work predicts outdated tests using co-evolution [16], but targets maintenance rather than change-based testing. While co-evolution has been used to suggest relevant tests [79], it has not been used as label source for predictive test optimization.

VII. CONCLUSIONS

Large-scale, multi-language software systems require short feedback cycles, yet static and dynamic test optimization techniques struggle to scale [1]. Predictive test optimization [1] overcomes these limitations but depends on test failure records that may be unavailable or corrupted by flaky tests. We therefore introduced test and code co-evolution as an alternative labeling source that derives directly from version control.

Our evaluation on three large-scale projects—each over 1M LOC, two from industry—shows that co-evolution labels achieve an average *APFD* of 0.88, while not requiring test results, and remaining robust against flakiness.

These findings establish test and code co-evolution as a viable alternative and complementary signal for predictive test optimization. Practitioners can deploy co-evolution-based models immediately while collecting and validating test results. Co-evolution also strengthens test and code traceability: a failure the model does not predict may indicate flakiness.

Future work could analyze mispredictions [80], incorporate code and test representations via vectorization [81], [82], and labeling strategies for broad-scoped, rarely co-evolving tests.

To foster further research, we publish our implementation and data, including co-evolution and failure data from *Gitlab* in full and *Teamscale* with anonymized test names.

ACKNOWLEDGEMENTS

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "Q-SOFT, 01IS22001A". The sole responsibility for this article lies with the authors. We thank Fabian Leinen for providing us with the Gitlab test failure and flakiness data for our study.

References

[1] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2019, pp. 91–100.

[2] C. Pan and M. Pradel, "Continuous test suite failure prediction," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2021, pp. 553–565.

[3] S. Eldh, "On Technical Debt in Software Testing—Observations from Industry," in *Proceedings of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, 2022, pp. 301–323.

[4] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Journal of Software Testing, Verification and Reliability*, pp. 67–120, 2012.

[5] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Information and Software Technology*, pp. 14–30, 2010.

[6] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Conference on Software Maintenance (ICSM)*, 1999, pp. 179–188.

[7] R. Haas, R. Nömmer, E. Juergens, and S. Apel, "Optimization of automated and manual software tests in industrial practice: A survey and historical analysis," *IEEE Transactions on Software Engineering*, pp. 2005–2020, 2024.

[8] Á. Beszédes, T. Gergely, L. Schrettner, J. Jász, L. Langó, and T. Gyimóthy, "Code coverage-based regression test selection and prioritization in webkit," in *Conference on Software Maintenance (ICSM)*, 2012, pp. 46–55.

[9] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. C. Briand, "Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts," *IEEE Transactions on Software Engineering*, pp. 1615–1639, 2023.

[10] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. C. Briand, "Test case selection and prioritization using machine learning: a systematic literature review," *Empirical Software Engineering*, p. 29, 2022.

[11] A. Kondareddy, S. Azad, A. Singh, and T. A. D. Henderson, "Speculative testing at google with transition prediction," in *International Conference on Software Testing, Verification and Validation (ICST)*, 2025, pp. 510–521.

[12] D. Schwendner, M. Jungwirth, M. Gruber, M. Knoche, D. Merget, and G. Fraser, "Practical pipeline-aware regression test optimization for continuous integration," in *International Conference on Software Testing, Verification and Validation (ICST)*, 2025, pp. 371–381.

[13] D. Elsner, F. Hauer, A. Pretschner, and S. Reimer, "Empirically evaluating readily available information for regression test optimization in continuous integration," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2021, pp. 491–504.

[14] S. G. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 235–245.

[15] E. Fallahzadeh and P. C. Rigby, "The impact of flaky tests on historical test prioritization on chrome," in *International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2022, pp. 273–282.

[16] S. Wang, M. Wen, Y. Liu, Y. Wang, and R. Wu, "Understanding and facilitating the co-evolution of production and test code," in *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2021, pp. 272–283.

[17] A. Zaidman, B. V. Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, pp. 325–364, 2011.

[18] K. Beck, *Test driven development: By example*. Addison-Wesley, 2022.

[19] M. S. Farooq, U. Omer, A. Ramzan, M. A. Rasheed, and Z. Atal, "Behavior driven development: A systematic literature review," *i3eaccess*, pp. 88 008–88 024, 2023.

[20] N. Gupta, A. Sharma, and M. K. Pachariya, "An insight into test case optimization: ideas and trends with future perspectives," *i3eaccess*, pp. 22 310–22 327, 2019.

[21] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Transactions on Software Engineering and Methodology*, pp. 17:1–17:74, 2022.

[22] J. Micco, "The state of continuous integration testing @google," in *International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

[23] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models," *Empirical Software Engineering*, p. 72, 2023.

[24] F. Leinen, D. Elsner, A. Pretschner, A. Stahlbauer, M. Sailer, and E. Jürgens, "Cost of flaky tests in continuous integration: An industrial case study," in *International Conference on Software Testing, Verification and Validation (ICST)*, 2024, pp. 329–340.

[25] A. Joshi, S. Kale, S. Chandel, and D. K. Pal, "Likert scale: Explored and explained," *British journal of applied science & technology*, p. 396, 2015.

[26] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *International Conference on Software Engineering (ICSE)*, 2018, pp. 433–444.

[27] S. Robertson, "Understanding inverse document frequency: on theoretical arguments for idf," *Journal of documentation*, pp. 503–520, 2004.

[28] L. McInnes, J. Healy, S. Astels *et al.*, "hdbscan: Hierarchical density based clustering." *The Journal of Open Source Software*, p. 205, 2017.

[29] S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng, "Using semi-supervised clustering to improve regression test selection techniques," in *International Conference on Software Testing, Verification and Validation (ICST)*, 2011, pp. 1–10.

[30] U. Buatoom, W. Kongprawechnon, and T. Theeramunkong, "Document clustering using k-means with term weighting as similarity-based constraints," *Symmetry*, no. 6, p. 967, 2020.

[31] S. E. Robertson, S. Walker, M. Beaulieu, M. Gatford, and A. Payne, "Okapi at trec-4," *Nist Special Publication Sp*, pp. 73–96, 1996.

[32] Q. Peng, A. Shi, and L. Zhang, "Empirically revisiting and enhancing ir-based test-case prioritization," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2020, pp. 324–336.

[33] K.-J. Stol and B. Fitzgerald, "The abc of software engineering research," *ACM Transactions on Software Engineering and Methodology*, pp. 1–51, 2018.

[34] F. Leinen, M. Gruber, S. S. Erdogan, A. Stahlbauer, and A. Pretschner, "An empirical study of detected and undetected flaky test failures in real-world ci pipelines," unpublished manuscript.

[35] M. Jungwirth, S. Rummert, A. Scott, and G. Fraser, "Shifting gears in continuous integration: Bmw's strategies for high-velocity builds," in *International Conference on Software Testing, Verification and Validation Workshops (ICST-Workshops)*, 2025, pp. 158–159.

[36] M. Jungwirth, M. Gruber, and G. Fraser, "The good, the bad, and the flaky: A new sheriff in town to manage unstable tests at scale," in *International Conference on Software Testing, Verification and Validation Workshops (ICST-Workshops)*, 2026.

[37] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 2016, pp. 785–794.

[38] P. Plyusnin, A. Antonov, V. Ermakov, A. Khaybriev, M. Kikot, I. Alimova, and S. Moiseev, "Targeted test selection approach in continuous integration," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2025, pp. 1–12.

[39] X. Qu, M. B. Cohen, and K. M. Woolf, "Combinatorial interaction regression testing: A study of test case generation and prioritization," in *Conference on Software Maintenance (ICSM)*, 2007, pp. 255–264.

[40] M. Friedman, "The use of ranks to avoid the assumption of normality implicit in the analysis of variance," *Journal of the american statistical association*, pp. 675–701, 1937.

[41] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, pp. 101–132, 2000.

[42] S. Ananthanarayanan, M. S. Ardekani, D. Haenikel, B. Varadarajan, S. Soriano, D. Patel, and A. Adl-Tabatabai, "Keeping master green at scale," in *Proceedings of EuroSys Conference*, 2019, pp. 29:1–29:15.

[43] M. Jungwirth, M. Gruber, and G. Fraser, "Improving merge pipeline throughput in continuous integration via pull request prioritization," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2025, pp. 565–575.

[44] ——, "Predictive pull request batching to accelerate merge pipelines in continuous integration at scale," in *International Conference on Software Testing, Verification and Validation (ICST)*, 2026.

[45] B. Vancsics, T. Gergely, and Á. Beszédes, "Simulating the effect of test flakiness on fault localization effectiveness," in *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2020, pp. 28–35.

[46] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Conference on Neural Information Processing System*, 2017, pp. 4765–4774.

[47] P. Hsia, D. Kung, and C. Sell, "Software requirements and acceptance testing," *Annals of software Engineering*, pp. 291–317, 1997.

[48] R. Kazmi, D. N. A. Jawawi, R. Mohamad, and I. Ghani, "Effective regression test case selection: A systematic literature review," *ACM Computing Surveys*, 2017.

[49] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, pp. 225–237, 2007.

[50] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, pp. 929–948, 2001.

[51] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *International Conference on Software Engineering (ICSE)*, 2002, pp. 119–129.

[52] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *International Conference on Software Engineering (ICSE)*, 2015, pp. 713–716.

[53] Q. D. Soetens, S. Demeyer, A. Zaidman, and J. Pérez, "Change-based test selection: an empirical evaluation," *Empirical Software Engineering*, pp. 1990–2032, 2016.

[54] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2017, pp. 12–22.

[55] J. A. P. Lima and S. R. Vergilio, "A multi-armed bandit approach for test case prioritization in continuous integration environments," *IEEE Transactions on Software Engineering*, pp. 453–465, 2022.

[56] M. Bagherzadeh, N. Kahani, and L. Briand, "Reinforcement learning for test case prioritization," *IEEE Transactions on Software Engineering*, pp. 2836–2856, 2022.

[57] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo, "Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration," in *International Conference on Software Engineering (ICSE)*, 2020, pp. 1–12.

[58] J. Son, G. An, J. Hong, and S. Yoo, "Evaluating machine learning-based test case prioritization in the real world: An experiment with sap hana," in *International Conference on Software Testing, Verification and Validation (ICST)*, 2025, pp. 522–532.

[59] A. Sharif, D. Marijan, and M. Liaaen, "Deeporder: Deep learning for test case prioritization in continuous integration testing," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 525–534.

[60] E. Jabbar, H. Hemmati, and R. Feldt, "Investigating execution trace embedding for test case prioritization," in *International Conference on Software Quality, Reliability and Security (QRS)*, 2023, pp. 279–290.

[61] X. Jin and F. Servant, "Hybridcisave: A combined build and test selection approach in continuous integration," *ACM Transactions on Software Engineering and Methodology*, pp. 1–39, 2023.

[62] B. Busjaeger and T. Xie, "Learning for test prioritization: an industrial case study," in *International Symposium on Foundations of Software Engineering (FSE)*, 2016, pp. 975–980.

[63] J. Zhang, Y. Liu, M. Gligoric, O. Legunsen, and A. Shi, "Comparing and combining analysis-based and learning-based regression test selection," in *International Conference on Automation of Software Test (AST@ICSE)*, 2022, pp. 17–28.

[64] S. Shimmi and M. Rahimi, "Patterns of code-to-test co-evolution for automated test suite maintenance," in *International Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 116–127.

[65] ——, "Leveraging code-test co-evolution patterns for automated test case recommendation," in *International Conference on Automation of Software Test (AST@ICSE)*, 2022, pp. 65–76.

[66] A. Agrawal and R. K. Singh, "Predicting co-change probability in software applications using historical metadata," *IET Software Journal*, pp. 739–747, 2020.

[67] M. Mondal, C. K. Roy, and K. A. Schneider, "Prediction and ranking of co-change candidates for clones," in *International Conference on Mining Software Repositories (MSR)*, 2014, pp. 32–41.

[68] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan, "Cross-project build co-change prediction," in *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 311–320.

[69] L. Liu, S. Wang, Y. Liu, J. Deng, and S. Liu, "Drift: Fine-grained prediction of the co-evolution of production and test code via machine learning," in *Internetware*, 2023, pp. 227–237.

[70] B. V. Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *Conference on Software Maintenance and Reengineering (CSMR)*, 2009, pp. 209–218.

[71] R. M. Parizi, S. P. Lee, and M. Dabbagh, "Achievements and challenges in state-of-the-art software traceability between test and code artifacts," *IEEE Transactions on Reliability*, pp. 913–926, 2014.

[72] N. Ali, F. Jaafar, and A. E. Hassan, "Leveraging historical co-change information for requirements traceability," in *Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 361–370.

[73] C. Ziftci and I. Krueger, "Feature location using data mining on existing test-cases," in *Working Conference on Reverse Engineering (WCRE)*, 2012, pp. 155–164.

[74] J. Sohn and M. Papadakis, "CEMENT: on the use of evolutionary coupling between tests and code units. A case study on fault localization," in *International Symposium on Software Reliability Engineering (ISSRE)*, 2022, pp. 133–144.

[75] C. Tantithamthavorn, A. Ihara, and K. Matsumoto, "Using co-change histories to improve bug localization performance," in *Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2013, pp. 543–548.

[76] D. Beyer, "Ccvisu: automatic visual software decomposition," in *International Conference on Software Engineering: Companion Proceedings (ICSE Companion)*. ACM, 2008, pp. 967–968.

[77] M. Mondal, C. K. Roy, and K. A. Schneider, "Insight into a method co-change pattern to identify highly coupled methods: An empirical study," in *International Conference on Program Comprehension (ICPC)*, 2013, pp. 103–112.

[78] L. L. Silva, D. Felix, M. T. Valente, and M. de Almeida Maia, "Modularitycheck: A tool for assessing modularity using co-change clusters," *CoRR*, vol. abs/1506.05754, 2015.

[79] S. K. Bhrugumalla and A. K. Jha, "TREC: A regression test recommender for java projects," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2024, pp. 903–907.

[80] J. Cito, I. Dillig, S. Kim, V. Murali, and S. Chandra, "Explaining mispredictions of machine learning models using rule induction," in *International Symposium on Foundations of Software Engineering (FSE)*, 2021, pp. 716–727.

[81] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys*, pp. 81:1–81:37, 2018.

[82] A. Kicsi, L. Tóth, and L. Vidács, "Exploring the benefits of utilizing conceptual information in test-to-code traceability," in *International Conference on Software Engineering (ICSE)*, 2018, pp. 8–14.