Optimization of Automated and Manual Software Tests in Industrial Practice: A Survey and Historical Analysis

Roman Haas, Raphael Nömmer, Elmar Juergens and Sven Apel

***Abstract*—*Context*: Both automated and manual software testing are widely applied in practice. While being essential for project success and software quality, they are very resource-intensive, thus motivating the pursuit for optimization.**
***Goal*: We aim at understanding to what extent test optimization techniques for *automated* testing from the field of test case selection, prioritization, and test suite minimization can be applied to *manual* testing processes in practice.**
***Method*: We have studied the automated and manual testing process of five industrial study subjects from five different domains with different technological backgrounds and assessed the costs and benefits of test optimization techniques in industrial practice. In particular, we have carried out a cost–benefit analysis of two language-agnostic optimization techniques (test impact analysis and Pareto testing a technique we introduce in this paper) on 2,622 real-world failures from our subject's histories.**
***Results*: Both techniques maintain most of the fault detection capability while significantly reducing the test runtime. For automated testing, optimized test suites detect, on average, 80% of failures, while saving 66% of execution time, as compared to 81% failure detection rate for manual test suites and an average time saving of 43%. We observe an average speedup of the time to first failure of around 49 compared to a random test ordering.**
***Conclusion*: Our results suggest that optimization techniques from automated testing can be transferred to manual testing in industrial practice, resulting in lower test execution time and much lower time-to-feedback, but coming with process-related limitations and requirements for a successful implementation. All study subjects implemented one of our test optimization techniques in their processes, which demonstrates the practical impact of our findings.**

***Index Terms*—Software testing, manual testing, test optimization**

I. Introduction

SOFTWARE test suites grow with their systems under test [1]. So, for large software systems, the corresponding test suites are typically large [2]. Large test suites, no matter whether for automated or manual testing, take substantial time to run. Besides being expensive to execute, long test suite run times also prevent early and meaningful feedback to developers [3]–[5]. In a study among 38 testing teams of industrial software projects, Haas et al. found that a single execution of a manual test suite takes, on average, 1.5 person months, in extreme cases even up to six person months [6]. The test suites of our industry research partners (see Section III-B) run, on average, a whole work week for automated tests, and, on average, 5 person months for manual tests. They suffer from late feedback and lack of resources to run all tests in reasonable time.

In the literature, there are several techniques for improving test feedback times for long-running test suites. Two common techniques are *test case selection* [7] and *prioritization* [8], on which we focus in this work. While test case selection and prioritization are well-understood for automated tests [9]–[18], the transferability to manual testing is challenging. Inherently, manual tests are conducted less frequently. This leads to substantial code changes between test cycles, and functionality usually tested on the system level, resulting in each test covering a significant portion of code. Beside their different nature, manual tests fail to fulfill prerequisites of existing optimization techniques [6], for example, manual tests may be under-specified, resulting in different execution traces for multiple runs of the same test case. Additionally, execution traces may not be easily separable for different test runs if manual test environments are not isolated. Finally, different test processes, software development and test environments, and test suite run times dictate how aggressive an optimization technique needs to be to provide fast feedback while still keeping the fault detection rate as high as possible.

Research Gap: There is a lack of evidence of the extent to which test optimization techniques for automated testing can be applied to manual testing processes in industrial practice, and what limitations need to be accepted.

Solution: To address this research gap, we, first, analyze the processual differences between manual and automated testing in five large-scale industrial software projects, and we investigate whether their different test processes require different optimization strategies. Our aim is to discern the implications of the implemented test processes for the suitability of different optimization approaches. Second, we conduct a field experiment [19] applying two optimization techniques to five industrial software projects that implement automated or manual testing. Specifically, we apply two general language-agnostic optimization approaches to ease setup and allow for comparison of results between our study subjects: (1) test impact analysis, a code-change-dependent test selection and

Manuscript received January 1, 2134; revised February 1, 2134. This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant “Q-SOFT, 01IS22001A”, and the German Research Foundation (DFG), grant “AP 206/14-1”. The responsibility for this article lies with the authors. R. Haas and R. Nömmer were with the Saarbrücken Graduate School of Computer Science, Germany. R. Haas, R. Nömmer and E. Juergens were with CQSE GmbH, Germany. S. Apel was with Saarland University, Saarland Informatics Campus, Germany. (R. Haas and R. Nömmer contributed equally to this work.) (Corresponding authors: R. Haas and R. Nömmer.)

prioritization approach [20], and (2) Pareto testing, a technique we introduce in this paper that performs a static test selection that maximizes test coverage while minimizing test execution time.

Conduct and Results: To identify process differences between manual and automated testing, we have conducted a survey among our industry research partners. In our field experiment, we use historic test runs of our real-world study subjects to evaluate the costs and benefits of two general and language-agnostic test optimization techniques. Our results show that they are applicable in practice for automated and manual testing processes. For automated tests, 80% of failures are detected by the optimized test suites, on average, while saving 66% of execution time, compared to 81% failure detection rate for manual test suites and a time saving of, on average, 43%. All five industry partners that participated in our empirical study have adopted test impact analysis or Pareto testing into their processes following our results.

Contributions: Our contributions are the following:

- *Field experiment on five industry projects.* We investigate two language-agnostic test selection and prioritization techniques on a set of five test suites suffering from long execution times from industrial software projects from various domains, using different technologies, implementing manual and automated test processes.
- *New optimization technique.* We present a new optimization technique called Pareto testing, which is based on existing techniques from test prioritization and minimization and which aims at being simpler than test impact analysis in terms of set up and maintenance efforts.
- *Differences between automated and manual testing.* We carve out differences that are relevant for test suite optimization between automated and manual testing processes by surveying five test engineers and by querying the corresponding test suites.
- *Experiment on test optimization applicability in practice.* We conduct a field experiment to learn to which extent the optimization techniques are applicable to solve the issue of long-running test suites.
- *Analysis of test histories.* We analyze real data from more than 43,300 test cases, including test-wise coverage and, in total, 2,622 test failures from the study subjects' test histories to gain insights on the effectiveness of the optimization techniques.
- *Practice-oriented guidelines.* We provide a set of lessons learned enabling practitioners to implement the most suitable optimization technique in their testing process.

The questionnaire and details on the data analysis are available on a supplementary Web site (see Section VIII).

II. Background and Related Work

In this section, we describe terminology and notations that we use throughout the paper as well as relevant related work.

A. Software Testing and Test Reports

We follow the standard terminology of ISO/IEC/IEEE 29119-1 [21]. We use the terms *manual testing* and *automated testing* to differentiate between software verification processes where, following the IEEE standard, test cases are either "run manually by a human test executor, or [are] executed by a test automation tool" [21]. In the context of our empirical study, manual tests are usually conducted at a high abstraction level, that is, they are system tests, whereas automated tests span from unit to system levels. Given that our industry partners use their test suites for regression testing, performance testing, and user acceptance testing (refer to Section IV for specifics), our research centers on these test activities, also known as testing types per the IEEE standard. We refer to a *test cycle* to describe the execution of test cases with the same test goal (e.g., verification of one software version in a CI pipeline run or an entire test phase during manual release testing). For the test optimization techniques we consider in this work, we need data about each test case. For this, we use test reports, which are generated while running the test suite. Figure 1 illustrates a stylized test report and the stored data from each test run.

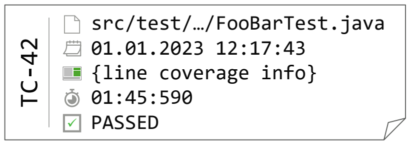


Fig. 1: Example of a test report; test reports consist of test identifier, source path, execution timestamp, line coverage information, duration, and the result

B. Test Case Prioritization

Test case prioritization aims at ordering a test suite $T = \{t_1, ..., t_n\}$ such that executing the tests t_i finds faults as early as possible. Technically, test case prioritization approaches strive for an optimal order of T where all failing t_f are executed before all passing t_p:

$$\forall t_i, t_j \in T, 1 \leq i, j \leq n \;:\; t_i \text{ fails} \land t_j \text{ passes} \Rightarrow i < j \quad (1)$$

As test results may be unpredictable, heuristics have been devised to predict failing tests and to derive prioritization approaches from these [22]. Typically, heuristics approximate the *fault revelation capability* of the test cases: $P_{\text{fr}}(t)$ denotes the probability of t to fail in the presence of a fault in the system under test. Especially, for manual testing, there are typically strict resource constraints (time and work force), so it is imperative to prioritize test cases to find as many faults as early as possible.

C. Test Case Selection

Test case selection strives for choosing a subset T_{fr} from a test suite T with the goal of saving testing efforts. It involves gathering test cases deemed execution-worthy based on a heuristic. While various criteria, such as high code coverage or testing critical code segments, could define the worthiness of a test case, we concentrate our selection process on tests anticipated to potentially yield failures:

$$\forall t \in T_{\text{fr}} \subseteq T : P_{\text{fr}}(t) > 0 \quad (2)$$

That is, we apply an *unsafe* technique, which might not succeed in selecting all failing tests. In practice, for large and complex software systems, safe selection techniques are infeasible (or ineffective) [11], [17], [23]. So, state-of-the-art selection techniques are unsafe [24], for example, because of dynamic dependencies [15], language boundaries [25], or non-code changes [26]. There has been substantial research on test case selection [27], [28]. Usually, an approximation of the single test case fault revelation capability is used to identify potential failures. In practice, the approximations are usually based on source code changes [27]. Intuitively, a test case that covers changed code might reveal new faults that have not been found by previous test runs. A previously passing test case that runs through unmodified code is expected not to change its behavior (which is not always true).

D. Optimization Techniques for our Field Experiment

For our field experiment, we selected two optimization techniques. First, we provide an overview of the requirements that the techniques need to meet. These criteria were collaboratively defined with our study subjects, as discussed in Section III-B. While all subjects suffered from the same problem—long-running test suites—there was a broad range of criteria concerning technologies, processes, and diverse economic, social, and legal requirements. We structured our field experiment to ensure some level of comparability across the subjects' contexts. Achieving this involved selecting optimization techniques that could be universally applied. This posed a significant challenge for several reasons. Firstly, we aimed for programming language-agnostic techniques to cater to the diverse range of real-world software systems. Secondly, our optimization techniques needed to be suitable for both automated and manual testing processes, each presenting their unique challenges (as detailed in Section I and discussed in Section V). Thirdly, our study subjects varied in size and complexity, and had slightly different optimization goals. Some focused on reducing test execution time by running a selective, change-focused subset of tests, while others prioritized running a diverse subset of tests for smoke tests. Fourthly, our subjects demanded well-established, intuitive optimization techniques that yield explainable and trust-worthy results from their perspective. Additionally, compliance with regulations was vital for systems in highly regulated industries. For instance, adherence to data protection regulations and stringent limitations on the processing of personal data (such as that of testers) based on legislation were critical considerations.

Recent techniques based on machine learning (for example, Yaraghi et al. [29]) are ruled out by these requirements, as many of the features are either not available for manual tests or not applicable in industrial contexts. For manual tests, first, features based on test source code are not available since the tests are documented in natural language. Second, since manual tests are executed far less frequently than automated tests, the number of historic test execution data available for each test case is limited. Third, in the case that historic test executions are available, features such as execution time and coverage vary more widely than they do for automated tests, which reduces their usefulness for a learning-based approach. For automated tests, we faced the additional challenge of large data sizes with some of our study subject. In the case of TIME, the test coverage report for a single test run was 13 GB in size. Since this accumulates quickly over many test executions each day, the developers only keep historic data for a single week. This, again, limits the possibility of using these data for learning-based test optimization. Finally, some data such as the committers for a file or the experience of a developer cannot be used due to data protection regulations.

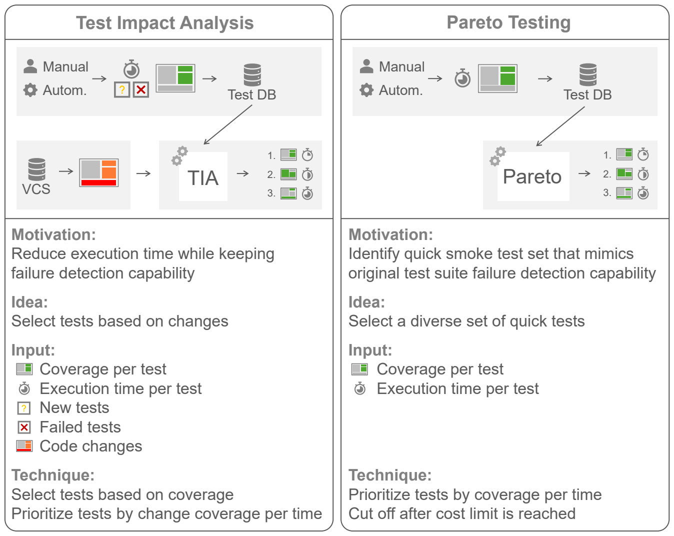
Given the resource-intensive nature of manual software testing, we limited the selection to two optimization techniques for the field experiment to manage evaluation costs per study subject effectively. We selected test impact analysis because—besides fulfilling the criteria outlined above—it is based on selection and prioritization approaches that have demonstrated effectiveness in the literature [30]–[33]. In our preliminary experiments, examining test impact analysis mostly on much smaller and open-source subjects with automated tests, we found that it performs well: For instance, the test suites optimized by test impact analysis maintain more than 90% of the original test detection capability, while saving 64% of test execution time, and have a median relative time-to-first-feedback of 2% [20]. These results encouraged us to investigate the applicability, benefits, and limitations of test impact analysis in large industry contexts including automated and manual testing, which we focus on in this work. It is important that the approaches are based on an intuitive fault model so that their results are easily understandable by the testers of our industry partners: Most bugs are introduced by code changes [34], so we assume that selecting all tests that cover changed code is intuitive to most people who work in software development. On top of the intuitive selection criteria, test impact analysis allows us to trace which tests were selected for a single changed method which, again, can give testers confidence in the technique. In addition, our test impact analysis implementation is based on a tooling platform that allows us to handle various programming languages and testing frameworks and environments [35], [36].

Our choice of Pareto testing was based on the existing implementation of test impact analysis, which provides a very solid basis. The idea of Pareto testing (i.e., we can only detect faults in parts of a system that we cover with tests), is also easily understood. Pareto testing reuses parts of the implementation but reduces the hardware and implementation requirements for data collection and processing that test impact analysis incurs, especially for very large projects. This reduction was a requirement for continuous use for some of our industry partners. Since its implementation is based on test impact analysis, it also benefits of the above-mentioned wide support for popular languages, frameworks, and testing environments.

Details of the two techniques will be outlined in the remaining section.

E. Test Impact Analysis

An overview of test impact analysis can be found on the left of Figure 2. Test impact analysis combines test case



Motivation:
Reduce execution time while keeping failure detection capability

Idea:
Select tests based on changes

Input:
- Coverage per test
- Execution time per test
- New tests
- Failed tests
- Code changes

Technique:
Select tests based on coverage
Prioritize tests by change coverage per time

Motivation:
Identify quick smoke test set that mimics original test suite failure detection capability

Idea:
Select a diverse set of quick tests

Input:
- Coverage per test
- Execution time per test

Technique:
Prioritize tests by coverage per time
Cut off after cost limit is reached

Fig. 2: Overview of the chosen test optimization techniques: test impact analysis (left) and Pareto testing (right)

selection and prioritization. It aims at identifying test failures as quickly as possible on the basis of code changes in a given timespan $[t_{\text{base}}, t_{\text{end}}]$. It aggregates the code changes since the last release or in the last sprint and provides an ordered set of test cases which—according to a heuristic—have the highest fault revelation capability, which is denoted as P_{fr}. Formally speaking, test impact analysis calculates an ordered set $O \subseteq T$ such that

$$\forall t_i, t_j \in T,\, 1 \leq i, j \leq n \;:\; P_{\text{fr}}(t_i) > P_{\text{fr}}(t_j) \Rightarrow i < j \tag{3}$$

In our context, test impact analysis selects test cases that execute code that has been modified within $[t_{\text{base}}, t_{\text{end}}]$, which failed in their most recent run, or which are new. The prioritization is based on a greedy cost–benefit calculation, where the costs $C(t)$ refer to the test execution time, opposing the benefit $B(t)$ of additional change coverage [36], [37]. The algorithm calculates a prioritization where for each rank $r \in [1, n]$ the test with the best cost–benefit ratio is chosen iteratively: So, in every iteration $i \in [1, n]$, $O[i] = t_i$ is chosen such that

$$t_i = \underset{t_j \,\in\, T \setminus O}{\arg\max} \frac{B(t_j)}{C(t_j)} \tag{4}$$

In the literature, there are several approaches that combine a test selection and prioritization strategy. For example, Greca et al. [38] propose a hybrid test optimization approach, which combines the tools *Ekstazi* for test selection [12] and *FAST* for test prioritization [39]. A recent systematic literature review summarizes the field of machine-learning-based test case selection and prioritization [18].

F. *Pareto Testing*

We have developed Pareto testing as a simpler alternative to test impact analysis, since practical usage at industry partners showed that obtaining and processing the input data for test impact analysis can be a substantial effort. For instance, a single test run at one of our study subjects produces 13 GB of coverage data. They implement continuous integration for all feature branches, have a large monolithic system, and more than three hundred active developers. Altogether, this setup leads to terabytes of coverage data that would need to be processed on a daily basis to run test impact analysis. Moreover, executing only a dynamically calculated subset of tests (e.g., based on the results of test impact analysis) is not always supported by the test runners implemented in industry, which can hinder the integration of test impact analysis in continuous integration setups. To overcome these drawbacks of test impact analysis, Pareto testing provides a more simplistic approach that can be set up and productively used with less effort. The Pareto test list T_p is not intended to be recalculated for every single test cycle, but less frequently (e.g., nightly or weekly), which reduces the costs of continuous coverage collection and integration into build pipelines. As a consequence, the underlying test case prioritization technique needs to be independent of a particular changeset.

We provide a brief overview of Pareto testing on the right of Figure 2: Pareto testing draws on ideas from the area of test suite minimization and test prioritization [5], [40]: it collects a diverse set of quick tests. For this, it uses a prioritization technique that orders tests to always include the one that adds the most additional coverage in the least amount of time. Once everything has been covered, the existing coverage is reset and starts again. The goal of this is to order the tests so that failing tests are run early. While the prioritization mechanism is the same as for test impact analysis (see Equation 4), Pareto testing is independent of a timespan or change set, and it does not include recently failed or new tests. Second, the highest prioritized tests from the list are picked and included in the test selection. Pareto testing is related to test suite minimization in that it selects a change-independent set of most relevant tests. However, it is not quite the same, since the selection does not aim at removing redundant tests permanently.

Formally, Pareto testing calculates an ordered subset T_p of a test suite T that runs within a cost limit of L. Each test case t has a cost function $C(t)$, representing its execution time. First, a test prioritization is applied, aiming for an ordering O such that failing test cases t_f are executed before passing ones t_p (see Equation 1). Then, the maximal number k of test cases fitting into the cost limit of L is determined:

$$k = \underset{1 \leq i \leq n}{\arg\max} \left(\left(\sum_{1 \leq l \leq i} C(t_l) \right) \leq L \right) \tag{5}$$

The selected test cases $t_1 \ldots t_k$ that comprise T_p are then the first k elements of the ordered tests O.

G. *Optimization of Manual Tests*

Research on test optimization has focused on automated test suites in the past. In practice, however, manual testing is widely used for large, complex, and regulated systems [6]. As the execution time of a manual test case is typically several orders of magnitude longer than it is for an automated test, the underlying issue of unmanageable test suite run times is even worse for manual testing [6], [41]. Consequently, there are several approaches that transfer results from automated test optimization to manual testing. Test selection techniques [6],

[42]–[46] as well as test prioritization [47]–[50] have been discussed, but only on a single or several similar subjects. In our empirical study, we include subjects from different domains relying either on automated or manual testing processes and implementing their system under test in different languages. Other optimization techniques refer to failure prediction [4], [51], test automation [52], and test suite reduction [44], [53].

H. Related Work

So far, we gave an overview about different test optimization techniques and provided references to core contributions of the field. Also, the optimization techniques we are using in this work are based on well-researched approaches. In this subsection, we delineate the setups of previous work to distinguish them from our work.

Evaluation focused on open-source systems: Most of the work evaluating the optimization techniques discussed so far focused on seeded faults and open source systems. More recent work incorporates other oracles, too, but still rely on open source systems as their study subjects for availability reasons. Peng et al. investigated several approaches for test case prioritization [54]. They combined and analyzed coverage, cost, historical failure, and information-retrieval-based prioritization approaches, and evaluated them on a large set of open source projects using faults from the projects histories. Cheng et al. prioritize test cases for cloud configuration testing and evaluate their work on 5 open source docker images [55]. Wang et al. combine test selection based on code and configuration changes, and use the same subjects as Cheng et al. to evaluate their approach [56]. Yaraghi et al. prioritize test cases in continuous integration contexts and evaluate their work on a set of more than 400 open source projects [29]. We conduct an empirical study on large industrial systems which come with their own difficulties and might show different behavior from what has been seen in open source projects.

Evaluation based on industry systems: There have been several papers focusing on test selection or prioritization for industry projects. In most cases, they are using a single industrial subject, often complemented by additional open source or generated subjects. For example, Marijan et al. apply a test selection approach that focuses on coverage and historical failures on an industry project that is supplemented by generated subjects based on the industry project [57]. Their approach is based on eliminating redundancy, that is, selecting tests that do not cover the same parts of the source code and do not fail together. There are also some studies looking into Google's [58] and Facebook's [16] approaches to test optimization. These studies deliver impressive results but are based on internal Google and Facebook projects, respectively, and do not consider applicability in other contexts an important factor. Recent work on cross-language regression test selection for C++ binaries [23] and on severity-aware prioritization of system-level regression tests [59] focuses on specific technologies and evaluates them in one specific industrial context. In summary, the focus of our work is different from most previous work in that we evaluate common test optimization techniques that fit well with our industry partners. Collaborating with them, we conducted a field experiment in a highly realistic environment across a wide range of industry systems, encompassing various languages and technologies.

III. Empirical Study: Survey and Field Experiment

We conduct an empirical study consisting of (i) a survey of our subjects' test leads and (ii) a field experiment to evaluate the selected test optimization techniques on our subjects' systems. By means of the survey, we collect insights into how testing is implemented in their industry contexts and how they intend to evolve their processes. We use these results to highlight differences between automated and manual testing. In our field experiment, following Stol and Fitzgerald [19], we adhere to their definition where the study takes place in a natural setting, that is, a realistic software development environment, involving changes directly impacting the studied entity, that is, the testing process. By this means, we investigate the applicability of two test optimization techniques for both testing strategies.

A. Research Questions

We address research questions (RQ) from two research areas (RA): (1) differences between automated and manual testing and (2) comparison of test impact analysis and Pareto testing.

RA$_1$: Test Strategies: Our study subjects employ automated and manual software testing (see Section III-B). As the underlying processes are quite different—which might affect optimization potentials and goals—we compare them on the basis of our subjects.

RQ$_{1.1}$: What test activities are performed and what are major characteristics of the test processes? We collect relevant data and provide an overview over the testing activities of our subjects to better understand their test processes and goals.

RQ$_{1.2}$: How much testing and maintenance effort is invested into automated and manual testing? We investigate the divergent maintenance and execution efforts between automated and manual testing in order to uncover optimization potential.

RQ$_{1.3}$: What are major bottlenecks in the testing process? We aim to understand the bottlenecks in our subjects' lengthy test processes to identify the most effective optimization potentials.

RQ$_{1.4}$: What are costs and benefits of the current testing process? We obtain a baseline for the evaluation of our optimization approaches (RA$_2$) and share quantitative data on the effectiveness of individual test cases and test suites to shed light on the structural differences between automated and manual tests.

RA$_2$: Test Optimization: We apply two optimization techniques, a test selection and prioritization (test impact analysis) and a minimization technique (Pareto testing), to automated and manual test suites of our subjects to learn when to apply which approach.

RQ$_{2.1}$: To what extent does the optimization technique influence the fault revelation capability of automated or manual tests? We focus on unsafe optimization techniques, which may not execute all potentially failing tests. As discussed in Section II, this is reasonable in an industry context. Still, we strive for optimization techniques that preserve the fault revelation capability of the test suites as much as possible.

RQ$_{2.2}$: What reasons lead to missed test failures for the optimization techniques? To uncover the reasons behind missed test failures by our optimization techniques, we aim to expose their practical limitations. Additionally, we seek to determine if these limitations vary between automated and manual testing.

RQ$_{2.3}$: What are costs and benefits of the optimization techniques? Using this research question, we aim to provide guidelines when to use what optimization technique in practice, possibly including a differentiation between automated and manual testing.

B. Study Subjects

Our empirical study aims at improving our understanding of test optimization in industrial practice. To obtain meaningful results, we work with data from private, public and even public-sector companies from different fields relying on different implementation and testing technologies, see also Table I. The project sizes range from a few ten thousands lines of code and teams with less than ten developers to several million lines of code with more than a hundred developers. The third column of Table I on test processes denotes whether we used their automated (A) or manual (M) test suite in our field experiment. Automated tests refer mostly to lower test levels, that is, Unit (U) or Integration (I), while manual tests are on the System (S) level. In the last column, we list the number of versions we analyzed for the each subject. Depending on the type of tests, that is, automated or manual, a version refers to a different interval, as described in Section III-E. The big differences in the number of versions are caused by the very different testing processes. While the automated tests of our subjects are in some cases run daily, the manual tests are only executed a few times a year and the collection of data required continuous support from our side, which limited the number of versions we were able to investigate. Also, the size and complexity of data of subject TIME limited the number of versions we could analyze there.

TABLE I: Overview of study subjects

Company	Domain	Test Proc	Test Levels	Team Size	SLOC	Lang	Versions
TIME[1]	Time Mgt.	A	U, I, S	50	8 M	Java	2
BVK	Finance	A	I, S	20	300 K	Java	543
DOLBY	Audio	A	U, I	10	28 K	C	111
ILP	ERP	M	S	5	831 K	C#	1
ZEISS	Optics	M	S	90	6 M	C#	1

[1] Subject anonymized due to NDA

Next, we introduce our subjects, their background, and their motivation for test suite optimization. We investigate the test processes in more detail in Section IV.

TIME is a software vendor (name changed for anonymization), has more than 600 employees, and a revenue of more than €100 million. In our field experiment, we are considering one of their core products. For this product, they have a very large test suite of 80,000 test cases, but even though they run their tests on 50 machines in parallel, it is not possible to test all maintained versions every night.

Bayerische Versorgungskammer (BVK) is Germany's largest pension group under public law and has about 1,490 employees. They build software for internal use and their customers. The project we are working with is running fast unit tests frequently, but the integration tests take around 10 h. The teams run nightly automated regression tests and to get faster feedback on their changes, they already employ Pareto testing during the day.

DOLBY works in the audio domain and has approximately 2,330 employees worldwide and offers a broad range of audio encoding, decoding and compression solutions. Some projects are using tests which need to cover lots of different configurations of audio systems, which opens up a large space of possible configurations for test cases. To avoid long feedback cycles, they were already using test impact analysis to select only the tests and test configurations that are relevant for recent changes.

ILP develops ENTRA®ERP, an enterprise resource planning system and is the smallest company among our study subjects with nine employees. They focus exclusively on manual testing, which is continuously performed—with more change-focused testing before a release, where the testing expert in the team selects tests that might be affected by the code change. Their test process includes both structured manual tests via Azure DevOps [60] and ad-hoc testing.

Carl Zeiss Microscopy (ZEISS) is a company that is manufacturing microscopes and has approximately 3,000 employees. We are looking at their ZEN MICROSCOPY Software which interfaces with their microscopes and provides control and configuration options, as well as visual results. They test their interfaces extensively with frequent automated tests, as well as manual tests on the actual hardware. They aim to test changed code as it is more likely to reveal bugs. For this purpose, they perform a manual expert-driven selection and prioritization of test cases.

On our supplementary Web site (see Section VIII), we present a table summarizing the study subjects along with some key statistics, for example, team size, the system under test's size, and the main implementation language.

C. Operationalization

We employed two methods to answer our research questions. Firstly, we designed a questionnaire that we sent to representatives of the testing teams of our subjects (see also Section III-D). Secondly, we analyzed data from the subject's test-suite management systems, including historical test results and coverage. The following section outlines how we utilized these data sources to address our research questions. More details on the data analysis are available on the supplementary Web site (see Section VIII).

RA$_1$: Test Strategies: We conduct a survey targeting the test leads of our subjects to capture the automated and manual testing processes currently in place. This includes the process characteristics as well as its intended evolution. See Table II for an overview on the mapping between the questionnaire and our research questions.

TABLE II: Survey questions to answer the research questions

RQ	Survey question
1.2	How many test engineers (e.g., testers, test developers) are there in your project?
1.2	How many test engineers spend their whole working time on testing?
1.2	How many test engineers work with the automated test suite?
1.2	How many test engineers work with the manual test suite?
1.1	Which test activities are performed via automated tests?
1.2	How much time do you estimate is invested into maintaining the automated test suite?
1.3	What are bottlenecks in your automated test process?
1.3	Are there flaky automated test cases?
1.1	Which test activities are performed manually?
1.1	How big is the manual test suite overall?
1.2	How many manual test cycles take place per year?
1.2	Is the entire manual test suite executed in every test cycle?
1.2	How many test cases are executed per manual test cycle?
1.1	Which events trigger the execution of a manual test case?
1.2, 1.4	How long does it take to execute the entire manual test suite?
1.2	How much time do you estimate is invested into maintaining the manual test suite?
1.3	What are bottlenecks in your manual test process?
1.3	Are there flaky manual test cases?

RQ$_{1.1}$: What test activities are performed and what are major characteristics of the test processes? We ask our survey participants about their automated and manual testing processes and what specific testing activities they implement, for example, regression testing or user acceptance testing. In addition, we consider the test suite size and the trigger events for test executions. If a manual test suite is maintained, we query its size and test execution triggers in the survey. For automated test suites, we obtain these data from the subjects' test management systems.

RQ$_{1.2}$: How much testing and maintenance effort is invested into automated and manual testing? We collect a set of statistics on our subjects' teams. To learn about the effort invested into automated testing, we query the execution time per test case and per test cycle from the test management systems. For manual tests, we ask in our questionnaire how often our subjects execute how many test cases as well as the time it takes to run a single test cycle. Additionally, we ask about the efforts that are spent for maintaining the manual and automated test suites.

RQ$_{1.3}$: What are major bottlenecks in the testing process? Our survey contains two open questions on the bottlenecks our subjects perceive in their testing process, as well as two questions on the existence of flaky tests. We cluster the responses systematically and report the relevant insights.

RQ$_{1.4}$: What are costs and benefits of the current testing process? To obtain a baseline for the evaluation of the two optimization approaches, we approximate the testing costs by the execution time per test case and test cycle. We measure the coverage per test suite as proxy for test benefits, as well as the fault revelation probability per cycle, and the average number of test failures revealed per cycle.

RA$_2$: Test Optimization: To answer our research questions regarding test optimization, we rely on historic development and test data of our subjects (TIME, BVK, DOLBY, ILP, ZEISS). For this purpose, they provided us access to their version control systems, test result history, historic test traces, covering many months or even years of data.

RQ$_{2.1}$: To what extent does the optimization technique influence the fault revelation capability of automated or manual tests? We run both optimization techniques on historic versions from our subjects' test suites and investigate the fault revelation capabilities of the optimized test suites. An ideal optimization technique would preserve the original fault revelation capability; that is, previous test failures are still detected, while running only those tests that find faults. Still, since the investigated optimization techniques are heuristics, missed test failures are possible. Section III-E explains in detail the measurement setup for automated and manual test suites to obtain the fault revelation capabilities for test impact analysis and Pareto testing.

To answer RQ$_{2.1}$ for test impact analysis, we determine the fault detection rate. For Pareto testing, we obtain the same metric for a set of cost limits: As described in Section II, Pareto testing takes a cost limit L as input parameter. This parameter is given by the subject's context. For our evaluation, we run the optimization technique with a set of cost limits $\mathscr{L} = \{1\%, 2\%, 3\%, 5\%, 10\%, 15\%, 20\%, 25\%, 30\%, 40\%, 50\%, 60\%, 80\%\}$.

For automated test suites, we report the detection rate of new failures (excluding subsequent failures), and we investigate whether applying test impact analysis and Pareto testing to an automated test suite leads to missed failing builds (this would imply that developers miss critical feedback as the optimized test suite passes while the original one failed). For manual tests, we do not report new test failures because data on new failures was not available at our subjects.

RQ$_{2.2}$: What reasons lead to missed test failures for the optimization techniques? To control the effort, we randomly select for both test impact analysis and Pareto testing a sample of $k = 10$ missed test failures (or all if there are fewer) for each subject and manually investigate why they were not detected.

RQ$_{2.3}$: What are costs and benefits of the optimization techniques? We quantify costs and benefits of test impact analysis and Pareto testing to be able to contrast them. The costs refer to test failures that were not detected by the approaches (see also RQ$_{2.1}$) and to a potential loss in overall test suite coverage (see also RQ$_{1.4}$). The benefits arise from saved execution time and earlier feedback from failing tests, that is, we measure the time to first failure and calculate by which factor the optimization approaches are faster than the baseline. Since the original execution order was not available or was subject to change for subsequent runs (e.g., for manual tests), for consistency reasons, we use a random selection, rather than the default execution order, as a more challenging baseline for all subjects. To obtain the time to first failure for the random baseline, we used a sample size of 1,000. For both optimization approaches, we distinguish between automated and manual testing, if applicable. As far as test impact analysis is concerned, we state for each subject how much time is saved relatively to the total execution time. For Pareto testing, the cost limit L relates directly to the time saving. As we run our experiments with a set of cost limits $\mathscr{L}$, we first identify an optimal cost limit, which balances the time investment $\sum_{i=1}^{k} C(t_i)$ and the detected failures F_L, to answer this research question. The optimal cost limit L_o minimizes the euclidean distance to the theoretical optimum, which detects all failures F without any costs, that is, in no time. In a plot of cost limits L and their fault detection rates, L_o would have the smallest distance to the top left corner of the chart. Formally, L_o can be written as:

$$L_o = \underset{L \in \mathscr{L}}{\arg\min} \sqrt{\left(\sum_{i=1}^{k} C(t_i)\right)^2 + \left(1 - \frac{F_L}{F}\right)^2} \qquad (6)$$

For Pareto testing, we additionally use the APFDc metric for our cost–benefit analysis, which is a cost aware modification of APFD (Average Percentage of Faults Detected) that was introduced by Rothermel et al. [33]. We do not evaluate test impact analysis using this metric, since test suites optimized by test impact analysis violate the APFDc requirement of equal total execution times. While the original APFDc metric considers cost as well as fault severity, we use a simplified version considering only cost, since we have no values for fault severities, as has been done in previous papers [54], [61]. Also, like Peng et al., we assume the worst-case, a one-to-one failure-to-fault mapping, since we lack the necessary access to our subjects' infrastructures. We compare the APFDc with a random approach based on a sample size of $1{,}000$ (the highest sampling size found for similar studies [62]). APFDc indicates how quickly faults can be found by a test suite in a specific order:

$$\text{APFDc} = \frac{\sum_{i=1}^{m}\left(\left(\sum_{j=TF_i}^{n} C(t_j)\right) - \frac{1}{2} C(t_{TF_i})\right)}{\sum_{j=1}^{n} C(t_j) \times m} \qquad (7)$$

where n is the number of tests, m the number of test failures, and TF_i is the first test that reveals fault i. Again, we compare the APFDc value of Pareto testing to the more ambitious random order.

D. Questionnaire and Conduct

With our survey, we collect experience from the responsible test leads at our subjects to better understand how testing is currently done in the specific industrial contexts. We strive for a detailed account of their testing processes and their real-world context, while exploring potential pathways or opinions for change. To this end, we designed a questionnaire to get an overview about the currently implemented testing processes of our subjects, their specific characteristics, and to elaborate their wishes for process changes addressing RA$_1$. We asked a representative of each of our five subjects, typically the corresponding test lead, to answer the questionnaire beginning in February 2023; by May 2023 all subjects answered. On our supplemental Web page, there is a table mapping our questionnaire's questions to our research questions. We used open-ended survey questions, allowing participants to describe their context.

E. Measurement Setup

RQ$_{2.1}$ is central for our field experiment. Our research is concerned with the application of two optimization techniques, test impact analysis, and Pareto testing, across automated and manual testing processes, each inherently distinct. Subjects with automated test suites typically execute the entire test suite at regular intervals, such as weekly or nightly. In contrast, manual testing involves the phased execution of all tests amidst ongoing code changes. Tailored measurement setups are essential for both processes to determine the fault revelation capability of optimized test suites. Figure 3 illustrates which data we include in the calculation for automated test suites (a) and manual test suites (b). In what follows, we describe the setup disparities between automated and manual testing, elucidating variations in data collection methodologies for test impact analysis and Pareto testing.

Automated Testing: For automated testing, depicted on the left in Figure 3, we determine for each test run how many failing tests our optimization techniques would select if applied prior to the run. The first test report is excluded from our evaluation because we need an initial set of test coverage data to apply prioritization. We calculate a Pareto list for all (but the very first) test executions, denoted by ◔ . For the calculation, we use the code and coverage state right before the next test report. For test impact analysis, we select test cases based on the changes starting from the first commit after test report up to the last commit before the next test report. In Figure 3, these commit intervals are denoted as ⎵ .

Manual Testing: For manual testing, illustrated on the right in Figure 3, full coverage data is only available after the first test phase, as it requires, at least, one execution of all test cases. The Pareto list is calculated (◔) based on all test executions of the initial test phase. Consequently, the test impact analysis timespan (⎵) covers all changes for the subsequent test phase. For both optimization approaches, the goal is to reveal as many test failures of the subsequent test phase as possible. To obtain complete test reports, coverage recording per test case is required. So, we integrated our tooling deeply into the subject's test management tools to trace test begin and end to trigger and stop coverage recordings appropriately. We analyzed potential outliers with our subjects' teams and excluded manual test executions with implausible execution times (e.g., two seconds, or two weeks).

IV. Results

In this section, we present our findings on the test processes of our study subjects (RA$_1$) and the cost and benefits of test impact analysis and Pareto testing (RA$_2$).

RA$_1$: Test Strategies

RQ$_{1.1}$: What test activities are performed and what are major characteristics of the test processes? Four subjects have an automated test suite used for regression testing (4), user acceptance testing (2), performance testing (2), robustness testing (1), smoke testing (1), and compatibility testing (1). Four subjects rely on manual tests (where ILP has no automated tests) for regression testing (3) and user acceptance testing (2). In addition, manual testing is used for performance testing (1).

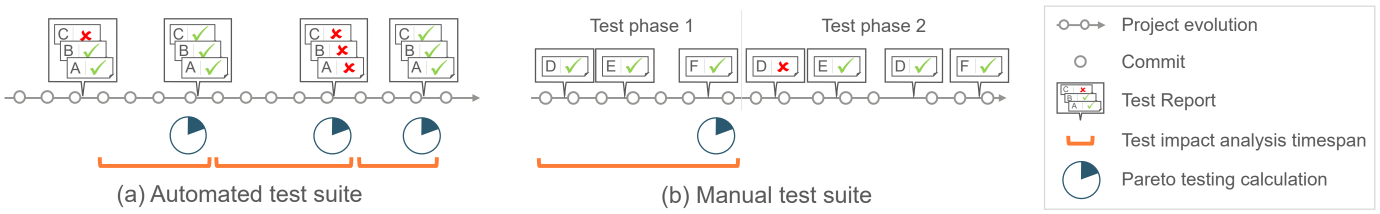


Fig. 3: Measurement setup for $RQ_{2.1}$ to determine the fault revelation capability of optimized test suites

TABLE III: Test suite and testing team sizes (numbers rounded)

Subject	Test Cases		Test Engineers	
	Automated	Manual	Automated	Manual
TIME	36,000	10,000	5	15
BVK	3,500	unknown	9	9
DOLBY	2,000	0	4	0
ILP	0	800	0	3
ZEISS	unknown	1,000	5	15

The test suite sizes vary greatly (second and third column of Table III): the relevant subjects have between two thousand and thirty-six thousand automated test cases. The manual test suites count between eight hundred and ten thousand test cases. All automated test suites are run regularly in a continuous integration pipeline, but the execution frequency ranges from runs per change to daily runs. Manual tests are run before scheduled releases, sometimes in every sprint, and on-demand for acceptance testing.

SUMMARY $RQ_{1.1}$. *All of our subjects with automated tests run them for regression testing, some perform additional automated test activities. The main purpose of manual tests are regression and user acceptance tests. The subjects' test suites tend to be large and are run frequently.*

$RQ_{1.2}$: How much testing and maintenance effort is invested into automated and manual testing? Table III, column four and five, list the number of test engineers (e.g., testers and test developers) involved in the software testing process. Overall, there are 3 to 20 test engineers per subject. Automated test cases take, on average, 11 s; the whole automated test suite, on average, 40.9 h. Up to 30% of the development and testing efforts are dedicated to maintaining automated tests. In contrast, most of the manual test suites are run only once per release, and there are 2 to 10 releases per year. Between 10 and 10,000 test cases are run per test cycle. Running the entire manual test suite takes between a few person-days and 225 person-days. Maintenance efforts are relatively low, ranging between 1 and 18 person-days per year.

SUMMARY $RQ_{1.2}$. *Our subjects invest considerable resources into automated and manual software testing. For automated testing, maintenance efforts are relatively large; whereas, for manual testing, the execution costs dominate maintenance costs.*

$RQ_{1.3}$: What are major bottlenecks in the testing process? For automated test suites, we found three major bottlenecks in the testing processes: specifying test cases, poor test suite architecture, and third-party components, which lack quick support of new framework versions. In what follows, we illustrate them with quotes from individual participants. Related to specifying test cases, ZEISS mentions, for example, the time-consuming "creation of tests" or at BVK it takes much time to "define the test preconditions". Regarding the test suite architecture, a participant stated that they "have much more tests on system level than on unit level", that is, their test suite has the shape of an ice cream cone instead of a test pyramid [63]. ILP and ZEISS report resource capacity as major bottlenecks in their manual testing process, and BVK reports the test data configuration during test case creation as major bottleneck. Flaky tests are perceived occurring more frequently for automated test suites (two out of four subjects); no flaky tests are reported in manual test suites.

SUMMARY $RQ_{1.3}$. *Our subjects identify the test suite design and the time-consuming creation of test cases as major bottlenecks of automated test suites. Resource capacities and test case creation are the major bottlenecks of manual testing. Flaky tests are bottlenecks only for automated test suites.*

$RQ_{1.4}$: What are costs and benefits of the current testing process? Figure 4 shows violin plots of the execution times per test case and the execution time per test cycle (which does not necessarily run the whole test suite). On average, a single automated test case takes 11 s to run, whereas a manual test runs for 29 min, on average. A test cycle for automated tests requires between 4 h and 122 h, while manual test cycles require 16 h to 180 h. On average, the test suites cover 69% of the methods of their respective systems; there is no substantial difference between automated and manual test suites with regard to coverage. Notably, the probability of, at least, one failing test per test cycle ranges from 13% to 89% for automated tests, while all manual test cycles revealed, at least, one failure. Overall, the probability of a failing cycle is 70%. On average, a test cycle revealed between 0.1 and 720 failures, where the extreme value of 720 stems from an automated test suite.

SUMMARY $RQ_{1.4}$. *An automated test cycle of our subjects runs up to 122 hours, while the longest manual test cycle requires 180 hours. Automated and manual test suites cover approximately 69% of the methods. 70% of all test cycles produce, at least, one test failure, all manual test cycles have produced, at least, one failure.*

RA_2: Test Optimization

In what follows, we present the results of our field experiment where we implemented test impact analysis and Pareto testing

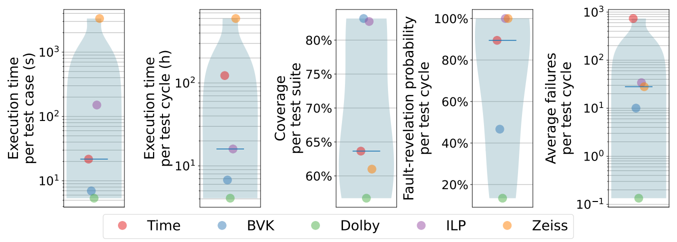


Fig. 4: Statistics of our subjects' current testing costs (execution time per test case and test cycle) and benefits (coverage and failures) (RQ$_{1.4}$)

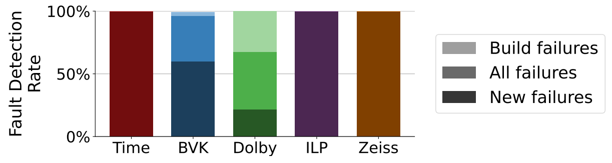


Fig. 5: Fault revelation capability of test suites optimized by test impact analysis

in the test processes of our five industrial subjects and ran a historical analysis. Our subjects have recorded their test execution history including test failure information and test-wise coverage information for, at most, three months up to several years. As described in Section III-E, this test history information allows us to answer the research questions on the test optimization approaches.

RQ$_{2.1}$: To what extent does the optimization technique influence the fault revelation capability of automated or manual tests? Figure 5 shows the fault revelation capability of test suites optimized by test impact analysis for all subjects. For all subjects, almost all build failures are detected. That is, when a build would fail with executing all tests, it would also fail with the optimized test set of test impact analysis. Only for subject BVK, test impact analysis misses 4 out of 542 build failures. For newly appearing test failures, on average, 76% are detected; an outlier is DOLBY with only 21%. Overall, on average, 93% of all failures are detected.

We show the results for Pareto testing in Figure 6. Since test selection is not based on changes, but on a chosen maximum execution time, we show failure detection over the range of cost limits $\mathscr{L}$ (see Section III-C). We observe that the number of detected build failures is increasing very early with the test execution time, while the number of failures and new failures increase a lot slower. For BVK and DOLBY, all build failures are detected fairly quickly for a comparably small $L \leq 10\%$. For TIME, which has 18 different build components, with $L = 1\%$ all build failures from 16 components are detected. To detect all build failures for TIME, we need to increase the test execution time limit to $L = 50\%$.

The overall fault revelation capability varies: on the one hand, for DOLBY, the cost limit $L = 5\%$ suffices to detect all failures. For BVK, on the other hand, $L = 80\%$ does not suffice to select all test failures. For BVK and DOLBY, the fault revelation capability for new failures is similar to the overall fault revelation capability.

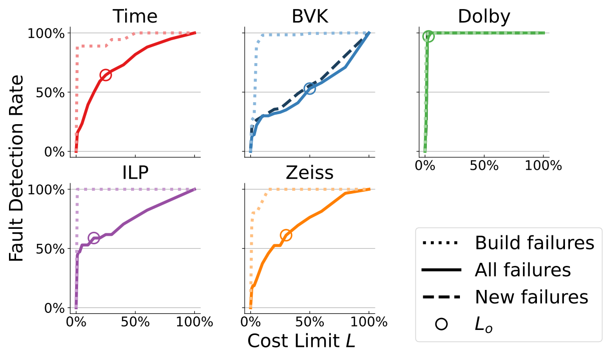


Fig. 6: Fault revelation capability of Pareto testing-optimized test suites per cost limit L

SUMMARY RQ$_{2.1}$. *For both automated and manual testing, test impact analysis detects, at least, 60% and up to 100% of historic failures (93%, on average). For Pareto testing, the cost limit L influences the fault revelation capability considerably, up to 100%. On average, it detects 53% of failures with $L = 10\%$.*

RQ$_{2.2}$: What reasons lead to missed test failures for the optimization techniques? As shown before, both optimization techniques missed some historical test failures. We analyzed for each technique and subject, if possible, ten missed failures and why they were not selected. Below, we summarize our findings and provide examples for illustration purposes.

Test impact analysis detected all test failures for three out of five subjects. For the remaining two subjects, 16 out of 20 investigated test failures are missed due to non-code changes of the system under test, its test suite, or environment changes, which are beyond the scope of test impact analysis. It did not detect XML-specified test case changes or failures of an infrastructure test, which only checks the testing environment.

For Pareto testing, we determined an optimal cost limit L_o that balances test execution time and failure revelation. Hence, missing some test failures, mostly from longer-running test cases, is expected. We observed this in our subjects, with unselected test cases revealing failures taking 30%–250% longer than selected tests. For one subject, all missed failures belonged to test cases for which the preceding run also failed. These failures may produce incomplete coverage information if the execution was cancelled, which may impact the prioritization of Pareto testing.

SUMMARY RQ$_{2.2}$. *For test impact analysis, 80% of missed test failures are related to a lack of coverage information for build or configuration files, which cannot be easily collected during test execution. For Pareto testing, some missed failures are to be expected due to the choice of L_o, and most missed test failures stem from long-running tests and previously failing tests.*

RQ$_{2.3}$: What are costs and benefits of the optimization techniques? In RQ$_{2.1}$, we addressed the fault revelation capability of test impact analysis (Figure 5) and Pareto testing (Figure 6). Figure 7 shows results regarding a cost–benefit analysis: *(a)* the relative coverage loss, *(b)* the relative execution time savings, *(c)* the speedup of time to first feedback compared with a random prioritization as baseline, and *(d)* the APFDc values of the optimized test suites. We examine these numbers in detail, since they require additional context to provide insights:

Test impact analysis for automated tests results in an average of 88% fault revelation of the full test suite. The main reason for the 12% loss in faults found are failures at BVK and DOLBY, where tests fail for reasons unrelated to code changes. At BVK, this is mainly due to the test descriptions, which are XML files and thus outside the scope of test impact analysis. At DOLBY, we found that many failing tests were caused by changes in the build, not by changes in the source code. The high loss of 43% coverage is mainly caused by many test runs at BVK, where test impact analysis on average selects very few tests. While this means that very little time has to be invested, it also results in very low coverage numbers. Since test impact analysis does not optimize for overall coverage but for change coverage, this is expected, when few changes happen in a time interval. In terms of benefits, test impact analysis saves 58% of test execution time, on average. Subject TIME has a big negative impact on this value. The test executions that we analyze for TIME have a longer interval of one week, even though their development is very active. This leads to the selection of all test cases, which reduces the overall time savings. In contrast, at BVK, we have short intervals and less development activity which allows for more time to be saved. Finally, the median time to first failure is 185 times faster compared to the random baseline, highlighting the effectiveness of change-based selection and, especially, prioritization in quickly identifying failures.

Test impact analysis for manual tests detects all historic failures and shows a negligible loss in coverage of, on average, 1% because of a comparably ineffective selection of, on average, 92% of the total test execution time. Test impact analysis selects this many test cases mainly because of two reasons. First, it needs to cover for both subjects a relatively long time span with many code changes, more than for our subject's automated test suites. Second, the manual test cases are end-to-end-tests, which cover much more code than, for example, automated unit tests. Since test impact analysis selects all test cases that cover code changes, a large set of code changes and test cases covering a lot of code lead to a large proportion of selected tests. Compared to a random baseline, the time-to-first-feedback is 4.75-times faster.

Pareto testing for automated tests has a lower fault detection rate than test impact analysis at 71%. Note that, for Pareto testing, this value depends on the optimal cost limits that we calculated (see Section III-C). As shown in Figure 6, the cost limits $\mathscr{L}_o$ are between 0.03 for DOLBY and 0.5 for BVK. Since test impact analysis selects tests based on code changes, it is expected to deliver a higher fault detection rate. The very low coverage loss of, on average, 1% is due to the fact that Pareto testing optimizes for coverage per time. The time savings are directly determined by our calculations, as mentioned in Section III-C. For the automated test suites, we achieve savings of, on average, 74%. Since this selection is based on overall coverage, and not on change coverage, there is no risk of selecting all tests. The median time to first failure is 5-times faster than for the random baseline. While still a solid improvement, this is far lower than for test impact analysis. Finally, the average APFDc value of 0.79 is very solid when comparing to recent results of Peng et al. [54].

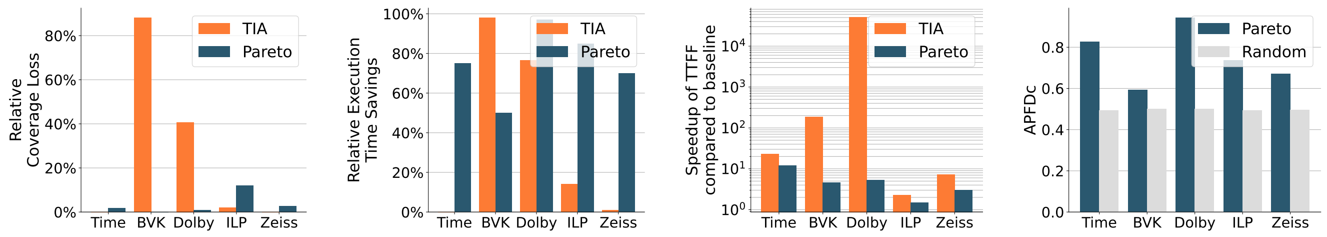
Pareto testing for manual tests results in a fault revelation rate of 60%, while saving 70–85% of execution time ($\mathscr{L}_o$ is 0.15 for ILP and 0.3 for ZEISS, see also Figure 6). We observe a small coverage loss of, on average, 5%. The time to first failure is 2.2-times faster than a random baseline, about half as fast as test impact analysis. The APFDc value of, on average, 0.7 is weaker than for automated tests, but still shows a good cost–benefit ratio.

SUMMARY RQ$_{2.3}$. *For automated tests, test impact analysis maintains 88% of the fault revelation capability of the full test suite, while saving 58% of execution time. We observe a median speedup of 185 for the time to first failure compared to random ordering. For manual tests, test impact analysis selects almost all test cases, so all failed tests are detected, but the time saving is only 8%. We observe a median speedup of 4.75. For automated tests, Pareto testing detects 71% of the failures while reducing the execution time by 74%. We observe a median speedup of 5. For manual tests, Pareto testing detects 60% of the failures while reducing the execution time by 78%. We observe a median speedup of 2.2.*

V. LESSONS LEARNED

In what follows, we summarize lessons learned supporting practitioners optimizing their testing processes. Our results show that test impact analysis and Pareto testing help to reduce testing efforts in industrial-scale automated and manual software testing processes, while still revealing the vast majority of test failures (refer to results from RA$_2$). We are convinced that other industry projects can benefit from the investigated test optimization techniques for their own testing processes. To assist practitioners in evaluating the applicability of these techniques in their context, we enrich the following guidelines with insights into the test strategies of our study subjects (as per RA$_1$).

Test optimization technique guidelines Test impact analysis is more sophisticated than Pareto testing; but it also comes with stricter requirements. Depending on the optimization goal, Pareto testing might be the better choice. It suggests a prioritized, diverse list of test cases within a given cost limit independently of code changes. This is useful to identify a (small) set of smoke tests, or (for very large test suites as for TIME) to identify a list of test cases that can be run overnight. In both cases, it is advisable to schedule additional, less frequent test executions of the whole test suite to update the coverage data of the test cases. Test impact analysis, in contrast, suggests a list of test cases on the basis of code changes within a given time range. This is useful for risk-based testing, as done at BVK and DOLBY, since modified code is more likely to contain bugs, which can only be revealed by test cases that execute the buggy code. It should be noted that, the more code has been changed, and the more test cases are required to cover all changes, the less effective is the selection of test cases.



(a) Relative coverage loss (b) Relative time savings (c) Speedup of time to first failure (TTFF) (d) APFDc (only relevant for Pareto testing)

Fig. 7: Costs and benefits of test impact analysis (TIA) and Pareto testing (with optimal cost limit L_o; see also Figure 6)

For example, for very large software projects with dozens or hundreds of developers on the same repository, for example TIME, the amount of code changes of one day might already exceed the available time for a test cycle overnight. The results for $RQ_{2.3}$ have shown that test impact analysis maintains a higher fault detection rate than Pareto testing for automated and manual tests, which comes with the cost of collecting test-wise coverage information regularly.

Importance of prioritization for manual testing End-to-end tests (such as most manual tests of our subjects) tend to cover more code regions per test than unit or integration tests. Moreover, manual tests are less frequently executed (2–10 cycles per year instead of daily or weekly runs, see $RQ_{1.2}$), which means that more changes need to be covered for test impact analysis. As a consequence, test impact analysis selected a large proportion of tests for both manual test suites in our subjects, which reduces the potential for time savings (see results of ILP and ZEISS for $RQ_{2.3}$). On the other hand, the subjects reported that the prioritization of test cases was very useful for them because it allows for great flexibility: The tests are executed in descending order of their fault revelation probability until there is no time left (e.g., end of planned test phase).

Variance in manual test reports We observed for both subjects implementing manual testing (ILP and ZEISS) that data derived from manual test reports exhibit greater variance than those from automated tests. This is mainly due to the inconsistencies inherent with manual testers in initiating, terminating, and executing test cases, for example, test cases can be under-specified [6], [42]. For coverage- or time-based test optimization techniques, this means that a slight variance in the results needs to be expected and that the exclusion of outliers may produce more useful results.

Cost limit parameter for test impact analysis In practice, for example, at BVK and ZEISS, the available time is often the limiting factor to run tests (see also the results for $RQ_{1.3}$). While Pareto testing has a cost limit by design, our implementation of test impact analysis does not consider such a cost limit. To address the need of an explicit optimization goal, we suggest for productive implementations of test impact analysis to add a cost limit input parameter, which is taken into account after the selection and prioritization of tests.

Transferability of optimization techniques Our field experiment shows that optimization techniques designed for automated testing can be applied for manual testing. To achieve uniform and comprehensive data, it is crucial to integrate test measurement tools, like a profiler, deeply into the tester's workflow. While the selection of manual test cases is less effective for our subjects, they find the prioritization very helpful. Overall, the optimization techniques provide useful results for automated and manual testing, and convinced our subjects to permanently implement them in their testing process. Concluding, this underpins the transferability of optimization techniques from automated testing to manual testing.

Post-study optimization of industrial testing processes Prior to our collaboration, the subjects were unaware of test optimization techniques, or there was no implementation available for their tech stack. Our language-agnostic implementations of test impact analysis and Pareto testing helped them evaluate these techniques within their own testing process. Notably, our field experiment's results convinced all five subjects to permanently implement an optimization technique in their testing process. For them, the benefits of optimization (e.g., the substantial resource savings and earlier feedback from failing tests) outweigh the costs (e.g., potentially missed failures and optimization data processing).

Recommendations for researchers and practitioners From our empirical study, we draw a number of recommendations for researchers and practitioners interested in implementing optimization techniques in large industrial software testing processes. Such processes involve many stakeholders with different experiences and stakes all of whom need to be on board and convinced of the benefits of the introduction of test optimization techniques for a successful implementation. Thus, it is vital to first understand the current testing process in detail (e.g., test strategies, tools, test environments, deployment strategies, test frequencies) and to discuss the optimization goals with the test management. This helps strengthen the understanding of people involved and to manage expectations, but also to fine-tune the optimization parameters.

We also recommend anticipating and prepare for technical challenges. Depending on the technical setup and the organization, these may be very different. Some likely issues that we have encountered during this empirical study and in other industry contexts:

- Access restrictions within test environments that needed to be addressed.
- For large systems, processing the volume of data required for the test optimization techniques poses a challenge.
- Non-isolated test environments can make it difficult to distinguish between concurrently running test cases.

Third, it is critical to involve the test teams that will be directly affected by the optimization results. Especially in the context of manual testing, using these techniques can have a big effect on the way teams perform their tests. They should be aware of the motivation for the process change, get an overview of the technical details of the optimization approach, learn about its impact on their testing process and have the opportunity to formulate their own expectations. Our experience working with manual test teams, in particular, has shown that there should be room for questions and suggestions for improvement, since the test teams usually have an excellent practical understanding of the process.

Finally, before rolling out an optimization technique to a larger testing process, we found that planning a dry run with a small subset of test cases or testers to identify potential blockers reduces the risk of major problems in the full rollout. In summary, we recommend treating the implementation of an optimization technique as a process change that requires active change and expectation management, as well as strong communication with all stakeholders.

VI. Threats to Validity

Re-running failing tests threatens the construct validity of our field experiment. Our field experiment is based on historic test runs of our study subjects. This real-world data set contains some repeatedly failing test cases. Since test impact analysis selects previously failed test cases for a re-run, repeatedly failing tests can influence the fault revelation capability of optimized test suites in our field experiment. As re-running failing tests is part our subjects' testing processes, we reflected this in the behavior of our test impact analysis implementation. We discuss the impact of re-running tests on our results in Section V.

Data quality presents a threat to internal validity. Both optimization techniques rely on testing data such as code coverage information. If this information is not accurate, it can lead to inaccurate test selection and worse performance for test prioritization. Thus, data quality is crucial for our field experiment. That is why we used field-tested tooling that is in productive use to measure test coverage, and we validated the data carefully with partners from our subjects to obtain meaningful results.

Non-code related failures threaten internal validity. For some subjects, automated test cases are specified in XML, which cannot be profiled by the profilers implemented at our subjects. As a consequence, test failures caused by test case modifications cannot be predicted by the optimization techniques, as there is no mapping between the XML test cases and the corresponding test executions. We encountered a similar situation with test cases that are concerned with build- or other non-code artifacts. In $RQ_{2.2}$, we have investigated limitations of the optimization techniques and discuss the impact of missing test specification and configuration data.

Flaky tests threaten the field experiment's internal validity as they might influence test impact analysis since it selects previously failed tests. While two subjects stated that they have flaky tests for their automated test suite, they only occur in low numbers.

The low number of versions for manual testing threatens conclusion validity. Recording the input data for manual test optimization proved to be a significant challenge in that we had to continuously support our study subjects with setup and usage of the infrastructure to obtain reliable results (see also Section III-E). For both subjects with manual tests, it took several test phases, each of which took several months, until we got complete and reliable data. Even though the final number of versions for manual tests is low because of these constraints, the data provide unique insights for optimizing manual testing processes that are inherently challenging to study, which makes our study results all the more valuable for practitioners.

Variance in manual test times threatens internal validity. The time recordings of manually executed tests can be imprecise (see Section V), which may impact both optimization techniques. We mitigated the threat of human errors by integrating our tooling as closely as possible into the regular testing process, so no additional actions needed to be taken by testers to record the data for the optimization techniques. In addition, we validated the manual test execution data carefully in close collaboration with our subjects to make sure that no invalid data go into our results. For example, we excluded outlier test cases with too small (a few seconds) or too large execution times (multiple days).

Generalizability of results in empirical software engineering research is often limited [64], which is a threat to external validity. This applies to our empirical study as well: Our subjects are not representative of all real-world software systems and their giant range of technologies and development and test processes. However, we considered a diverse set of five industrial software projects from different companies, domains and building up on different technologies. Our survey for RA_1: Test Strategies is meant to be a case study, so it does not claim to be representative for all industrial software engineering projects, nor that its results are generalizable across all industrial projects. We believe that it is important to understand the specific contexts of our subjects so that other researchers and practitioners can judge to what extent our results could also apply in their contexts, which is subject of an ongoing debate in the software community [65]. Still, in contrast to prior work on industrial systems, our empirical study on software testing optimization techniques is not tailored to individual subjects and, hence, is likely to be more transferable to other contexts. Our results show that the optimization techniques can be used for automated and manual testing processes, help to reduce time to feedback, and, in general, maintain the test suite's fault detection capability.

VII. Conclusion

Software testing is a common practice in industry, including both automated or manual processes. In this paper, we investigate to what extent optimization techniques that are typically used for automated software testing can be transferred to manual software testing. We have conducted an empirical study on five subjects from different domains, different tech stacks, varying regulatory requirements, and implementing different testing strategies. Their test processes are resource-intensive: up to twenty test engineers are involved in testing and a single test cycle runs up to four weeks. To carve out differences in their automated and manual testing processes, and their implications on optimizations, we conducted a survey among the test leads of our subjects. Then, in a field experiment, we applied two optimization techniques that select and prioritize test cases, test impact analysis and Pareto testing, and compared their costs and benefits in a historical analysis on our subjects' test suites. Our results show that both optimization techniques are applicable for automated and manual testing, even on large industry systems, and yield execution time savings of up to 98% for automated tests and 85% for manual tests, while preserving a fault detection capability of up to 96%. In conclusion, test optimization strategies—such as test selection, test prioritization, and test minimization—traditionally used for automated tests can be effectively transferred to manual testing, with only manageable limitations to be considered. More importantly, our results have practical impact, since all of our subjects implemented them in their software testing processes.

VIII. Data Availability

The raw data obtained in our empirical study cannot be shared because of confidentiality agreements. For reproducibility, we publish aggregated data and the analysis scripts, along with additional details on our subjects, our questionnaire, and a subject specific discussion of results on a supplementary Web site: https://zenodo.org/records/11502386 .

References

[1] A. Zaidman, B. van Rompaey, S. Demeyer, and A. van Deursen, "Mining Software Repositories to Study Co-Evolution of Production & Test Code," in *Proceedings of the International Conference on Software Testing, Verification, and Validation*. IEEE, 2008, pp. 220–229.

[2] L. S. Pinto, S. Sinha, and A. Orso, "Understanding Myths and Realities of Test-Suite Evolution," in *Proceedings of the International Symposium on the Foundations of Software Engineering*. ACM, 2012, pp. 1–11.

[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*. Springer, 2022, pp. 301–323.

[4] H. Hemmati and F. Sharifi, "Investigating NLP-Based Approaches for Predicting Manual Test Case Failure," in *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 2018, pp. 309–319.

[5] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Software Testing Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[6] R. Haas, D. Elsner, E. Juergens, A. Pretschner, and S. Apel, "How Can Manual Testing Processes Be Optimized? Developer Survey, Optimization Guidelines, and Case Studies," in *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2021, pp. 1281–1291.

[7] E. Engström, P. Runeson, and M. Skoglund, "A Systematic Review on Regression Test Selection Techniques," *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, 2010.

[8] C. Catal and D. Mishra, "Test Case Prioritization: A Systematic Mapping Study," *Software Quality Journal*, vol. 21, no. 3, pp. 445–478, 2013.

[9] S. Elbaum, A. Malishevsky, and G. Rothermel, "Prioritizing Test Cases for Regression Testing," in *Proceedings of the International Symposium on Software Testing and Analysis*. IEEE, 2000, pp. 101–112.

[10] ——, "Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2001, pp. 329–338.

[11] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for Improving Regression Testing in Continuous Integration Development Environments," in *Proceedings of the International Symposium on the Foundations of Software Engineering*. ACM, 2014, pp. 235–245.

[12] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight Test Selection," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2015, pp. 713–716.

[13] M. Harrold, R. Gupta, and M. Soffa, "A Methodology for Controlling the Size of a Test Suite," *Transactions on Software Engineering and Methodology*, vol. 2, no. 3, pp. 270–285, 1993.

[14] M. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker, "Empirical Studies of a Prediction Model for Regression Test Selection," *Transactions on Software Engineering*, vol. 27, no. 3, pp. 248–263, 2001.

[15] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An Extensive Study of Static Regression Test Selection in Modern Software Evolution," in *Proceedings of the International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 583–594.

[16] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive Test Selection," in *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 2019, pp. 91–100.

[17] A. Philip, R. Bhagwan, R. Kumar, C. Maddila, and N. Nagppan, "FastLane: Test Minimization for Rapidly Deployed Large-Scale Online Services," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2019, pp. 408–418.

[18] 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*, vol. 27, no. 2, pp. 1–43, 2022.

[19] K. Stol and B. Fitzgerald, "The ABC of Software Engineering Research," *Transactions on Software Engineering and Methodology*, vol. 27, no. 3, 2018.

[20] E. Juergens, D. Pagano, and A. Goeb, "Test impact analysis: Detecting errors early despite large, long-running test suites," CQSE GmbH, White Paper, 2018.

[21] n. A., "International standard—software and systems engineering – software testing –part 1:general concepts," *ISO/IEC/IEEE 29119-1:2022(E)*, pp. 1–60, 2022.

[22] M. Khatibsyarbini, M. A. Isa, D. N. A. Jawawi, and R. Tumeng, "Test Case Prioritization Approaches in Regression Testing: A Systematic Literature Review," *Information and Software Technology*, vol. 93, pp. 74–93, 2018.

[23] D. Elsner, S. Kacianka, S. Lipp, A. Pretschner, A. Habermann, M. Graber, and S. Reimer, "BinaryRTS: Cross-language Regression Test Selection for C++ Binaries in CI," in *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 2023.

[24] D. Elsner, F. Hauer, A. Pretschner, and S. Reimer, "Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration," in *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2021, pp. 491–504.

[25] A. Celik, M. Vasic, A. Milicevic, and M. Gligoric, "Regression Test Selection Across JVM Boundaries," in *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2017, pp. 809–820.

[26] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso, "Regression Testing in the Presence of Non-Code Changes," in *Proceedings of the International Conference on Software Testing, Verification, and Validation*. IEEE, 2011, pp. 21–30.

[27] R. Kazmi, D. Jawawi, R. Mohamad, and I. Ghani, "Effective Regression Test Case Selection: A Systematic Literature Review," *Computing Surveys*, vol. 50, no. 2, pp. 1–32, 2017.

[28] J. Zhang, Y. Liu, M. Gligoric, O. Legunsen, and A. Shi, "Comparing and Combining Analysis-Based and Learning-Based Regression Test Selection," in *Proceedings of the International Conference on Automation of Software Test*. IEEE, 2022, pp. 17–28.

[29] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. C. Briand, "Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts," *Transactions on Software Engineering*, vol. 49, no. 4, pp. 1615–1639, 2023.

[30] Á. Beszédes, T. Gergely, L. Schrettner, J. Jász, L. Lango, and T. Gyimóthy, "Code Coverage-based Regression Test Selection and Prioritization in WebKit," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 2012, pp. 46–55.

[31] P. G. Frankl, G. Rothermel, K. Sayre, and F. I. Vokolos, "An Empirical Comparison of Two Safe Regression Test Selection Techniques," in *Proceedings of the International Symposium on Empirical Software Engineering*. IEEE, 2003, pp. 195–204.

[32] Y. F. Chen, D. S. Rosenblum, and K. phong Vo, "Test Tube: a system for selective regression testing," in *Proceedings of the International Conference on Software Engineering*, 1994, pp. 211–220.

[33] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test Case Prioritization: an Empirical Study," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 1999, pp. 179–188.

[34] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. M. Gonzalez-Barahona, "How bugs are born: a model to identify how bugs are introduced in software components," *Empirical Software Engineering*, vol. 25, pp. 1294–1340, 2020.

[35] R. Haas, R. Niedermayr, and E. Juergens, "Teamscale: Tackle Technical Debt and Control the Quality of Your Software," in *International Conference on Technical Debt*. IEEE, 2019, pp. 55–56.

[36] J. Rott, "Test Intelligence: How Modern Analyses and Visualizations in Teamscale Support Software Testing," in *Proceedings of the International Workshop on Visualization in Testing of Hardware, Software, and Manufacturing*. IEEE, 2022, pp. 15–21.

[37] R. Noemmer, "Conception and Evaluation of Test Suite Minimization Techniques for Regression Testing in Practice," Technical University of Munich, Master's Thesis, 2019.

[38] R. Greca, B. Miranda, M. Gligoric, and A. Bertolino, "Comparing and Combining File-based Selection and Similarity-based Prioritization towards Regression Test Orchestration," in *Proceedings of the International Conference on Automation of Software Test*. IEEE, 2022, pp. 115–125.

[39] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "FAST Approaches to Scalable Similarity-Based Test Case Prioritization," in *Proceedings of the International Conference on Software Engineering*. ACM, 2018, pp. 222–232.

[40] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and Combining Test-Suite Reduction and Regression Test Selection," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2015, pp. 237–247.

[41] E. Enoiu, D. Sundmark, A. Čaušević, and P. Pettersson, "A Comparative Study of Manual and Automated Testing for Industrial Control Software," in *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 2017, pp. 412–417.

[42] E. Juergens, B. Hummel, F. Deissenboeck, M. Feilkas, C. Schlögel, and A. Wübbeke, "Regression Test Selection of Manual System Tests in Practice," in *Proceedings of the European Conference on Software Maintenance and Reengineering*. IEEE, 2011, pp. 309–312.

[43] S. Eder, B. Hauptmann, M. Junker, R. Vaas, and K.-H. Prommer, "Selecting Manual Regression Test Cases Automatically using Trace Link Recovery and Change Coverage," in *Proceedings of the International Workshop on Automation of Software Test*. ACM, 2014, pp. 29–35.

[44] D. Di Nardo, N. Alshahwan, L. C. Briand, and Y. Labiche, "Coverage-Based Regression Test Case Selection, Minimization and Prioritization: A Case Study on an Industrial System," *Software Testing, Verification and Reliability*, vol. 25, no. 4, pp. 371–396, 2015.

[45] T. Nakagawa, K. Munakata, and K. Yamamoto, "Applying Modified Code Entity-Based Regression Test Selection for Manual End-To-End Testing of Commercial Web Applications," in *Proceedings of the International Symposium on Software Reliability Engineering Workshops*. IEEE, 2019, pp. 1–6.

[46] H. Zhong, L. Zhang, and S. Khurshid, "TestSage: Regression Test Selection for Large-Scale Web Service Testing," in *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 2019, pp. 430–440.

[47] H. Hemmati, Z. Fang, and M. Mäntylä, "Prioritizing Manual Test Cases in Traditional and Rapid Release Environments," in *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 2015, pp. 1–10.

[48] R. Lachmann, M. Nieke, C. Seidl, I. Schaefer, and S. Schulze, "System-Level Test Case Prioritization using Machine Learning," in *Proceedings of the International Conference on Machine Learning and Applications*. IEEE, 2016, pp. 361–368.

[49] S. Y. Shin, S. Nejati, M. Sabetzadeh, L. C. Briand, and F. Zimmer, "Test Case Prioritization for Acceptance Testing of Cyber Physical Systems: A Multi-Objective Search-Based Approach," in *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 49–60.

[50] M. Bagherzadeh, N. Kahani, and L. C. Briand, "Reinforcement Learning for Test Case Prioritization," *Transactions on Software Engineering*, vol. 48, no. 8, 2021.

[51] M. Golagha, A. Pretschner, and L. C. Briand, "Can We Predict the Quality of Spectrum-based Fault Localization?" in *Proceedings of the International Conference on Software Testing, Validation and Verification*. IEEE, 2020, pp. 4–15.

[52] H. Kirinuki and H. Tanno, "Automating End-to-End Web Testing via Manual Testing," *Journal of Information Processing*, vol. 30, pp. 294–306, 2022.

[53] A. Shi, A. Gyori, S. Mahmood, P. Zhao, and D. Marinov, "Evaluating Test-Suite Reduction in Real Software Evolution," in *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 84–94.

[54] Q. Peng, A. Shi, and L. Zhang, "Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization," in *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2020, pp. 324–336.

[55] R. Cheng, L. Zhang, D. Marinov, and T. Xu, "Test-Case Prioritization for Configuration Testing," in *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2021, pp. 452–465.

[56] S. Wang, Y. Lian, D. Marinov, and T. Xu, "Test Selection for Unified Regression Testing," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2023, p. n. p.

[57] D. Marijan and M. Liaaen, "Practical Selective Regression Testing with Effective Redundancy in Interleaved Tests," in *Proceedings of the International Conference on Software Engineering*. ACM, 2018, pp. 153–162.

[58] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming Google-Scale Continuous Testing," in *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 2017, pp. 233–242.

[59] R. Wuersching, D. Elsner, F. Leinen, A. Pretschner, G. Grueneissl, T. Neumeyr, and T. Vosseler, "Severity-Aware Prioritization of System-Level Regression Tests in Automotive Software," in *Conference on Software Testing, Verification and Validation*. IEEE, 2023, pp. 398–409.

[60] (2023) Microsoft azure devops server. [Online]. Available: https://azure.microsoft.com/en-us/products/devops/server/

[61] Q. Luo, K. Moran, D. Poshyvanyk, and M. Di Penta, "Assessing Test Case Prioritization on Real Faults and Mutants," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 2018, pp. 240–251.

[62] P. Yi, H. Wang, T. Xie, D. Marinov, and W. Lam, "A Theoretical Analysis of Random Regression Test Prioritization," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 217–235.

[63] M. Fowler. (2012) Testpyramid. [Online]. Available: https://martinfowler.com/bliki/TestPyramid.html

[64] J. Siegmund, N. Siegmund, and S. Apel, "Views on Internal and External Validity in Empirical Software Engineering," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2015, pp. 9–19.

[65] L. Briand, D. Bianculli, S. Nejati, F. Pastore, and M. Sabetzadeh, "The case for context-driven software engineering research: generalizability is overrated," *IEEE Software*, vol. 34, no. 5, pp. 72–75, 2017.