Prioritizing Test Gaps by Risk in Industrial Practice: An Automated Approach and Multi-Method Study

Roman Haas, Michael Sailer, Mitchell Joblin, Elmar Juergens, and Sven Apel

***Abstract*—*Context.* Untested code changes, called *test gaps*, pose a significant risk for software projects. Since test gaps increase the probability of defects, managing test gaps and their individual risk is important, especially for rapidly changing software systems.**

***Objective.* This study aims at gaining an understanding of test gaps in industrial practice establishing criteria for precise prioritization of test gaps by their risk, informing practitioners that need to manage, review, and act on larger sets of test gaps.**

***Method.* We propose an automated approach for prioritizing test gaps based on key risk criteria. By means of an analysis of 31 historical test gap reviews from 8 industrial software systems of our industrial partners Munich Re and LV 1871, and by conducting semi-structured interviews with the 6 quality engineers that authored the historical test gap reviews, we validate the transferability of the identified risk criteria, such as code criticality and complexity metrics.**

***Results.* Our automated approach exhibits a ranking performance equivalent to expert assessments, in that test gaps labelled as risky in historical test gap reviews are highly ranked, on average, on the 23rd percentile. In some scenarios, our automated ranking system even outpaces expert assessments, especially for test gaps in central code—for non-developers an opaque code property.**

***Conclusion.* This research underscores the industrial need of test gap risk estimation techniques to assist test management and quality assurance teams in identifying and addressing critical test gaps. Our multi-method study shows that even a lightweight prioritization approach helps practitioners to identify high-risk test gaps efficiently and to filter out low-risk test gaps.**

***Index Terms*—Software Testing, Test Gap Analysis, Risk-based Testing**

I. Introduction

FUNCTIONAL correctness is crucial for the success and acceptance of a software product. A solid testing process is imperative to uncover defects before they are deployed in the field. Since resources are limited, especially for large software systems, it is important that test efforts are allocated such that the most critical defects are detected as soon as possible. This requires an estimation of which parts of the system are expected to be particularly defect-prone. The research field of defect prediction aims at revealing faulty code, for example, through static program analysis, possibly enhanced by heuristic search or machine learning [1]. Even though a large variety of studies has been conducted in this area, the results are often not generalizable [2], and the approaches perform poorly in real-world settings [3], [4]. As a consequence, they are rarely applied in practice [5], [6], with notable exceptions, though [7].

Addressing the notorious issues of defect prediction of our partners in industry (in particular, Munich Re and LV 1871), we strive for an approach that is viable in practice and matched the needs of our industry partners: a *prioritization* of *test gaps* by their *risk*. A *test gap* is a method, function, or module that has been modified during a specific period of time (e.g., start of last development phase or iteration) and has not been executed in its most recent version during testing (e.g., automated unit test or manual acceptance test). Intuitively, defects are introduced by code changes, and defects cannot be detected if they were not tested. In this vein, the literature suggests that modified code tends to be more defect-prone [8]–[11]. For example, Eder et al. found in an industrial case study that (1) despite a structured testing process, approximately half of the changes went into production untested, and (2) that untested changes contained up to five times more defects than other parts of the system. This clearly emphasizes the value of test gap analysis in the testing process. For this reason, test gaps are taken into consideration by test management to decide whether testing is completed.[1]

Problem Statement: The number of test gaps that need to be investigated by test management and quality assurance depends on many parameters, especially on the number of code changes and the depth of testing. In practice, when test management assesses test gaps as part of test-end criteria evaluation, there are typically dozens, hundreds, or even thousands of test gaps [12] which were not covered by automated or manual test runs. More importantly, the risk of test gaps may vary greatly. Risky test gaps such as logic modifications or data manipulations might be hidden among many safe changes such as refactorings [15]. Obtaining an overview about test gaps and their risk requires significant effort, and the results may be subjective. Furthermore, in the context of large industrial software systems with changing development teams, it can be hard for individuals to have sufficient knowledge about the system to reliably assess the criticality of changes in the entire code base. Clearly, an *automatic prioritization of test gaps* would be most helpful to reduce the time necessary for the

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 was with the Saarbrücken Graduate School of Computer Science, Germany. R. Haas, M. Sailer and E. Juergens were with CQSE GmbH, Germany. M. Joblin and S. Apel were with Saarland University, Saarland Informatics Campus, Germany. (Corresponding author: R. Haas.)

Manuscript received October 18, 2024; to be revised in February 2025.

[1]There are several test gap analysis tools available, for example, Teamscale [12], [13] and Sealights [14].

manual inspection and the risk of missing critical ones, which is also confirmed by our industrial partners.

Research Gap: There is a lot of research in the field of defect prediction, which aims for the identification of defective code [5], [16]–[18]. For our industrial research setting, unfortunately, the costs of applying state-of-the-art approaches outweigh the potential benefits—a notorious challenge for the application of defect prediction in practice [4], [6]. Moreover, state-of-the-art approaches typically do not consider prior testing efforts that are focussed by test gap analysis [1], [7], [18]. Another related field, *test case prioritization*, aims at finding failures as quickly as possible by prioritizing tests by their failure revelation probability [19]. Test case prioritization addresses a different problem, though, since it orders test cases, whereas test gap analysis reveals code which has not been tested by existing test cases. In particular, our industrial partners aim for a more general form of risk mitigation: while the probability of introducing a defect is a key risk factor [20], [21] the potential damage caused by a defect is an additional risk factor that needs to be taken into account. That is, core functionality needs to be tested particularly well because potential defects can cause major damage, such as degradation of core business processes—even though the probability of introducing a defect may be relatively low. Since there is only little related work on prioritization or risk estimation of test gaps in industrial practice, it is still unclear what makes a test gap more risky than others. Hence, we seek to evaluate the feasibility of a simple prioritization approach that fits the setting and requirements of our industry partners and suggest improvements for future deployment.

Approach: Based on key risk criteria identified from industrial developer experience and the defect prediction literature, we propose our *score-based* approach, called TESTGAPRADAR, to automatically derive a risk-based prioritization of a set of test gaps. This way, a large list of open test gaps can be sorted and, for example, test management can analyze the riskiest test gaps first. To evaluate our approach, we conducted a multi-method study: We compare the prioritization results of our approach with manual test gap risk assessments that have been made by experts in eight real-world industrial projects. For this purpose, we use historic real-world risk assessments from Munich Re and LV 1871, two large companies within the financial domain, where quality assurance experts continuously assessed eight well-established and maintained software systems. We conducted semi-structured interviews with these experts to better understand deviations between our automatic ranking and their manual assessment. Overall, we found that our approach yields a test gap ranking that is shown to be correlated with risk (i.e., higher rank corresponds to more risk) and that the approach can achieve human (domain expert) level performance. Interestingly, quality engineers reported that insights from TESTGAPRADAR allowed them to recognize where their prior assessment missed risks, especially for central code (i.e., code that is central in the program's dependency structure [22]–[24]). The quality engineers of our industrial partners acknowledged the significance and relevance of our test gap risk criteria and underline that TESTGAPRADAR would help them in their daily work to identify risky test gaps more efficiently.

Contributions: In summary, we contribute the following:

- *Automated Prioritization of Test Gaps.* We introduce TESTGAPRADAR, an automated score-based approach that prioritizes test gaps by estimated risk supporting industrial development teams—including test management and quality assurance roles—in gaining a quick overview about the riskiest gaps.
- *Empirical Study.* We conducted a field study of TESTGAPRADAR on thirty-one historical test gap reviews of open test gaps for eight industrial software systems providing insights into the applicability of risk criteria and process in our industrial setting.
- *Quality Assurance Expert Survey.* We conducted eight semi-structured interviews with six quality assurance experts that authored the test gap reviews of our industrial partners, showing that the automatic ranking of TESTGAPRADAR is on par with expert rankings, and in some cases, even outperforms the expert ranking.

II. BACKGROUND AND RELATED WORK

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

A. Test Gap Analysis

Test gap analysis is one possible answer to the question of test resource allocation. It reveals untested changes in the code, which are known to be particularly defect-prone [11] and should receive special attention in the testing process [12]. The entity of interest for test gap analysis is often code at the *function or method* level (see Sec. II-B). Figure 1 shows a typical test gap analysis result for all code changes within a release cycle of several months of a large software system of our partner Munich Re (before they aligned their testing activities along untested code changes). It visualizes the entire code base as a tree map and depicts test gaps in orange and red colors; in this case, there are *thousands* of test gaps. Due to the enormous number of code changes and testing activities, it is challenging in practice to identify the most risky test gaps, which is needed to direct the limited testing resources to the mitigation of the largest risks.

Test gap analysis is always performed for a certain timeframe $[b, t]$, defined by a baseline b and an end t. A function is considered as changed if, at least, one line of code has been altered within $[b, t]$. Note that some behavior-preserving changes, such as the renaming of a variable, are ignored. Some entities that can be considered too trivial to test, for example, simple variable access functions (getter and setter), can also be excluded [25].

A function is considered as tested if, at least, one line of the function has been executed by a test after the latest change in $[b, t]$. This is not meant as a test adequacy criterion, that is, a function that is tested by means of test gap analysis is not necessarily tested sufficiently. But, test gap analysis is designed to highlight functions that have not

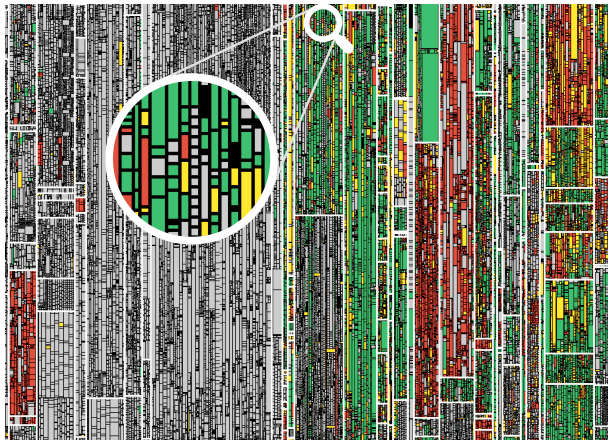

Fig. 1. Test gap analysis results for an industrial business information system of our partner Munich Re for all code changes within a release cycle of several months. The tree map visualizes the hierarchical components, classes, and functions of a software system. The size of rectangles corresponds to size in lines of code; the color indicates unchanged code, tested changes, untested modifications, and untested new functions, where the latter two are test gaps. In this example, there are thousands of test gaps.

been tested at all. This simple notion of function coverage is capable of providing a valuable high-level overview of *untested* changes [12]. More fine-grained coverage criteria, for example, on the statement level, do not provide sufficient benefits to outweigh their cost [12]. They may even be counter-productive as they produce more complicated results, which require more effort to interpret, especially for large software systems with huge code churn. Note that the claim is not that unchanged and tested parts of the system do not contain any errors, but that the chances are higher to detect defects in untested changes than in other parts of the system.

B. Related Work

There is only little related work that is specific to test gap prioritization. In addition, we discuss related work from the broader field of software defect prediction.

1) Test Gap Prioritization: Sailer [26] conducted a structured online interview to shed light on developers' criteria for assessing the risk of test gaps. In their study, they randomly selected a subset of test gaps from a six-week interval of the developers' application and asked the developers to assess and reason about the risk of test gaps. These rationales provide valuable insights for our work into what makes a test gap appear risky. The most-mentioned reasons are *change complexity*, *centrality* for high risk, and *refactoring* as rationale for lower risk. In contrast to Sailer who evaluates their approach on a single software system, we investigate the transferability of risk criteria and their applicability for test gap prioritization to other large and independent software systems from our industry partners.

Brandt et al. [27] used a fuzzer to generate partial tests and investigated whether developers from Mozilla would extend those to functional tests. They found that developers do not consider all test gaps test-worthy. To address this, they implemented a filter function to only generate partial tests for relevant test gaps. The filter function excludes test gaps that are single-line or are early-return. They found that developers consider test gaps irrelevant if the tested code is unlikely to be reached, deemed bug-free, or already covered by other tests. By means of the code centrality criterion, we also prioritize code down that is unlikely to be reached, which is similar to Brandt et al.'s approach. They focus on helping developers close test gaps, while we focus on helping test managers and quality assurance teams identify the most important ones. Their filter function employs a simple prioritization strategy. We present a more comprehensive approach for prioritizing test gaps, focusing on different roles in the software development process.

2) Software Defect Prediction: Test gaps are known to have a higher probability of defect than unchanged code [11], so, the field of software defect prediction [5], [16], [18] is related to our work, with notable differences, though. While a test gap could imply a defect, a defect is not necessarily a test gap. For our prioritization of test gaps, we are focusing on risk, namely, magnitude of impact and probability of event. In contrast, defects may be classified by severity levels [28], but these do not match test gap risk.

The respective models of defect prediction approaches are based on manifold metrics, for instance, source code complexity [29], code smells [30], history of changes [31], commit messages [32] and defects [33], organizational [34], process [35], or developer-centered metrics [36], ticket information [37], [38], or static analysis results [39]. While our prioritization approach relies on some of these metrics, the focus of our work is different since our industry partners align their testing process with untested code changes, for which we aim at a risk-based prioritization.

Like with test gap analysis, it is an established best practice to use function-level defect prediction, which shows better performance than relying on coarse-grained units such as files or modules [4], [40]: Despite a great number of studies in this area, function-level defect prediction is still unsolved as it provides low precision for cross-project classifiers and, when evaluated under realistic circumstances, existing approaches do not significantly outperform a random classifier [3], [4]. There are also more fine-grained approaches of defect prediction, for example, on the line-level [41], [42]. These cutting-edge approaches are still experimental and therefore—from our industrial partners' perspective—not mature enough for implementation in real-world development processes, yet. As of now, our industrial partners need more actionable approaches such as test gap analysis and prioritization of test gaps to direct their testing efforts.

III. TestGapRadar: A Score-based Approach

Test gap analysis yields an unsorted set of untested modified code units (functions, in our case). The goal of TestGapRadar is to rank this set of test gaps by their estimated risk. In what follows, we explain the selection criteria for the metrics used for ranking test gaps. Furthermore, we provide details on the metrics and their calculation and insights into the normalization of metric values. Finally, we describe how a risk score is computed, which is used to rank test gaps

amongst each other. In this section, we focus on our approach in a general form; implementation details like weights and the choice of parameters that we used for our evaluation are outlined in Section IV-D.

A. Selection Criteria for Metrics

The basis for ranking is a risk score, which is computed for each test gap from a combination of metrics (see also Sec. III-B). To simplify the setup for our industrial study subjects (see also Sec. IV-B) and allow for comparison of results, we aim for a lightweight, uniform approach for all study subjects. To this end, we discuss in this work a simplistic approach to investigate to which extent a simple approach like ours is able to help practitioners in identifying risky test gaps more effectively. We selected product and process metrics that were also used by related work (see also Sec. II-B) and which were feasible to obtain for all of our study subjects. We had to decide against criteria which did not meet the requirements of our industry partners. Specifically, a broad spectrum of technologies and processes, and diverse set of social and legal requirements needed to be fulfilled. For example, ABAP—a programming language used by several of our industry partners' SAP systems—comes with the limitation that there are no commit messages available as they are known from popular version control systems such as git. Furthermore, defect information is stored heterogeneously, that is, in different bug tracking systems using different bug reporting schemes, impeding its structured analysis. Lastly, developer-centered metrics might not always meet the compliance requirements of our industry partners, so they could not be taken into consideration for our work. For instance, the separation of responsibilities between internal and external employees is mandated by European regulation [43].

B. Overview of Selected Metrics

Table I provides an overview of metrics that TESTGAPRADAR uses to estimate test gap risk. These product and process metrics were available at our industrial partners, while other data could not be obtained. They include different risk factors, that is, code criticality and complexity and static code analysis results. In the following, we discuss these risk factors, the corresponding metrics, and their computation.

1) Risk Factor: Code Criticality: We implement two metrics for code criticality: code centrality and changed files.

Code Centrality: Sailer [26] found in their study that one of the most often mentioned reasons for critical test gaps, that is, gaps that need to be closed by testing, is code centrality. This matches our notion of test gap risk since defects in central code (e.g., core functionality) potentially cause great harm, and therefore test gaps in central code are considered more risky. For the purpose of test gap prioritization, TESTGAPRADAR includes a metric for the *centrality of the function*. To compute the centrality of the function at time t, we rely on static analysis of dependencies between sources, as suggested in the literature [22]–[24]. We apply the PageRank algorithm [44] on the dependency graph of a software system to rank the nodes by their relevance (i.e., centrality). The dependency graph is traversed by either following a link or randomly jumping to another node. We used the random jump probability of 0.0001, as suggested by Steidl et al. [22]. Additionally, we take inverted edges into account to reflect that a function may not only be important if many other functions depend on it, but also if it depends on many important functions. For this paper, we set the weight of an inverted edge to $\frac{1}{3}$ as compromise between the suggested values of $\frac{1}{2}$ and $\frac{1}{4}$ [23].

TABLE I
OVERVIEW OF METRICS USED BY TESTGAPRADAR TO ESTIMATE TEST GAP RISK

Risk Factor	Metric	Short
Code Criticality	Code centrality	CEN
	Changed functions	CHF
Complexity	Length of reference function	LEN
	Changed lines of code	CLI
	Complexity of reference function	COM
	Complexity change	COC
Static Code Analysis Results	Added normal findings	ANF
	Unresolved normal findings	UNF
	Removed normal findings	RNF
	Added critical findings	ACF
	Unresolved critical findings	UCF
	Removed critical findings	RCF

Changed Functions: In the context of our industrial partners, large change sets, which affect many files, often occur in system- or component-wide refactorings of the source code, which do not contain functional changes. These refactorings introduce less defects than smaller change sets and thus, the defect density of large change sets is typically smaller than for small change sets. This is in line with the literature in which change sets with many changed files have been found to introduce less defects [31]. Since we are working on the more fine-granular level function level, we count the number of *changed functions* per change set. We decided against using more complex metrics as most of these have been shown to have a high correlation with lines of code [45], [46].

We use a simple heuristic to implement this metric. The idea is, the more functions are modified within a change set (e.g., issue, ticket, change request), the more likely it is a less risky change, for example, a refactoring. For the change set that contains the test gap, we calculate the function churn c_i, that is, the number of modified functions. The metric grows, damped by the power function, up to a function churn threshold c_t. The metric is bound by 1 and calculated by

$$v_{\text{ref}} = \min\left(\left(\frac{c_i}{c_t}\right)^4, 1\right)$$

For the size of the change set, we define a custom normalization function to model the influence on test gap risk. We do not expect a linear correlation of size of the change set and test gap risk. For instance, we consider the influence of a medium and a big change set on the test gap risk as similar, but the difference to a small change set is notable. Note that, for our multi-method study (see Sec. IV), we manually validated that this heuristic has assigned high changed functions metric values only to test gaps arising from a refactoring activity.

2) Risk Factor: Complexity: We implement four metrics to model the risk factor complexity: *length of reference function* and *changed lines of code* serve as metrics for code complexity, while test complexity is measured by *complexity of reference function* and *complexity change*. The *reference function* is the version of the function before it became a test gap, that is, either the version at the baseline b or, if the function has been tested after the baseline, the last-tested version. If the function has been added after the baseline, the reference state is the empty function $\emptyset$. We call the function at version t the *end state* of the function.

Code Complexity: In accordance with several defect prediction approaches [5], we use different metrics for the complexity. The first two metrics refer to code complexity: *length of the reference function* and the *number of changed lines* in the function between reference and end state. We hypothesize that long and complex functions are harder to understand and change and, hence, more defect-prone. In addition, long and complex changes are more defect-prone than small and simple ones [47]. The length of the reference function is the number of source lines of code, that is, excluding comments.

There are multiple ways of computing a metric value for number of changed lines. Code changes are typically displayed as unified diff, that is, the added and deleted lines between both versions. A change to one line is represented as a combination of one added and one deleted line. The same, however, is true for a change that deleted one line and added another, that is, two changed lines. Prior work used the sum of deleted and added lines as value for this metric [33]. In contrast, we doubled the weight of added lines to take our industrial partner's experience into account, which shows that adding lines is more defect-prone then deleting lines (see also Sec. IV-D).

Test Complexity: To obtain an indication of how many tests might be needed to test a function [48], we use *cyclomatic complexity* as defined by McCabe [49]. The *complexity change* is the difference in cyclomatic complexity of the function between end state and reference state. In particular, this means that the value can be negative if the change reduced the function's complexity.

3) Risk Factor: Static Code Analysis Results: We use the number of static code analysis results (i.e., findings) as a proxy to gauge the diligence with which code changes were made. Our industrial partners rely on Teamscale [50] for static code analysis. Teamscale differentiates between new findings and unresolved findings in modified code, both of which are expected to be fixed in the contexts of our industrial partners who have a quality control process in place [51]. Findings are categorized into critical quality deficits, such as bug patterns, and normal quality deficits, such as incomplete documentation. We distinguish between six different finding metrics (the third character of metric abbreviations from this risk factor in Table I is `F`), that is, the cross product of two severity levels and three finding states. First, we distinguish between the two severity levels critical (the first character is `C`) and normal (`N`) because we consider critical findings to be more risky than normal ones. Second, we separate the finding states added (the third character is `A`), unresolved findings in modified code (`U`), and removed findings (`R`). Hasty code changes that have not been carefully reviewed (e.g., by a static analysis tool) can introduce new defects, increasing the risk of a test gap. This is the case for code changes that add *new findings*, either critical or normal ones, or that *do not remove existing findings*, where the latter ones are expected by our industry partners to be less risky because their quality control process ensures that unresolved findings are less relevant. In contrast, changes that *remove findings* improve code quality and therefore may be less risky. For each test gap, we count the number of findings that were affected by the code changes in that test gap.

C. Normalization of Metric Values

The selected metrics have different value ranges. To balance their influence on a score-based test gap prioritization, it is necessary to normalize them. We choose the value range of $[0, 1]$ for all metrics except COC. COC is capable of taking negative values and, as such, is mapped to the interval $[-1, 1]$. For each metric, the set of values S is normalized with respect to its maximum and minimum value. The normalization aims at a relative prioritization of test gaps inside one set. However, it is not possible to compare the metric values between different sets of test gaps, as their normalization is based on different maximum and minimum values. In particular, this means that TESTGAPRADAR does not determine the risk of single test gaps. Instead, we aim for an approach that *ranks a set of test gaps based on their estimated risk*.

D. Computation of Risk Score

In the final step, the normalized values of all metrics are combined into one risk score for every test gap. This is used as basis for prioritization. Every test gap m has a set of normalized metric values V_m. The set W contains the corresponding metric weights (see also Sec. IV-D for details on W we used in our multi-method study). Each metric has exactly one weight, which is the same for all test gaps. Thus, with the number of metrics $k = |W| = |V_m|$ the risk score r_m for a test gap m is

$$r_m = \sum_{i=1}^{k} V_m[i] \cdot W[i].$$

IV. MULTI-METHOD STUDY

We conduct a multi-method study to evaluate the practical applicability of TESTGAPRADAR, our score-based approach for risk-based prioritization of test gaps presented in Section III, in an industrial setting. Initially, in a field study[2], we compare the risk estimations with test gap reviews of eight software systems across two industrial partners, and we use this data set to compare our approach with a random ranking strategy as baseline. Subsequently, through semi-structured interviews,

[2]Following Stol and Fitzgerald [52], a *field study* "refers to any research conducted in a specific, real-world setting to study a specific software engineering phenomenon".

we discuss our approach with the industrial quality engineers who were involved in the test gap reviews. We follow the guidelines of Jedlitschka et al. [53] to report on our research.

A. Research Questions

RQ$_1$: How does TESTGAPRADAR *perform as compared to risk assessments of quality assurance experts?* In RQ$_1$, we aim at comparing the risk assessments of TESTGAPRADAR with historical risk assessments provided by two industry partners in the form of test gap reviews. In these, quality engineers analyze open test gaps for risky gaps and report them in their review. We analyze whether test gaps, which were labelled as risky by quality engineers, are highly ranked by TESTGAPRADAR.

RQ$_2$: Which metrics of the risk score are most important and are the weights robust? Our approach calculates a risk score that is influenced by numerous metrics. We study the individual metric importances to learn which of the metrics have the highest impact on detecting test gap risks. For this purpose, we investigate which metrics of TESTGAPRADAR are decisive to identify risky test gaps from the historical risk assessments. To shed light on the robustness and reliability of TESTGAPRADAR, we perform a sensitivity analysis and a scenario analysis.

RQ$_3$: How much better is TESTGAPRADAR *compared to a random ranking strategy?* We compare our approach to a random ranking strategy as baseline, to validate whether a sophisticated approach like ours pays off by better test gap prioritization results. That is, test gaps labelled as risky in test gap reviews are assigned a higher rank, with the ranking indicating a higher estimated level of risk.

RQ$_4$: Do quality engineers find that the test gap prioritization process can support them in their day-to-day work and if so, how? We conducted semi-structured interviews with the authors of the original test gap reviews, that is, six professional quality engineers of our industry partners, to discuss the practical value of our work. First, we investigate whether they agree with the metrics used for the risk-based prioritization of test gaps. Second, we explore the reasons and practical implications behind deviations observed in RQ$_1$. Third, we query whether and in which ways our approach could support them in their day-to-day work.

B. Industrial Study Subjects

For the purpose of our evaluation, we have selected eight industrial study subjects from our industrial partners. An overview of all study subjects of our multi-method study is given in Table II; the provided contextual data meets the criteria by Hall et al. [2]. All study subjects are industrial[3], closed-source systems which have been in successful use for many years and are still actively developed and maintained. The subjects are internally used software systems implementing core business processes or products, and are of mediocre (100 K LOC) to large size (1,900 K LOC). Their implementation relies on different technologies, all of which are supported by our language-agnostic approach. For all study subjects, a well-established issue tracking and testing process is in place. Some subjects adopt automated testing in a CI environment, others focus on manual testing in dedicated testing environments, and some adopt both approaches. All subjects stem from two large, independent players in the finance and insurance domain from Germany, which is strictly regulated by the European Union [54]:

Munich Re is one of the world's leading providers of reinsurance, primary insurance and insurance related risk solutions. It has about 43,000 employees, and a revenue of more than 52.9 billion Euro. Conscious of the great responsibility for software quality, they have a standardized development process, which includes test gap analysis, but without prioritization of test gaps. A dedicated team is reviewing code and test activities of all software systems in the portfolio manually, resulting in the monthly assessment reporting. For our study, we used the test gap review data from five systems within Munich Re (no. 1–5).

LV 1871 is a German specialist for life and pension insurance. It has ca. 500 employees, and generates 7 billion Euro in revenue. Emphasizing code quality, the company works with an external team of quality engineers for code retrospectives and test gap analysis. For this study, we used test gap reviews from three of their software systems (no. 6–8).

The development processes of both industry partners include quality control metrics, including external, manual reviews of test gap analysis results (see also Popeea-Simeth et al. [55]). In fact, we chose the systems of our industrial partners as study subjects because external quality assurance experts conduct handcrafted *test gap reviews* that assess test gaps based on risk. Our industrial partners implement test gap reviews for many years already, so that we can use this valuable historic information as reference data in our study. For our 8 subject systems, we use a series of up to 7 test gap reviews from 2023 (see also Table II), pointing to 181 risky test gaps (out of a total of 2,039 test gaps). We evaluate whether test gaps that were identified as risky are ranked high by our score-based approach. For transparency, we note that the external quality assurance experts work for the same company as some authors. No author was involved as interviewee in our semi-structured interviews, though.

TABLE II
OVERVIEW OF STUDY SUBJECTS

					# Test Gaps	
Company	Subject	LOC	Lang.	# Reviews	Risky	Total
Munich Re	1	1,600 K	C#	5	59	77
	2	140 K	C#	7	29	161
	3	370 K	ABAP	3	21	29
	4	560 K	ABAP	4	9	32
	5	1,900 K	ABAP	4	26	53
LV 1871	6	310 K	Java	3	5	622
	7	100 K	Java	3	28	1,052
	8	150 K	Java	2	4	13

[3]The names of the individual software systems have been anonymized on request of the providing industry partners.

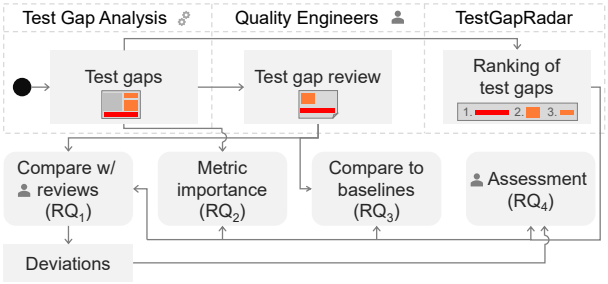

Fig. 2. Study data used to answer our research questions

2023-12: 58 test gaps. 39.2% of new or changed functions appear untested. Some test gaps appear to be of minor importance, but there are some relevant ones as well, for example:

- Function `foo` in class `A` [Link1]
- Function `bar` in class `B` [Link2]

Fig. 3. Example for a test gap review of testing activities in December 2023 (highlighting indicates existence of risky gaps). The two anonymized functions `foo` and `bar` are labelled as *risky* by the quality engineers that authored the review.

C. Study Design & Operationalization

We apply multiple methods to answer our research questions of Section IV-A. Figure 2 provides an overview of the study data we used to answer the research questions. In the following, we provide an overview about our study data, that is, historical test gap reviews, and explicate the study design and operationalization for our research questions.

a) Historical Test Gap Reviews: We test the performance of TESTGAPRADAR on study subjects from our industrial partners. We use existing test gap reviews T as reference data for our study (see Sec. IV-B). A test gap review refers to all open test gaps since the review baseline b (cf. Sec. II-B1). Typically, it spans over a period of 1–3 months, depending on the amount of development activity. In all subject systems, test gap reviews are conducted regularly by an external party, to ensure neutrality. A review states the number of open test gaps and the proportion of open vs. closed test gaps. In addition, it lists test gaps that the reviewer considers *risky* and, consequently, which they recommend being closed.

DEFINITION "RISKINESS OF TEST GAPS". A test gap is classified as *risky* if it has been identified as pertinent in a test gap review. Pertinent test gaps present a notable risk of introducing defects and are advised by quality engineers to be resolved by the relevant development and test teams. Conversely, a test gap is categorized as *less risky* if it has not been mentioned in a test gap review or if it has been referenced in a review where the quality engineer indicated a low risk of defect introduction.

An exemplary test gap review of the testing activities in December 2023 from one of our study subjects is given in Figure 3. Note that test gap reviews point to test gaps that are considered (most) risky. So, this list may not include all test gaps, and less risky test gaps are not mentioned.

b) RQ$_1$: Comparison with Manual Assessments: To answer RQ$_1$, we adopt two methods. First, we conduct a correlation analysis between the experts' test gap risk assessments from test gap reviews T and TESTGAPRADAR's risk scores R. We examine the correlation using Kendall's τ and the associated p value [56] (computed using SciPy [57]). Kendall's τ is defined for two rankings x and y with P concordant pairs, Q discordant pairs, and T ties only in x and U ties only in y:

$$\tau = \frac{P - Q}{\sqrt{(P + Q + T) \cdot (P + Q + U)}}$$

Second, we investigate the risk score rankings of risky and less risky test gaps. For this purpose, we calculate an agreement value $v \in [0, 1]$, where values closer to 0 stand for a better ranking agreement (that is, risky gaps are ranked before less risky gaps). As basis for v, we sum up the ranks $\text{rank}(t)$ of risky test gaps $t \in G_r \subset G$, where G are all test gaps of a test gap review, and divide them by the number of test gaps $|G|$ and the number of risky test gaps $|G_r|$ to obtain v':

$$v' = \frac{\sum_{t \in G_r} \text{rank}(t)}{|G| \cdot |G_r|}$$

The best possible minimum $\min(v')$ is:

$$\min(v') = \frac{\sum_{i=1}^{|G_r|} i}{|G| \cdot |G_r|}$$

All risky test gaps are ranked above all other test gaps. For example, for $|G_r| = 3$ and $|G| = 7$, we obtain $\min(v') = (1/7 + 2/7 + 3/7)/3 = 0.29$

The worst possible agreement value $\max(v')$ is:

$$\max(v') = \frac{\sum_{i=|G|-|G_r|+1}^{|G|} i}{|G_r| \cdot |G|}$$

All risky test gaps are ranked below all other test gaps. For example, for $|G_r| = 3$ and $|G| = 7$, we obtain $\max(v') = (5/7 + 6/7 + 7/7)/3 = 0.86$.

To obtain the agreement value v, v' is min–max scaled to $v \in [0, 1]$ (with a mean of 0.5):

$$v = \frac{v' - \min(v')}{\max(v') - \min(v')}$$

We report a false-low rate of test gap review rankings. For this purpose, we consider the fraction of test gap reviews exhibiting a poor agreement value ($v \geq 0.5$) against all test gap reviews.

Lastly, we use the Mann-Whitney U test to test the null hypothesis that the ranks of test gaps deemed risky in the test gap reviews determined by their risk score ($\in R$) and the same ranks of less risky test gaps stem from the same distribution.

c) RQ_2: Metric Importance: We implement a multifactorial ANOVA (analysis of variances) to answer RQ_2. That is, we employ ANOVA to assess the influence of the independent variables (i.e., the score metrics) on the dependent variable (the test gap risk assessments from T). The null hypothesis is that there is no relation between an independent variable and the dependent variable. If the p value is below $\alpha = 0.05$, we reject the null hypothesis and assume that there is a relationship between the independent and the dependent variable. To this end, we report the corresponding F and p values. In a post-hoc analysis employing linear regression, we investigate the strength of these relationships. In particular, we report the p values, the coefficient values, and R^2 for the regression models. Additionally, we report results of the correlation analysis in the form of a correlation heatmap.

We report global sensitivity indices as suggested by Sobol´ [58], since they are designed for nonlinear models (like ours) which allow for decomposition of ranking contributions from individual parameters. We implement the sensitivity analysis using SALib [59] and use $N \times (2D + 2)$ model evaluations, where $N = 2,039$ is the number of samples and $D = 12$ is the number of metrics. The first-order sensitivity indices S_{1_i} represent the effect of each metric on the risk score variance when all other factors remain constant. The total-order sensitivity indices S_i^{tot} capture both the individual effects and the interactions with other metrics. For the scenario analysis, we iteratively vary weights of parameters with the highest and lowest impact on the risk score and report the scenario performance which we measure by the median relative test gap ranking of risky test gaps (see also RQ_1).

d) RQ_3: Comparison to Random Baseline: To answer RQ_3, we compare the ranking performance of our score-based approach TESTGAPRADAR with a *random strategy*: The random strategy simulates 1,000 test gap analysis sessions for all test gap reviews without any indication about the risk available. That is, we assign all test gaps from the test gap reviews a random risk score in $[0, 1]$ and rank them by this random score. From the 1,000 simulations, we calculate the average rank of risky gaps and the ranking variance. To assess the ranking performance, we consider the relative median ranks $\tilde{R} \in [0, 1]$ of test gaps deemed risky and their variance $\text{var}(R)$. Additionally, we use a Mann-Whitney U test to test the null hypothesis that test gaps deemed risky in the test gap reviews ($\in T$), ranked by the risk score (R) of TESTGAPRADAR and the ranks determined by the baseline stem from the same distribution.

e) RQ_4: Expert Assessment: We answer RQ_4 based on semi-structured interviews with the six industrial quality engineers who conducted the original test gap reviews used in the earlier research questions. Details of the interview questions can be found on our supplementary website (see also Sec. IX). All interviewees are experts in the field of software quality. Their professional experience in coding, software testing, and quality consulting activities ranges from 4 to 20 years. All of them have a Master's degree in software engineering, two of them even have a PhD in software engineering. They are experts in test gap analysis tools and have been using them in their daily work for years.

For each study subject, we chose the test gap review with the *lowest* agreement between the review and the risk-score-based ordering of test gaps for our interview. We manually ensured that the set of chosen test gaps is as diverse regarding associated change types and risks as possible. To foster a lively conversation allowing for deep dives into the data, we conducted for each study subject a joint semi-structured interview with author and reviewer of the test gap review between April and June 2024. Our primary goals were to shed light on the experts' reasoning behind risk assessments, to gain insights into reasons for deviations between R and D, and to collect feedback on practical applicability of our approach.

Each semi-structured interview consisted of three parts: First, we asked the participants to construct a pairwise comparison of 3–4 test gaps of the assessment based on their subjective risk. For this task we selected test gaps where the professional assessment did not match the automatic ranking. Second, we discussed our test gap prioritization, in general, and specifically the TESTGAPRADAR's generated ranking for the 3–4 test gaps. Third, we ask about the expert's background and their feedback on our research. The interview sessions took ca. 30 minutes, each.

We applied qualitative content analysis methods [60] to systematically analyze the data from the semi-structured interviews. This involved employing the QCAmap tool by Mayring et al. [60] for qualitative content analysis and applying inductive techniques for data categorization.

D. Implementation and Calibration

We have implemented TESTGAPRADAR and used a data-driven approach to tune the weights of the risk score function (see also Sec. III-D) For this, we used preliminary (training) data of former test gap reviews (pre-2023) from our industrial partner Munich Re that we obtained in the initiation phase of our research effort. In this pilot study, we fine-tuned the weights of our approach so that risky test gaps from pre-2023 test gap reviews are ranked highly. We considered our sample size too small for automated fine-tuning mechanisms, so we used manual fine-tuning instead. Manual fine-tuning helps us provide rationales for the weights, which increases trust in the tool and makes the prioritization results more understandable, which is key for broad adoption in practice. To mitigate the risks of subjectivity and potential biases in weight selection, we investigate weight robustness and reliability as part of RQ_2, and provide guidelines for practitioners to implement our approach in their testing process (see Sec. V). Generally, we applied the risk score function consistently for all study subjects. We avoided overfitting by using separate training and test data sets.

Initially, all weights were assigned a default value of 1. The rationale behind weight selection, ensuring explainability of the prioritization, includes:

1) setting positive weights as defaults, while indicators of improved code quality (e.g., readability) are allocated negative weights (i.e., CHF, RCF, RNF);
2) assigning higher weights to factors frequently cited by developers as critical in related studies on test gap

prioritization [26] receive a higher weight (i.e., CEN, CLI, COC);
3) prioritizing code change metrics over reference function metrics motivated by the notion that added complexity signifies greater risk than existing complexity (i.e., CLI, COC);
4) providing greater weight to metrics capturing critical findings than those for normal findings, with new findings deemed riskier than existing ones (i.e., ACF, ANF, UCF, and RCF).

The final weight calibration is outlined in Table III.

TABLE III
OVERVIEW OF METRIC WEIGHTS USED FOR TESTGAPRADAR IN THE EVALUATION

Metric	Short	Weight
Code centrality	CEN	2
Changed functions	CHF	-1
Length of reference function	LEN	1
Changed lines of code	CLI	2
Complexity of reference function	COM	1
Complexity change	COC	2
Added normal findings	ANF	2
Unresolved normal findings	UNF	1
Removed normal findings	RNF	-1
Added critical findings	ACF	4
Unresolved critical findings	UCF	2
Removed critical findings	RCF	-2

The changed functions metric has a parameter, t, which refers to the size of a change set so that it is considered as less risky. In preliminary experiments, we found that $t = 100$ modified functions per change set are suitable to identify refactorings for our study subjects.

E. Results and Discussion

In what follows, for each research question, we present the results of our multi-method study and discuss them.

1) RQ1: Comparison with Manual Assessments: We use Kendall's τ to investigate whether the test gap assessments from test gap reviews and the normalized risk scores correlate. We find a small [61], negative monotonic correlation between T and R ($\tau = .29$, $n = 2039$, $p < .001$), meaning that risky test gaps from test gap reviews receive higher rankings from TESTGAPRADAR.

Figure 4 shows a kernel density plot of test gap ranks for the two populations of risky and less risky test gaps from the 31 test gap reviews from our eight industrial study subjects. The graph of risky test gaps is right skewed, while the graph of less risky test gaps is left skewed. That is, risky test gaps are ranked higher (i.e., better) by TESTGAPRADAR than less risky test gaps. The median ranking for risky test gaps is 0.27, while less risky test gaps are ranked lower (median = 0.73); the average ranking is 0.5. In total, TESTGAPRADAR ranked risky test gaps for 3 out of 31 test gap reviews too low (i.e., "false-low", $v \geq 0.5$), so the false-low rate is below 10%. The distributions in the two groups differed significantly (Mann–Whitney U = 127.5, n_r = 181, n_{lr} = 1858, $p < 0.05$, less).

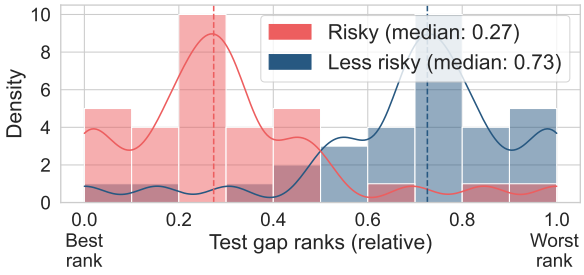

Fig. 4. Kernel density plot for agreement values (v) of test gaps labelled risky (red) or less risky (blue); from 31 test gap reviews of our 8 industrial study subjects

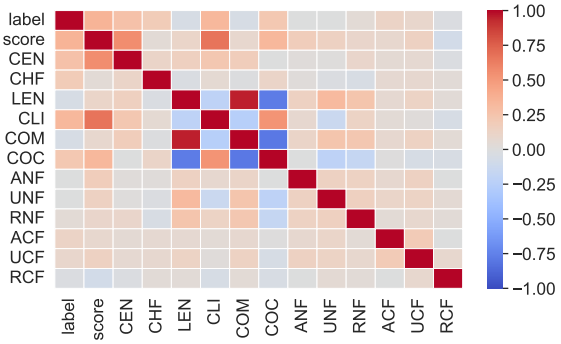

Fig. 5. Correlation heatmap of the assessment label, the risk score, and the score metrics

Overall, the accuracy of the results of our application compared to the expert assessments appears rather good: An average ranking of risky gaps on the 27th percentile and a low false-low rate below 10% shows that risky test gaps are ranked higher by TESTGAPRADAR than less risky test gaps. In the discussion of RQ4, we explore reasons for deviations and their implications for the practical use of TESTGAPRADAR.

SUMMARY RQ1. *Our approach achieves a good ranking performance: Risky test gaps are significantly more likely to be ranked higher by* TESTGAPRADAR *than non-risky test gaps.*

2) RQ2: Metric Importance: Figure 5 shows a correlation heatmap among the risk assessment labels ("risky" or "less risky" from the test gap reviews T), the risk score (determined by TESTGAPRADAR), and associated metrics (see also Table I). Noteworthy correlations include a moderate positive link between the risk score and CEN, CLI, and COC. LEN is moderately negatively correlated with COC but strongly correlated with COM. Additionally, COM has a moderate negative correlation with COC.

Table IV shows the results of ANOVA with F statistics and p value for the independent variables (metrics). There are seven variables that have a strong relation ($p < 0.05$) with the assessment label: CEN, LEN, CLI, COM, COC, UNF, and ACF. For these, we ran a post-hoc analysis with linear regression models; Table V shows the results. The p

TABLE IV
ANOVA TABLE WITH F STATISTICS AND p VALUE PER INDEPENDENT VARIABLE

Metric	Short	F	p
Code centrality	CEN	123	.000
Changed functions	CHF	1.95	.163
Length of reference function	LEN	39.7	.000
Changed lines of code	CLI	482	.000
Complexity of reference function	COM	12.0	.000
Complexity change	COC	65.6	.000
Added normal findings	ANF	.68	.408
Unresolved normal findings	UNF	15.0	.000
Removed normal findings	RNF	2.44	.118
Added critical findings	ACF	19.5	.000
Unresolved critical findings	UCF	.01	.936
Removed critical findings	RCF	.14	.706

value for all seven independent variables is well below the significance level $\alpha = 0.05$. This indicates a statistically significant relation between each of the variables and the manual risk assessment of test gap reviews. The largest coefficients have CLI, ACF, and CEN, so they appear to have a strong relationship with the manual risk assessment. Note that the R^2 values are relatively small across all models, suggesting limited explanatory power and the presence of other factors influencing the risk labels. This is expected in complex tasks such as test gap risk estimation for several reasons. Firstly, the subjectivity in risk label assignment by quality engineers is a factor. Secondly, background knowledge, domain expertise, and system familiarity influence the perceived risk of test gaps, elements not easily generalized or captured by our heuristics. Thirdly, there are diverse goals and risk factors in software testing, spanning functional, technical, economical, legal, and organizational aspects, which go beyond the scope of our work and exceed current software engineering methodologies.

There are three further noteworthy observations from the data. First, the correlation analysis suggests that longer function changes do not add as much complexity as new or small functions, particularly in the context of grown systems (which all of our study subjects are): Changes on existing—potentially grown—functions tend to be small compared to new metrics—added in new functions—that also add new, and on average, more complexity. Second, the complexity of reference function correlates strongly with function length and yields a modest F statistic in the ANOVA. Consequently, a simplification of our model might be to eliminate the COM metric. Third, from the variables that we found most important for a suitable risk score, that is, CLI, ACF and CEN, there is only a weak correlation between CLI and CEN. Hence, they all contribute significantly to the ranking performance of TESTGAPRADAR.

The first-order sensitivity indices of COC and ACF are the highest ($S_{1_i} = 0.28$) which confirms their significant individual influence on the risk score. The five metrics LEN, COM, ANF, RNF, and CHF have the lowest indices ($S_{1_i} \leq 0.02$), indicating that individually, they have minimal impact on the risk score when all other factors remain constant. Similarly, COC and ACF have the highest total-order sensitivity indices ($S_i^{\text{tot}} = 0.28$). The close similarity to their first-order indices

TABLE V
POST-HOC ANALYSIS WITH LINEAR REGRESSION

Metric	Short	p	coef	R^2
Code centrality	CEN	.000	.465	.136
Length of reference function	LEN	.000	.180	.014
Changed lines of code	CLI	.000	.741	.187
Complexity of reference function	COM	.000	.207	.018
Complexity change	COC	.000	.380	.075
Unresolved normal findings	UNF	.000	.153	.004
Added critical findings	ACF	.000	.713	.015

suggests these parameters have limited interaction effects with other parameters, reaffirming their role as primary contributors to output variance. The five metrics mentioned before also have the lowest total-order sensitivity indices ($S_i^{\text{tot}} \leq 0.02$), showing negligible differences between first-order and total-order indices. This supports the conclusion that their interactions with each other or with dominant metrics are minimal.

We shed light on the robustness and reliability of TESTGAPRADAR by running a scenario analysis. First, we reduce and increase the weights of the most influential metrics by a factor of 2. Then, the median relative test gap rank is 0.33, or 0.28, respectively, which indicates a deterioration in the ranking performance of TESTGAPRADAR. Second, we simplify our model by setting the weights of the five least influential metrics (with $S_{1_i} \leq 0.02$) to 0. We consider different scenarios in this case: a model where all five metrics receive a weight of 0, and five other models where for each only one of the metrics has a weight of 0. Our results show that the ranking performance improves when leaving out all five metrics (median relative rank of risky test gaps is 0.25). The ranking performance increases for all models where only one of the metrics is left out. In fact, the models which remove LEN or COM show the same ranking performance as the model where all five metrics are removed. Evaluating further scenarios, we realized that a model which leaves out the five least influential metrics and doubles the weight of ACF improves the median ranking performance to 0.23. That is, in development contexts similar to our study subjects, a model simplification with only 7 instead of 12 metrics can even achieve better prioritization than the approach implemented and calibrated for this study.

SUMMARY RQ_2. *Changed lines, complexity change, added critical findings, and code centrality are key metrics in our model to predict test gap risk. 5 metrics decrease the ranking performance on our sample set.*

3) RQ_3: Comparison to Random Baseline: Table VI shows the relative median ranking of the risky test gaps $\tilde{R}$, the ranking variance var(R), and the results of a U test of our score-based approach and the random ranking strategy. Our null hypothesis H_0 can be rejected (marked with ✗ in the table). The median ranking of TESTGAPRADAR outperforms the baseline with regard to the relative median ranking $\tilde{R}$ and shows a lower ranking variance var(R). That is, the score-based approach clearly outperforms the random baseline.

TABLE VI
BASELINE COMPARISON WITH A RANDOM BASELINE

Ranking Strategy	$\tilde{R}$	var(R)	U stat.	p	H_0
TESTGAPRADAR	.27	.05			
Random	.5	.06	235	.00	✗

SUMMARY RQ$_3$. TESTGAPRADAR *outperforms the baseline in ranking risky test gaps.*

4) RQ$_4$: Expert Assessment: In our semi-structured interviews, we observed that all six quality engineers (Q_{1-6}) found the metrics we used for test gap prioritization meaningful and representative of test gap risk. $Q_{1,2,5,6}$ saw a special value in the information about code centrality (CEN), since they usually do not have this information at hand when preparing test gap reviews, so TESTGAPRADAR can provide valuable extra information to the experts in test gap reviews. Additionally, Q_{1-5} explicitly agreed on our choice of putting lower weight on test gaps that refer to simple refactorings which we identify by the number of changed functions (CHF). All experts Q_{1-6} underlined the importance of complexity indicators for risk assessments, since code complexity makes it harder for developers to implement code changes correctly, thus requiring thorough testing. Also, $Q_{1,2,5}$ emphasize their commitment on code quality by verifying static analysis results, putting special focus on critical findings.

In two out of eight interviews, the quality engineers deviated from their original sorting of the interviews after learning about the prioritization of TESTGAPRADAR, so the tool prioritization outperformed the original expert sorting. In both cases, the information about code centrality was the decisive factor. For example, Q_1 stated *"I have to agree with code centrality of [this method], which looks pretty important to me. In this case, I'd vote for ranking it higher because it is more important than the other gaps"*. That is, TESTGAPRADAR was able to detect central test gaps that implied risky code changes, for which the experts retrospectively agreed that they would have considered code centrality if they had known about this factor beforehand. Conversely, when the quality engineers did not change their prioritization based on the reasoning of our approach, they justified their stance by citing several factors. These included the perceived higher risk associated with new functions compared to modified functions due to their lack of production history (2 cases). Additionally, they argued that the type of code (e.g., test code or generated code) could mitigate the risks associated with test gaps (2 cases). Furthermore, they expressed concerns about the extensive deletion of logic (1 case) and considered placeholder implementations (function stubs) to be less risky (1 case). In fact, they noted that an automated tool—while clearly helpful to them—can hardly capture all risk factors for test gaps, since risks can arise from other levels than source code, such as usage information, domain knowledge, or project context.

All quality engineers Q_{1-6} underline in the interviews that they see this tool as part of a semi-automated process, which still needs an expert in the loop. In this context, Q_4 praises that it can help to work in a *"much more structured way and identify relevant, risky test gaps much more quickly"*, and as Q_3 articulates, *"filtering out irrelevant gaps"*. Q_2 was quite enthusiastic and stated *"overall, the results here were exactly in line with my assessment, especially for the riskier items, which is a very exciting result"*.

SUMMARY RQ$_4$. *The experts consider* TESTGAPRADAR *valuable, providing them with additional information such as the centrality of test gaps, enhancing their daily work. Identifying high-risk gaps and filtering out low-risk ones improves their efficiency.*

V. GUIDELINES FOR PRACTITIONERS

High code churn and limited testing resources are nearly omnipresent circumstances in active industrial software development and contribute to large numbers of test gaps. TESTGAPRADAR addresses the problem of identifying the riskiest test gaps among potentially large sets of test gaps by sorting them according to estimated risk. We designed our approach to ease practical adoption and outline guidelines for practitioners in this section.

There are some requisites to implement our approach. First, the development process should require code changes to be successfully tested (e.g., in the definition of done). Second, test gap analysis needs to be established, that is, source code is under version control (e.g., using git) and all relevant testing environments are profiled.

To adopt our sorting and risk estimation approach, metrics and weights need to be chosen. For optimal ranking performance, we recommend to use the 7 most important metrics and adapt the weight of ACF, as we reported in the scenario analysis for RQ$_2$ of our multi-method study in Sec. IV-E2. A detailed configuration is depicted in Table VII. Optionally, the weights can be fine-tuned by means of context and domain knowledge.

TABLE VII
OVERVIEW OF METRIC WEIGHTS FOR WHICH TESTGAPRADAR OBTAINED THE BEST RANKING PERFORMANCE IN OUR MULTI-METHOD STUDY

Metric	Short	Weight
Code centrality	CEN	2
Changed lines of code	CLI	2
Complexity change	COC	2
Unresolved normal findings	UNF	1
Added critical findings	ACF	8
Unresolved critical findings	UCF	2
Removed critical findings	RCF	-2

For a successful implementation of the risk-based sorting of test gaps, the testing process needs to be enhanced. First, it is necessary to be able to differentiate between test gaps from finished development and work in progress. For example, a branching scheme could be implemented in the version control system so that stable code can be easily identified. Alternatively, code changes could be mapped to issues and the relevance of test gaps could then be inferred from the mapped ticket state. Second, the *test gap guard* role needs to

be established. The test gap guard is responsible for checking test gaps of finished code changes, for example, in a regular interval or in the testing phase before a release. From our experience, this role is taken either by test management, a test lead, a tester, or developers. Our approach comes into play when the test gap guard checks for open test gaps: They sort all test gaps within the timespan $[b, t]$ of their interest by estimated risk. Every test gap from the sorted list is then manually reviewed and risky test gaps need to be closed, typically by adding new test cases. Our approach helps to identify the most risky gaps early, resulting in more time to close the gaps, therefore increasing efficiency. In case there are too many test gaps to review all of them, effectivity is increased since review activities can be focused on more risky gaps.

VI. Threats to Validity

In this section, we discuss threats to internal and external validity and explain our mitigation strategies.

A. Internal Validity

The limited availability of data for the metrics at our study subjects represents a threat to internal validity because further metrics might result in better ranking performance. However, we have selected metrics that are related to well-known risk factors and, based on our experience, are readily collectible in highly regulated industrial projects employing test gap analysis. Consequently, prioritization can be readily incorporated and anchored in the development process. Our multi-method study results demonstrate that the ranking performance is sufficient for practical application.

The selection of parameters for our implementation poses a threat to internal validity, as the weights applied directly impact the risk score and subsequent ranking. We manually calibrated our selection using historical test gap reviews from one industrial partner (refer to Sec. IV-D). Our focus on evaluating a straightforward prioritization approach within an industrial setting forced us to conduct a multi-method case study under limited training data availability. Further refinements, particularly weight adjustments, are deferred, offering the potential for improved ranking outcomes.

Imbalance in data threatens internal validity. The data need to contain an appropriate balance between safe and risky test gaps. This is especially important as our approach estimates the relative risk in the respective set. As discussed in Section IV-E2, there is no universal definition of test gap risk. Therefore, there is no objective way to assess the validity of the set in this regard. However, a manual analysis showed that the study subjects contains a wide variety of test gaps, including several which can be considered as complex, as well as trivial ones.

B. External Validity

As true for most software engineering research, the huge diversity of software systems, processes, and teams, threatens the generalizability of our work [62]. All study subjects used in our evaluation are industrial, closed-source systems (which is not the case for most related work). While they implement sophisticated testing processes, there is a tremendous variety of testing in practice. For example, most open-source software projects often implement substantially different testing processes, for which TestGapRadar might produce different results. Nevertheless, with a technologically and process-related diverse set of study subjects from different industry partners, we share meaningful insights into the benefits and limitations of our work in practical use.

VII. Conclusion

The prevalence of test gaps introducing new defects presents a significant challenge in modern software development projects, for example, for test management and quality assurance, which need to review a large amount of test gaps. In this paper, we propose TestGapRadar, an automated approach for prioritizing test gaps based on their individual risk. For the risk estimation, we incorporated fourteen metrics reflecting three major risk factors, that is, code criticality, complexity, and static code analysis results. In a multi-method study, we validated our approach across eight large-scale software systems from two industry partners. Our study is based on an analysis of 31 historical test gap reviews for their systems and semi-structured interviews with the quality engineers who wrote those reviews. Our study showcased the effectiveness of TestGapRadar in ranking risky test gaps, on average, at the 23rd percentile. In a quality assurance expert survey, the external quality engineers of our industry partners underline the meaningful representation and potential superiority of the automated risk assessment of TestGapRadar over the expert judgments in certain scenarios. Our study's results underscore the significance of test gap risk estimation for facilitating risk-driven prioritization, empowering test management and quality assurance teams to efficiently pinpoint and manage critical test gaps. Our work enables practitioners to implement a risk-focused safety net into their testing process to ensure that no potentially risky code change is released untested. The quality engineers at our industry partners are definitely planning to implement our approach to prioritizing test gaps as part of their quality assurance processes.

VIII. Future Research Openings

The risk estimation could be enriched by production usage data to filter test gaps which are not used in production and highlight test gaps in heavily used core features. Also, natural-language processing of the commit messages that resulted in a test gap could help to estimate the associated risk. Future test gap risk estimation methods could cluster test gaps and aggregate the risk of a set of related test gaps. More sophisticated risk estimation methods, possibly including line-level defect prediction, will help to identify risky test gaps without needing to compare them with other test gaps. Key factors for practical adoption for any sophisticated approach are to make them approachable and understandable. For the adoption of artificial intelligence in the field of defect prediction, explainability is crucial to convince developers

of potential problems, motivating them to fix the underlying defect. Further, practitioners could be guided how to close risky test gaps, for example, by generating test cases that close the gaps and finally mitigate their risk.

IX. Data Availability

The raw data obtained in our study cannot be shared because of confidentiality agreements. For reproducibility, we publish aggregated data and the analysis scripts, along with additional details on our studies on a supplementary website: https://github.com/cqse/test-gap-risk-study.

References

[1] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," *IEEE Transactions on Software Engineering*, vol. 45, no. 2, pp. 111–147, 2017.

[2] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, 2012.

[3] L. Pascarella, F. Palomba, and A. Bacchelli, "Re-evaluating method-level bug prediction," in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2018, pp. 592–601.

[4] ——, "On the performance of method-level bug prediction: A negative result," *Journal of Systems and Software*, vol. 161, pp. 1–15, 2020.

[5] Z. Li, X.-Y. Jing, and X. Zhu, "Progress on approaches to software defect prediction," *IET Software*, vol. 12, no. 3, pp. 161–175, 2018.

[6] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang, "Perceptions, expectations, and challenges in defect prediction," *Transactions on Software Engineering*, vol. 46, no. 11, pp. 1241–1266, 2018.

[7] S. Stradowski and L. Madeyski, "Industrial applications of software defect prediction using machine learning: A business-driven systematic literature review," *Information and Software Technology*, vol. 159, 2023.

[8] V. Y. Shen, T.-j. Yu, S. M. Thebaut, and L. R. Paulsen, "Identifying error-prone software—an empirical study," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 4, pp. 317–324, 1985.

[9] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel, "Early quality prediction: a case study in telecommunications," *IEEE Software*, vol. 13, no. 1, pp. 65–71, 1996.

[10] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 2005.

[11] S. Eder, B. Hauptmann, M. Junker, E. Juergens, R. Vaas, and K.-H. Prommer, "Did We Test Our Changes? Assessing Alignment Between Tests and Development in Practice," in *Proceedings of the International Workshop on Automation of Software Test*. IEEE, 2013, pp. 107–110.

[12] E. Juergens and D. Pagano, "Did We Test the Right Thing? Experience with Test Gap Analysis in Practice," 2016. [Online]. Available: https://teamscale.com/2016-did-we-test-the-right-thing

[13] 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.

[14] A. Schwartz, "How to use analytics to eliminate the risk of your team's technical debt," 2019. [Online]. Available: https://www.sealights.io/learn/how-to-maintain-low-risk-technical-debt-using-analytics/

[15] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. 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.

[16] Y. Kamei and E. Shihab, "Defect Prediction: Accomplishments and Future Challenges," in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*, vol. 5. IEEE, 2016, pp. 33–45.

[17] F. Matloob, T. M. Ghazal, N. Taleb, S. Aftab, M. Ahmad, M. A. Khan, S. Abbas, and T. R. Soomro, "Software Defect Prediction Using Ensemble Learning: A Systematic Literature Review," *IEEE Access*, vol. 9, 2021.

[18] Y. Zhao, K. Damevski, and H. Chen, "A Systematic Survey of Just-in-Time Software Defect Prediction," *ACM Computing Surveys*, vol. 55, no. 10, pp. 1–35, 2023.

[19] M. Khatibsyarbini, M. A. Isa, D. N. 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.

[20] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[21] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[22] D. Steidl, B. Hummel, and E. Juergens, "Using network analysis for recommendation of central software classes," in *Proceedings of the Working Conference on Reverse Engineering*. IEEE, 2012, pp. 93–102.

[23] I. Sora, "A pagerank based recommender system for identifying key classes in software systems," in *Proceedings of the International Symposium on Applied Computational Intelligence and Informatics*, A. Szakál, Ed. IEEE, 2015, pp. 495–500.

[24] R. Haas, R. Niedermayr, T. Roehm, and S. Apel, "Is Static Analysis Able to Identify Unnecessary Source Code?" *Transactions on Software Engineering and Methodology*, vol. 29, no. 1, 2020.

[25] R. Niedermayr, T. Röhm, and S. Wagner, "Too trivial to test? an inverse view on defect prediction to identify methods with low fault risk," *PeerJ Computer Science*, vol. 5, no. 2, p. e187, 2019.

[26] M. Sailer, "Grouping and Prioritization of Test Gaps," Master's Thesis, Technical University of Munich, 2019. [Online]. Available: https://teamscale.com/2019-grouping-and-prioritization-of-test-gaps

[27] C. Brandt, M. Castelluccio, C. Holler, J. Kratzer, A. Zaidman, and A. Bacchelli, "Mind the Gap: What Working With Developers on Fuzz Tests Taught Us About Coverage Gaps," in *Proceedings of the International Conference on Software Engineering (Software Engineering in Practice)*. ACM, 2024, pp. 157–167.

[28] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 2008, pp. 346–355.

[29] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, 2005.

[30] F. Palomba, M. Zanoni, F. A. Fontana, A. de Lucia, and R. Oliveto, "Smells like teen spirit: Improving bug prediction performance using the intensity of code smells," in *Proceedings of the International Conference on Software Maintenance and Evolution*. Los Alamitos, California: IEEE, 2016, pp. 244–255.

[31] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the International Conference on Software Engineering*. ACM, 2008.

[32] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "DeepJIT: an End-to-End Deep Learning Framework for Just-in-Time Defect Prediction," in *Proceedings of the International Conference on Mining Software Repositories*. IEEE, 2019, pp. 34–45.

[33] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[34] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality," in *Proceedings of the International Conference on Software Engineering*. ACM, 2008, p. 521.

[35] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri, "Within-project defect prediction of infrastructure-as-code using product and process metrics," *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 2086–2104, 2021.

[36] D. Di Nucci, F. Palomba, G. de Rosa, G. Bavota, R. Oliveto, and A. de Lucia, "A developer centered bug prediction model," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 5–24, 2018.

[37] P. Tourani and B. Adams, "The Impact of Human Discussions on Just-in-Time Quality Assurance: An Empirical Study on OpenStack and eclipse," in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*, vol. 1. IEEE, 2016, pp. 189–200.

[38] H. D. Tessema and S. L. Abebe, "Enhancing Just-in-Time Defect Prediction Using Change Request-Based Metrics," in *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2021, pp. 511–515.

[39] A. Trautsch, S. Herbold, and J. Grabowski, "Static Source Code Metrics and Static Analysis Warnings for Fine-Grained Just-in-Time Defect Prediction," in *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2020, pp. 127–138.

[40] D. Falessi, S. M. Laureani, J. Çarka, M. Esposito, and D. A. d. Costa, "Enhancing the defectiveness prediction of methods and classes via JIT," *Empirical Software Engineering*, vol. 28, no. 37, 2023.

[41] C. Pornprasit and C. K. Tantithamthavorn, " DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction ," *IEEE Transactions on Software Engineering*, vol. 49, no. 01, pp. 84–98, 2023.

[42] S. Yin, S. Guo, H. Li, C. Li, R. Chen, X. Li, and H. Jiang, "Line-level defect prediction by capturing code contexts with graph convolutional networks," *IEEE Transactions on Software Engineering*, 2024.

[43] European Parliament and of the Council, "Directive 2008/104/ec of the european parliament and of the council of 19 november 2008 on temporary agency work," Official Journal of the European Union, 2022. [Online]. Available: https://eur-lex.europa.eu/eli/dir/2008/104/oj

[44] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Technical Report, Stanford InfoLab, 1999.

[45] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 423–433, 1992.

[46] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[47] A. E. Hassan, "Predicting faults using the complexity of code changes," in *International Conference on Software Engineering*. IEEE, 2009, pp. 78–88.

[48] A. H. Watson, D. R. Wallace, and T. J. McCabe, *Structured Testing: A Testing Methodology using the Cyclomatic Complexity Metric*. US National Institute of Standards and Technology, 1996, vol. 500, no. 235.

[49] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[50] R. Haas, R. Niedermayr, and E. Juergens, "Teamscale: Tackle technical debt and control the quality of your software," in *Proceedings of the International Conference on Technical Debt*. IEEE, 2019, pp. 55–56.

[51] D. Steidl, F. Deissenboeck, M. Poehlmann, R. Heinke, and B. Uhink-Mergenthaler, "Continuous software quality control in practice," in *International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 561–564.

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

[53] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting Experiments in Software Engineering," in *Guide to Advanced Empirical Software Engineering*. Springer, 2008, pp. 201–228.

[54] European Parliament and of the Council, "Regulation on digital operational resilience for the financial sector and amending regulations (ec) no 1060/2009, (eu) no 648/2012, (eu) no 600/2014, (eu) no 909/2014 and (eu) 2016/1011," Official Journal of the European Union, 2022. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2022/2554

[55] C. Popeea-Simeth. (2018) Monthly assessments: How often to inspect the code quality in a code quality control process? Last retrieved 2024-01-16. [Online]. Available: https://teamscale.com/blog/en/news/blog/monthly-assessments

[56] M. Kendall, "The treatment of ties in ranking problems," *Biometrika*, vol. 33, no. 3, pp. 239–251, 1945.

[57] P. Virtanen, R. Gommers, and T. e. a. Oliphant, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[58] I. M. Sobol, "Global sensitivity indices for nonlinear mathematical models and their monte carlo estimates," *Mathematics and Computers in Simulation*, vol. 55, no. 1, pp. 271–280, 2001.

[59] J. Herman and W. Usher, "SALib: An open-source Python library for sensitivity analysis," *Journal of Open Source Software*, vol. 2, no. 9, p. 97, 2017.

[60] P. Mayring, *Qualitative Content Analysis: A Step-by-Step Guide*. SAGE, 2021.

[61] J. Cohen, *Statistical power analysis for the behavioral sciences—2nd ed.* Lawrence Erlbaum Associates, 1988.

[62] 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.