Test Case Prioritization: An Experience Report within Industrial Railway Signalling

Abiola Paterne Chokki[1], Onur Kilincceker[2] and Serge Demeyer[2]

[1]CETIC (Centre d'Excellence en Technologies de l'Information et de la Communication), Charleroi, Belgium
[2]University of Antwerp and Flanders Make, Antwerp, Belgium

Abstract

Many industrial test processes still rely on manual, system-level execution under a certified process. Running every test before a release then costs a lot of engineering effort, so the order of execution matters. Test case prioritization (TCP) reorders a test suite so that the tests most likely to fail run first. We report an industrial case study on TCP, carried out with a railway signalling supplier, built around three questions and checked against the wider TCP literature. We ask which factors help identify failing test cases, how manual, weighted, and machine-learning (ML) orderings compare in early failure detection, and which obstacles arise in a manual, document-driven process and whether the remedies from the literature address them. History- and requirement-based factors matter most, as in prior work, alongside other relevant factors we identified. Read cumulatively, the weighted heuristic was ahead of the manual ordering at most checkpoints, but by small margins on few failing tests, and the ML classifier fell behind on first runs, even after an extensive model and resampling search.

Keywords

test case prioritization, regression testing, software evolution, machine learning, safety-critical software, industrial case study, railway signalling

1. Introduction

In settings where testing is manual and happens at system level, regression testing is one of the more expensive parts of keeping a long-lived software system alive: every new baseline, a delivered version of the system, has to be re-validated against a test suite that keeps growing as the system picks up new requirements, features, and fixes. Test case prioritization (TCP) reorders the suite so that fault-revealing tests run first [1, 2]. Most TCP research evaluates on open benchmarks where a single, clean source of information (code, coverage, execution logs) is available for the whole suite [3, 4]. Rather than introducing a new TCP technique, we report an industrial case study examining how existing approaches behave in a safety-critical, document-heavy setting.

The case study was carried out with a railway signalling supplier, as part of a collaborative applied-research project on testing methods for complex, safety-critical cyber-physical systems. Section 3 describes the setting in detail. In short, requirements, tests, code, and change requests are tracked across dozens of documents under a process regulated by CENELEC (the European Committee for Electrotechnical Standardization) EN 50128 [5]. Test execution is mostly manual, and the ordering any new technique must outperform is not a random baseline, but one that draws on engineers' accumulated risk expertise and is adjusted based on previous test execution results. We organize what we learned around three research questions (RQs).

- **RQ1 (Factors).** Which factors have the most impact on correctly identifying failing test cases, and how do they relate to the factors reported in the TCP literature?
- **RQ2 (Performance).** How do expert-driven manual, factor-weighted, and machine-learning (ML) test orderings compare in early detection of failing tests, when the manual ordering already reflects engineering expertise?

BENEVOL 2026: The 25th Belgium-Netherlands Software Evolution Workshop, November 26–27, 2026, Madrid, Spain
abiola.chokki@cetic.be (A. P. Chokki); onur.kilincceker@uantwerpen.be (O. Kilincceker); serge.demeyer@uantwerpen.be (S. Demeyer)
0000-0003-4500-2141 (A. P. Chokki); 0000-0001-5996-4398 (O. Kilincceker); 0000-0002-4463-2945 (S. Demeyer)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

- **RQ3 (Challenges).** Which obstacles arise when applying TCP in a manual, document-driven V&V process, and do the remedies proposed in the literature address them?

This is a single case study, so we do not claim that our numbers would reproduce elsewhere. Following Yin's case study methodology [6], we offer analytic generalization: evidence-grounded lessons that other organizations in a similar situation can check against their own data.

This paper makes three contributions. First, an industrial characterization of the factors associated with test-case failure in a traceability-rich, safety-critical setting, including six factors, with their definitions and calculations (Section 5), that we did not find reported in a broad review of the TCP literature. Second, an empirical comparison of the existing manual ordering against a weighted heuristic and an ML-based alternative across five real baseline and run combinations, including possible reasons why ML(default) lags on first runs of a baseline, reported using a quartile-checkpoint and cumulative-curve approach (Section 4.2) since the field's usual metrics could not be computed from our data. Third, evidence on the practical barriers to applying TCP and ML techniques in this setting, cold start, unstable predictive relationships, class imbalance, incomplete traceability, and coarse-grained result logging, checked one by one against what the literature prescribes for each. The rest of the paper reviews the relevant literature (Section 2), describes the setting and methodology (Sections 3-4), answers RQ1-RQ3 in turn (Sections 5-7), then discusses limitations (Section 8) before concluding (Section 9).

2. Related Work

This section situates our case against five threads of prior work: the TCP taxonomy, the factors it uses, how it measures success, how it compares to manual testing, and the challenges it runs into.

2.1. Regression Testing and a Taxonomy of TCP Techniques

Rothermel et al. [1] formalized TCP and introduced the Average Percentage of Faults Detected (APFD) metric that most later work still uses. Two systematic literature reviews map the field at a larger scale: Khatibsyarbini et al. [3] review 52 studies (2007-2020) and find that TCP effectiveness clusters around cost, code coverage, and fault-detection ability, with six context factors shown to significantly affect how well a technique works; Pan et al. [4] do the same for ML-based TCP and test selection specifically. The techniques relevant to our setting fall into seven families: requirement-based (ReB), cost-effective (CeB), risk-based (RiB), dependency-based (DeB), coverage-based (CoB, which needs source-code access), machine-learning-based (ML), and re-prioritization (ReP), which keeps adjusting the ordering while tests run, using feedback from tests already executed; Pradhan et al.'s REMAP exemplifies the last family, reporting that only 21% of test cases, on average, are needed to detect 81% of the faults across five case studies [7].

2.2. Factors Used Across the Literature

Beyond the two systematic literature reviews, individual studies converge on a small set of factor families. History and fault-proneness show up the most: ROCKET's use of historical failure data at Cisco [8], Engström et al.'s history-based industrial study [9], and Mahdieh et al.'s fault-proneness estimates [10] all lean on past-failure signal. Requirement and risk factors appear in Srikanth et al.'s industrial risk-factor study and system-test work [11, 12]. Code-centric factors, coverage, complexity, and churn, dominate the open-benchmark ML literature: Anderson et al. [13], Khan et al. [14], Lachmann et al. [15], and the failure-prediction line of Noor and Hemmati and Palma et al. [16, 17]. Similarity and clustering factors appear in Carlson et al. [18]. Two studies lack clean code access, much like us: Magalhães et al. search for the "best criteria" for industrial UI regression testing without source code [19], and Di Nardo et al. report an industrial coverage-based study where coverage itself had to be approximated [20]. Ulewicz and Vogel-Heuser combine runtime data with change-impact analysis for production-automation systems [21], and Sharif et al.'s DeepOrder retrains continuously inside a continuous integration (CI) pipeline [22].

2.3. Metrics Used Across the Literature

Average Percentage of Faults Detected (APFD), and its cost-cognizant variant Average Percentage of Faults Detected considering test-case execution Cost (APFDc) (introduced by Elbaum et al. to account for test cost (usually execution time) and fault severity), are still the two dominant metrics; Normalized Average Percentage of Faults Detected (NAPFD) adapts APFD for cases where not all faults are known or executed. In Ahmed et al.'s systematic literature review, APFD appears in 24% of the surveyed studies, APFDc in 18%, and NAPFD in 9% [23]. All of these need a per-test, per-fault log: the exact position of each test in the execution order, and which fault it revealed. Section 6 explains that our partner's process, and probably many manual, document-mediated, safety-critical verification and validation (V&V) processes, simply does not produce logs at that level of detail.

2.4. Studies Comparing Automated Techniques to a Manual Baseline

Most TCP studies, including the foundational work [1, 24], compare a prioritized ordering against a random or untreated baseline rather than against an ordering produced by a human tester. The closest study we found to our setting is Hemmati, Fang, and Mäntylä's work on manual, black-box prioritization, where source code is unavailable and testers already apply their own judgment [25, 26]. Across four releases of desktop Firefox, none of the automated techniques they tried clearly outperformed the others or, implicitly, the manual process they were intended to improve. Similarly, Di Nardo et al., Srikanth and Banerjee, and Carlson et al. [20, 12, 18] report real but modest gains when evaluated in realistic industrial settings rather than against a random baseline.

2.5. Challenges and Proposed Solutions in ML and TCP Literature

Four challenges recur across the ML-based TCP literature reviewed by Pan et al. [4] and in the broader machine learning literature. Class imbalance is usually addressed with resampling, cost-sensitive learning (which penalises errors on the rare failing class more heavily), and ensembles (Kaur et al. [27]). Feature selection is typically handled with tree-based importance and multicollinearity filtering (Khan et al. [14]). Cold start for new test cases is addressed by combining static and dynamic features [16, 17] or by retraining often in a CI loop, as DeepOrder does [22]. Missing runtime feedback is addressed by re-prioritization such as REMAP [7].

3. Industrial Context

The case study was carried out with a railway signalling supplier that develops highly configurable, safety-related embedded control systems under CENELEC EN 50128 [5], as part of the collaborative project introduced in Section 1. System-level V&V is mostly manual. The partner framed the goal in terms of the classic quality-cost-delay triangle: find confirmed failing tests ("NOK Confirmed") earlier, within a fixed V&V budget. Cost here means data setup, test execution, and result analysis. A test execution ends as OK or NOK (failed), and NOK Confirmed marks a failure that the engineers confirmed after analysis. We count failing tests rather than faults, since one fault can make several tests fail, so our curves are test-level counterparts of APFD.

Figure 1 shows how the partner's documents relate to one another. The Requirements Traceability Matrix (RTM) lists the requirements for a baseline together with their traceability to test cases and test sheets. The Test Progress Report (TPR) tracks how many tests have been executed over time, which test cases remain to run for the baseline, the traceability between test sheets and test cases, and requirement changes. The edges labelled TL (Traceability Link) are dedicated traceability matrices, each linking two artefact types one lifecycle level apart. They chain the System Requirements Specification (SRS) to the System Architecture Description (SAD), the SAD onward to the code, and the SAD to an analogous document maintained for a specific subsystem; it is also what links Code, Component, the Baseline Configuration Record (BCR, a list of configuration parameters for a given baseline), and Safety Integrity

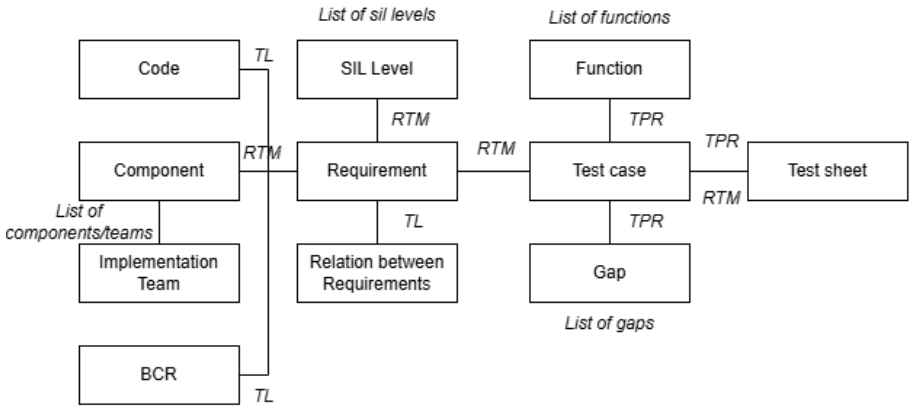


Figure 1: Traceability model of the partner's documents. Edge labels name the document type that provides the link.

Level (SIL) to Requirement in the figure. Across the baselines we studied, 89 such documents were imported, generating over 49,000 test-case-change records and 27,000 requirement-change records.

4. Methodology

This section describes the prototype and the two methodology versions it went through (V1.0, V2.0), then defines exactly how we compare orderings and measure their effectiveness.

4.1. Prototype Architecture and Methodology V1.0/V2.0

The prototype reads the partner's documents and turns each test case into a row of numbers, called factors, that describe how risky the test looks. We built a prototype (Django/Python, PostgreSQL) that imports and normalizes the RTM, TPR, traceability documents, and reference lists into a data warehouse, then computes, for each test case, the value of every retained factor for a given baseline and run. Version 1.0's design combined a top-down analysis, tracing requirements down to the code and feeding the ReB and CeB factors, with a bottom-up analysis intended to start from source code complexity and change history and feed the CoB and implementation-complexity factors. The bottom-up half was never actually computable: traceability between code, requirements, and test cases was too limited to calculate these factors with any confidence, and every result reported here rests on the top-down, requirement- and history-based factors alone. Through brainstorming with the partner's engineers we settled on fourteen top-down and three (unimplementable) bottom-up factors (F1 to F17), each with an elicited weight from 1 to 5; Section 5 defines the factors most central to this paper's findings.

The Weighted ordering combines these factors into one score per test case. Each factor j contributes to a per-test-case weight via

$$WP_i = \sum_j WeightFactor_j \cdot \frac{FactorValue_{ij}}{\max_k FactorValue_{kj}}, \tag{1}$$

where WP_i is the priority score of test case i, and tests with higher scores run earlier. $FactorValue_{ij}$ is the value of factor j for test i, for example the number of tests linked to its requirements. $WeightFactor_j$ is the weight of factor j, from 1 to 5 in V1.0 and derived from feature importance in V2.0. Dividing by the largest value of factor j over all tests rescales each factor to between 0 and 1, so that factors with large raw values do not dominate. There is one caveat for how to read the resulting order: test cases are first split into two strata by the delta or non-regression factor, change-impacted before non-regression, and only within each stratum are they ranked by WP_i.

The ML ordering instead learns from past outcomes. The same normalized factors also feed a supervised classifier that predicts whether a test will pass or fail. Nine model families (including logistic regression, random forest, gradient boosting, and a small neural network) were each combined, via grid search, with six ways of handling class imbalance (four resampling variants, NearMiss, and a cost-sensitive class-weight variant). None of these 54 combinations produced meaningfully better accuracy, precision, or F1 than the others . The search was expensive and did not surface a clear winner, a result we discuss in Section 7. Logistic regression with SMOTE (a method that creates synthetic examples of the rare failing class) was kept as the deployed default for being the fastest configuration among those with comparable classification performance. Its hyperparameters were tuned by grid search over 48 settings (regularisation strength, penalty, solver, and class weights), using 5-fold stratified cross-validation with a fixed seed on 80% of the historical pool, with SMOTE applied inside each fold. The search was run once, without repeated seeds. The decision threshold that maximises F1 was chosen once on the remaining 20% and kept when the classifier was refit on the full pool. ML(default) refers to this logistic-regression-plus-SMOTE configuration throughout, with the same delta-before-non-regression stratification applied to its predictions.

Version 2.0 added data cleaning, an expansion to 30 engineered features, the model and resampling search just described, and feature selection through tree-based importance and Variance Inflation Factor (VIF) filtering, reused as weights.

4.2. Evaluation Protocol and Reporting Approach

For each baseline and run we compare three orderings: Manual, Weighted (Equation 1 with V2.0 feature-importance weights), and ML(default), the logistic-regression classifier described above. Manual is the ordering the partner's engineers actually produce in practice: they choose which tests to run based on accumulated risk judgment, and adjust the remaining order as results from tests already executed in the same campaign come in. This matters for how to read our results, since it means Manual, unlike the other two orderings, already incorporates a form of within-campaign feedback that Weighted and ML(default) do not (Section 7). RUN1 is the first, full campaign for a baseline, covering both change-impacted and non-regression tests; later runs mainly re-test corrections, so RUN1 typically has far more test cases. The dataset holds three baselines in chronological order. The first, with two runs (4292 tests, 446 NOK), serves only as initial training data and is never tested. We report the two later baselines, B1 (three runs) and B2 (two runs). Each run is evaluated with models built only from earlier runs, so B1 RUN1 is trained on the two runs of the first baseline. Between 10 and 23% of the tests of the five evaluated runs are NOK Confirmed (Table 3). For every baseline and run, Weighted and ML(default) are computed using only information from previous baselines and runs; the evaluated baseline's own data is never used for training, weighting, or model selection.

We report the comparison in two complementary ways rather than relying on a single scalar metric, since, as Section 2.3 explains, the field's usual metrics cannot be computed from our data. In each case we track the same underlying quantity, the cumulative number of confirmed failing tests found as tests are executed. The first view is a table of cumulative values at 25, 50, and 75% of the suite, comparing all five baseline and run combinations side by side at a glance. The second is a graph of the full cumulative curve, tests executed on the horizontal axis and cumulative NOK-Confirmed found on the vertical axis, showing the shape of the detection process for a given run in more detail than a few checkpoints can. Both views summarize the cumulative fault-detection process used in APFD-family evaluations; we just report it as checkpoints and as a curve, aimed at practitioners, instead of collapsing it into a single area-under-curve number. Section 6 explains why we could not also report the literature's usual scalar metrics.

5. RQ1: Factors and Their Relation to Prior Work

This section reports which factors were actually predictive in our case, then checks that ranking against the literature reviewed in Section 2.

Table 1
Importance of the ten features ranked highest, ex-ante, across five (baseline, run) combinations.

ID	Feature	Importance				
		R1	R2	R3	R1′	R2′
F1	# direct requirement test cases	.149	.130	.116	.111	.122
F2	Hist. failed functions factor	.115	.082	.078	.086	.076
F3	Delta / non-regression factor	.086	.077	.083	.090	.082
F4	Hist. failed components factor	.083	.120	.120	.103	.116
F5	# change requests on code	.046	.039	.043	.044	.039
F6	Extent of requirement changes	.040	.057	.054	.057	.056
F7	# change requests on documentation	.038	.040	.039	.047	.036
F8	Req. volatility between baselines	.031	.042	.040	.032	.042
F9	Req. volatility during current baseline	.028	.032	.036	.035	.035
F10	Last-run status: OK	.020	.033	.036	.041	.047

R1-R3: baseline B1, runs 1-3.

R1′-R2′: baseline B2, runs 1-2.

5.1. Feature Importance in Our Case

We use "important" throughout this section to mean features receiving high model-based importance (from RandomForest, GradientBoost, and XGBoost) under the ex-ante training protocol of Section 4.

Table 1 shows the importance of the ten highest-ranked features (out of thirty computed in total), when training is restricted to the data available before each of five baseline and run combinations. The definitions and calculations of all thirty features are published in a separate Zenodo record[1]. The ranking is fairly stable at the top: the number of requirement-linked test cases, the historical failure rate of the functions and components involved, and the delta or non-regression factor are consistently among the most important features across all five runs. When we instead compute importance with hindsight, using only the data of the specific run being tested, the ranking changes further down the list. Features describing the outcome of the last test run and requirement volatility during the current baseline rise to the top positions in some runs, despite ranking lower ex-ante. This indicates that feature importance varies considerably across runs and is not stable from one run to another.

5.2. How This Lines Up with the Literature

History and fault-proneness factors are the most consistent predictors both in our case and across the literature surveyed in Section 2.2. Our top features (F2, historically failed functions; F4, historically failed components) line up closely with Engström et al.'s purely history-based prioritization [9], with ROCKET's use of historical failure data [8], and with Mahdieh et al.'s fault-proneness estimation [10]. Our requirement-linked factors echo the requirement-based risk factors of Srikanth et al. [11, 12].

Checking our factor set against this literature, that is, the two dedicated systematic literature reviews (one reviewing 52 studies [3], the other surveying ML-based TCP [4]) and the fifteen individual studies discussed above, six factors were proposed through brainstorming with the partner's engineers based on information available in their traceability infrastructure that was not represented by the factors identified in our literature review. Table 2 defines each one. All six depend on a traceability graph connecting requirements, functions, components, and change requests to test cases, which the open-benchmark literature does not have access to, so their absence there may say more about what benchmarks expose than about whether the factors are useful. F1 (`nb_direct_req_test_cases`), the number of test cases linked to the same requirement, is also our single most important feature (Table 1); it reflects a test case's position in the traceability graph itself, close in spirit to Magalhães et al.'s and Di Nardo et al.'s observation that, without clean code access, non-code artefacts have to carry the predictive signal [19, 20], though neither reports anything quite like it. The two historically-failed functions/components factors sit within the same broad failure-history family as Engström et

[1] https://doi.org/10.5281/zenodo.24039572

Table 2
Factors proposed through brainstorming with the partner's engineers that we did not find reported in the TCP literature we surveyed.

Factor	Definition	Calculation
`nb_direct_req_test_cases`	Requirement fan-in: total test cases linked to the same requirements as this test case	Sum, over the test case's linked requirements r, of the number of test case versions linked to r
`factor_historically_failed_functions`	Average historical failure rate of the functions linked to the test case	For each linked function f, failed / final executions across tests linked to f; averaged over the test case's linked functions
`factor_historically_failed_components`	Average historical failure rate of the components linked to the test case	Same construction as the function factor, aggregated over linked components instead
`delta_non_reg_factor`	Impact-analysis class: change-impacted (delta) test versus non-regression test; also the primary sort key for every ordering in this paper (Section 4)	1 if the test case's impact analysis contains "delta", 0 if it contains "non-reg" (default 0)
`nb_cr_code`	Number of code change requests that the test case is linked to	Count of code-side change-request references linked to the test case
`nb_cr_doc`	Number of documentation change requests that the test case is linked to	Count of documentation-side change-request references linked to the test case

al., ROCKET, and Mahdieh et al. [9, 8, 10], but aggregate failure rates over the functions and components a test exercises rather than over the test's own execution history, possible only because of the traceability graph. The delta or non-regression factor and the two change-request counts tie a test case to impact-analysis and change-request activity directly, which none of the fifteen studies we surveyed do.

Two further features that rank highly in Table 1, the extent of requirement changes and requirement volatility, refine factors the literature already reports (requirement volatility, code churn) rather than something new. Splitting volatility into a between-baselines and a during-the-current-baseline component is more fine-grained than the single factor usually reported, and this split matters empirically: the within-baseline component climbs in importance ex-post but not ex-ante (Section 5.1).

We propose the six factors of Table 2 as candidates worth trying in any organization with the same underlying capability, a traceability graph linking requirements, functions, components, and change requests to test cases, whether or not that organization operates under a formal safety-classification scheme.

6. RQ2: Performance, and How to Measure It

Answering RQ2 takes four steps. Section 6.1 reports how the three orderings actually compared, and Section 6.2 discusses possible reasons why ML(default) lagged on the first run of a baseline. Section 6.3 relates that comparison to the literature, and Section 6.4 addresses a question that turned out to be inseparable from the first: how to even measure TCP performance when the field's usual metrics cannot be computed.

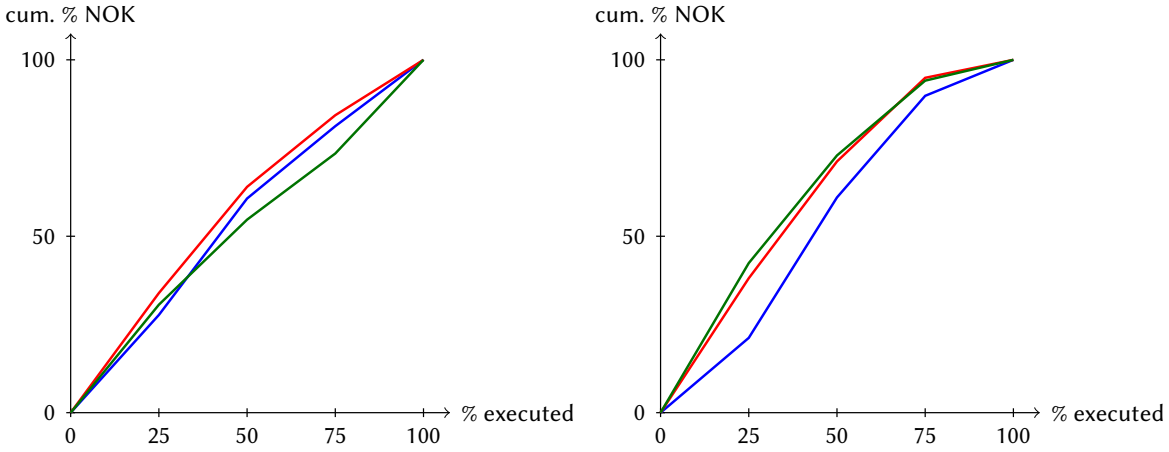
6.1. Results

Table 3 reports, for five baseline and run combinations, the cumulative percentage of confirmed failing tests found within each share of the executed suite.

We read the orderings cumulatively, since an engineer who stops early keeps everything found so far. Weighted is ahead of Manual at 11 of the 15 checkpoints, behind at 3, and tied at one, with margins

Table 3
Cumulative percentage of confirmed failing tests found after 25, 50, and 75% of the executed test suite, and share found in the last quarter, for the Manual, Weighted, and ML(default) orderings, across five real (baseline, run) combinations, with the number of tests executed and of confirmed failing tests (NOK) in each run. The best value at each checkpoint is in bold.

Baseline / Run	Ordering	25%	50%	75%	Last 25%
B1 - RUN1 2128 tests, 448 NOK	Manual	27.7	60.7	81.2	18.8
	Weighted	**33.9**	**64.0**	**84.3**	15.7
	ML(default)	30.6	54.7	73.5	26.5
B1 - RUN2 570 tests, 118 NOK	Manual	21.2	61.0	89.8	10.2
	Weighted	38.1	71.2	**94.9**	5.1
	ML(default)	**42.4**	**72.9**	94.1	5.9
B1 - RUN3 128 tests, 29 NOK	Manual	**41.4**	41.4	62.1	37.9
	Weighted	13.8	**62.1**	**86.2**	13.8
	ML(default)	31.0	62.0	82.7	17.3
B2 - RUN1 1911 tests, 263 NOK	Manual	14.1	47.6	**81.1**	18.9
	Weighted	**16.0**	44.1	77.2	22.8
	ML(default)	13.7	**51.0**	71.5	28.5
B2 - RUN2 599 tests, 60 NOK	Manual	31.7	68.4	78.4	21.6
	Weighted	**46.7**	68.4	**91.7**	8.3
	ML(default)	35.0	**76.7**	**91.7**	8.3



(a) B1, RUN1 (b) B1, RUN2

Figure 2: Cumulative NOK-Confirmed detection curves (derived from Table 3) for Manual (blue), Weighted (red), and ML(default) (green). On RUN1 (a), ML(default) visibly lags behind both other orderings for most of the campaign; on RUN2 (b), it is competitive with Weighted and ahead of Manual.

of about 2 to 24 points where it leads. ML(default) is ahead of Manual at 10 of 15 checkpoints, but behind on both first runs after 75% of the suite. Manual is best at only 2 checkpoints, and it falls far behind after a strong start on RUN3 of B1. No ordering is best everywhere, and the differences rest on few failing tests: on the shortest run (29 failing tests), one test moves a value by about three points.

6.2. Why ML(default) Lags on First Runs: Possible Readings

Figure 2 puts the cumulative detection curves for RUN1 and RUN2 of the same baseline side by side. The same pattern is observed for both available RUN1 cases presented in Table 3: ML(default) leaves more confirmed failing tests for the last quarter of the suite than Manual on both baselines' RUN1 (26.5 versus 18.8% for B1, and 28.5 versus 18.9% for B2), while it improves on Manual for later runs of the same baseline. We see three possible readings, and our data cannot separate them. First, it is

the first campaign of a newly delivered baseline, so training data comes only from previous baselines, which means cold start (Section 2.5) is at its worst: many test cases are new or newly linked to changed requirements: in RUN1, 68% of the tests of B1 and 41% of those of B2 are new, against at most 4% in later runs. A new test has no execution history, so history features such as last-run status, age, and last failure age take fixed default codes (0 or -1) and carry no information. Features derived from the traceability graph, such as requirement fan-in, stay defined once the test is linked, so cold start affects only part of the feature set. Second, a model trained on prior baselines is most exposed to the ex-ante/ex-post feature-importance instability of Section 5.1 on the first run of a baseline it has never seen; we read this as suggestive of concept or distribution drift, not something five baseline and run combinations can formally establish. Third, later runs may simply be easier than RUN1: they mainly re-test corrections, and last-run status and within-baseline volatility carry information from earlier runs of the same baseline. This matters in practice because RUN1 is also the run where the quality-cost-delay objectives of Section 3 are most sensitive to getting the ordering right, and exactly where the automated approaches are weakest. Manual does not show this weakness, plausibly because it is not a static model trained on past baselines at all: engineers choose the next tests based on accumulated risk judgment and adjust as results from the current campaign come in (Section 4.2), a within-campaign feedback signal that neither Weighted nor ML(default) has, and which may explain why Manual stays competitive overall. With two baselines only, we treat the first-run effect as a hypothesis.

6.3. Performance Relative to the Literature

Our finding that gains over the expert ordering are small and not consistent lines up with Hemmati et al.'s result for manual, black-box prioritization: across four Firefox releases, they also found that none of the automated techniques they tried clearly beat the others [25, 26].

6.4. Measuring Performance When APFD Is Not an Option

The second part of RQ2 concerns the feasibility of measuring TCP performance with the available industrial logs. We could not compute APFD, APFDc, or NAPFD for our data, even though these are the dominant metrics in the literature (Section 2.3): the partner's RTM and TPR tooling records confirmed failing tests at the campaign or slot level, not as a timestamped, per-test, per-fault trace. TCP metric choice implicitly assumes an automated, timestamped result-logging setup that CI-based studies (ROCKET [8], DeepOrder [22]) get for free, but that a manual, campaign-based V&V process does not. The metric-focused systematic literature reviews [23, 3] we consulted focus on the metrics reported by existing studies and do not address whether industrial infrastructure provides the information required to compute them. The checkpoint table and cumulative-curve pairing of Section 4.2 is our response: a reporting approach suited to what this infrastructure can support, not a lesser substitute for APFD.

7. RQ3: Challenges, and Whether the Literature's Fixes Worked

Every obstacle below was already reported somewhere in the TCP or general ML literature (Section 2.5); none of the six is a new problem. What we did not find answered is whether the remedies deliver in a setting like ours: document-mediated traceability and sequential rather than single-split evaluation. Table 4 lists the six obstacles, the literature's diagnosis and remedy, whether we applied it, and whether it delivered.

Two patterns cut across Table 4. The remedies borrowed from the general ML literature (C0, C3, and the model half of C4) helped least, not for lack of trying. They target a single, stationary dataset, while the feature-outcome relationship appears unstable across sequential baselines (RQ1), which most TCP-ML studies do not test. The remaining gap, C2's cold start, is partial because a genuinely new test case has no history to retrain onto.

Table 4
Recurring challenges: literature diagnosis and remedy, versus what we applied and observed.

Challenge	Literature remedy	Applied?	Outcome and why
C0. Weak and unstable predictive signal	Richer feature engineering, ensembles [13, 14]	Yes (30 features, RF/GB/XGB)	Did not fix it; recall stayed close to chance. Feature engineering assumes a stable feature-outcome relationship, but Section 5.1 shows it is unstable between baselines, suggestive of concept drift, rarely discussed in the TCP-ML papers we read.
C1. Training data has inconsistent, incorrect, or missing values	No dedicated TCP fix in the literature; general data-quality engineering (validation, deduplication)	Yes (cleaning, dedup logic)	Made the pipeline usable, but only fixed structural problems, not links that were never captured: traceability between code, requirements, and tests was too limited to compute the intended bottom-up, code-based factors.
C2. No differentiating features for predicting new or changed test cases	Combine static and dynamic features [16, 17]; retrain often [22]	Yes (static and dynamic features; retrained on all prior baselines before each new one)	Still only partial (Section 6.2): retraining on cumulative history happens before every new baseline, but cannot manufacture history for a test case that has none.
C3. Class imbalance (10 to 23% NOK Confirmed in the evaluated runs)	SMOTE, cost-sensitive learning, ensembles [27]	Yes (six resampling strategies tried, see C4)	Only a marginal improvement, across every variant tried. If the predictive signal is unstable across baselines (RQ1), no resampling scheme built from past baselines represents the current decision boundary, a failure mode Kaur et al.'s survey of static, single-dataset imbalance does not cover.
C4. No method to identify important features, the best model, or optimal weights, instead of subjective user-defined inputs	Tree-based importance, correlation or VIF filtering, grid search [14]	Yes, fully (9 models $\times$ 6 imbalance strategies; importance/VIF as weights)	The feature and weight halves worked: importance-based weighting performed best overall (Table 3). The model half was attempted in full, which is what makes this informative: none of the 54 combinations differentiated meaningfully on accuracy, precision, or F1, at real running-time cost. Logistic regression with SMOTE was kept for being fastest among equally mediocre options, a symptom of C0, not a separate, fixable gap.
C5. No runtime feedback loop	Rule mining plus similarity-based re-prioritization (REMAP) [7]	No, not for Weighted or ML(default)	Manual already has an informal version of this (Section 4.2): engineers adjust as results from the current campaign come in, which may be part of why it stays competitive. A formal loop for Weighted and ML(default) would add genuinely new information, so it remains worth trying (Section 9), though C0 and C4 both suggest existing factors carry little signal regardless of model choice.

8. Threats to Validity

This section covers four threats: bias in our internal choices, whether our metric measures the right thing, whether the study can be checked independently, and how far the findings generalize.

Internal validity. The V1.0 weights came out of brainstorming rather than a formal utility-elicitation process. The model comparison behind ML(default) was thorough (nine models, six resampling strategies, Table 4, C4), but all of it was run on the same, possibly weak, factor set; a different feature set might still change the outcome, and we cannot rule that out from this data alone. Five baseline and run combinations is not a lot, which limits how much statistical weight the comparison can carry.

Construct validity. Our quartile-based metric is built from the same underlying curve as APFD

(Section 4.2), at a coarser resolution and without cost or severity weighting, which the data did not allow (Section 6.4).

Verifiability. The dataset underlying this study belongs to the industrial partner and cannot be released, because of a confidentiality agreement. We mitigate this by documenting every factor's definition and calculation in full (Table 2, Section 5, and a separate Zenodo record with all thirty features[1]) and by describing the pipeline, the model and resampling search, and the evaluation protocol precisely enough for another organization with comparable traceability data to reproduce the method, if not the exact numbers.

External validity and generalization. This is a single case study, one industrial partner, one regulatory context, one specific set of documents; our numbers would not necessarily show up the same way elsewhere. Following Yin's analytic generalization [6], the six obstacles in Section 7 and the factors identified in Section 5 are best read as evidence-based observations about a particular kind of setting, regulator-driven, document-heavy, low-fault-rate, with a competent manual process already in place, for other organizations to check against their own data. Readers outside this scope, with higher fault rates, direct code-coverage access, or fully automated CI testing, should expect different, likely more favourable, results for ML-based TCP, consistent with Section 2.2.

9. Conclusion and Future Work

Much of the TCP literature we reviewed evaluates settings where the relationship between features and outcomes is stable and code coverage is available. Test results are logged automatically at a fine-grained level, capturing the execution order, pass/fail outcome, and faults detected by each test, which are needed to compute APFD. Our case lacked the required information. New factors were needed because there was no clean code signal to fall back on (RQ1). Read cumulatively, the weighted heuristic was ahead of the manual ordering at most checkpoints, by small margins on few failing tests. Manual, which adapts within a campaign, stayed competitive, and ML(default) fell behind on first runs. We also could not fall back on the field's default metrics either, because the feature-outcome relationship appears unstable across baselines and the result logs were too coarse (RQ2), which is why we report our comparison using the quartile-checkpoint and cumulative-curve approach instead of treating the gap as a dead end. The usual fixes for class imbalance and unstable predictions only partly worked, because they are built for a single, static dataset rather than a sequence of evolving baselines (RQ3).

For a practitioner in a similar setting: collect and preserve traceability metadata before investing in prediction, since this is where most of our predictive signal came from. Start instrumenting finer-grained, per-test, per-fault logging now, since without such logging, APFD and its variants cannot be computed directly from the current records. And do not assume more model search will fix weak predictions: our search across nine models and six resampling strategies mainly showed that model choice was not the bottleneck, the features were; a transparent weighted heuristic over factors already available may be a practical first step.

Future work includes trying the runtime-information step sketched for a later version, REMAP-style rule mining and test-case similarity [7], tempered by C4's finding; and pursuing the bottom-up, code-based factors that could not be computed this time, once traceability between code, requirements, and tests improves. Large language models are a further direction: the structured natural-language requirements could guide prioritization, provided that privacy and non-determinism are handled, for example with models deployed on premises.

Acknowledgments

This work was partially supported by Walloon Region CyberExcellence Project (Grant Agreement No. 2110186).

Declaration on Generative AI

During the preparation of this work, the authors used Claude in order to: grammar and spelling check, paraphrase and reword. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] G. Rothermel, R. H. Untch, C. Chu, M. J. Harrold, Prioritizing test cases for regression testing, IEEE Transactions on Software Engineering 27 (2001) 929–948. doi:10.1109/32.962562.

[2] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: A survey, Software Testing, Verification and Reliability 22 (2012) 67–120. doi:10.1002/stvr.430.

[3] M. Khatibsyarbini, M. A. Isa, D. N. A. Jawawi, R. Tumeng, Test case prioritization approaches in regression testing: A systematic literature review, Information and Software Technology 93 (2018) 74–93. doi:10.1016/j.infsof.2017.08.014.

[4] R. Pan, M. Bagherzadeh, T. A. Ghaleb, L. Briand, Test case selection and prioritization using machine learning: A systematic literature review, Empirical Software Engineering 27 (2022) 29. doi:10.1007/s10664-021-10066-6.

[5] CENELEC, EN 50128:2011 – railway applications: Communication, signalling and processing systems – software for railway control and protection systems, 2011. European Committee for Electrotechnical Standardization.

[6] R. K. Yin, Case Study Research and Applications: Design and Methods, 6th ed., Sage Publications, 2018.

[7] D. Pradhan, S. Wang, S. Ali, T. Yue, M. Liaaen, REMAP: Using rule mining and multi-objective search for dynamic test case prioritization, in: Proceedings of the 11th IEEE International Conference on Software Testing, Verification and Validation (ICST), 2018, pp. 46–57. doi:10.1109/ICST.2018.00015.

[8] D. Marijan, A. Gotlieb, S. Sen, Test case prioritization for continuous regression testing: An industrial case study, in: Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM), 2013, pp. 540–543. doi:10.1109/ICSM.2013.91.

[9] E. Engström, P. Runeson, A. Ljung, Improving regression testing transparency and efficiency with history-based prioritization – an industrial case study, in: Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST), 2011, pp. 367–376. doi:10.1109/ICST.2011.27.

[10] M. Mahdieh, S.-H. Mirian-Hosseinabadi, K. Etemadi, A. Nosrati, S. Jalali, Incorporating fault-proneness estimations into coverage-based test case prioritization methods, Information and Software Technology 121 (2020) 106269. doi:10.1016/j.infsof.2020.106269.

[11] H. Srikanth, C. Hettiarachchi, H. Do, Requirements based test prioritization using risk factors: An industrial study, Information and Software Technology 69 (2016) 71–83. doi:10.1016/j.infsof.2015.09.002.

[12] H. Srikanth, S. Banerjee, Improving test efficiency through system test prioritization, Journal of Systems and Software 85 (2012) 1176–1187. doi:10.1016/j.jss.2012.01.007.

[13] J. Anderson, S. Salem, H. Do, Striving for failure: An industrial case study about test failure prediction, in: Proceedings of the 37th International Conference on Software Engineering (ICSE), 2015, pp. 49–58. doi:10.1109/ICSE.2015.134.

[14] M. A. Khan, A. Azim, R. Liscano, K. Smith, Y.-K. Chang, G. Seferi, Q. Tauseef, On the effectiveness of feature selection techniques in the context of ml-based regression test prioritization, IEEE Access 12 (2024) 1–15. doi:10.1109/ACCESS.2024.3459656.

[15] R. Lachmann, S. Schulze, M. Nieke, C. Seidl, I. Schaefer, System-level test case prioritization using machine learning, in: Proceedings of the 15th IEEE International Conference on Machine Learning and Applications (ICMLA), 2016, pp. 361–368. doi:10.1109/ICMLA.2016.0065.

[16] T. B. Noor, H. Hemmati, Studying test case failure prediction for test case prioritization, in: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE), 2017, pp. 2–11. doi:10.1145/3127005.3127006.

[17] F. Palma, T. Abdou, A. Bener, J. Maidens, S. Liu, An improvement to test case failure prediction in the context of test case prioritization, in: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE), 2018, pp. 80–89. doi:10.1145/3273934.3273944.

[18] R. Carlson, H. Do, A. Denton, A clustering approach to improving test case prioritization: An industrial case study, in: Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 382–391. doi:10.1109/ICSM.2011.6080805.

[19] C. Magalhães, A. Mota, L. Momente, UI test case prioritization on an industrial setting: A search for the best criteria, Software Quality Journal 29 (2021) 381–403. doi:10.1007/s11219-021-09549-y.

[20] D. Di Nardo, N. Alshahwan, L. Briand, Y. Labiche, Coverage-based test case prioritisation: An industrial case study, in: Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), 2013, pp. 302–311. doi:10.1109/ICST.2013.27.

[21] S. Ulewicz, B. Vogel-Heuser, Industrially applicable system regression test prioritization in production automation, IEEE Transactions on Automation Science and Engineering 15 (2018) 1839–1851. doi:10.1109/TASE.2018.2810280.

[22] A. Sharif, D. Marijan, M. Liaaen, DeepOrder: Deep learning for test case prioritization in continuous integration testing,

in: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2021, pp. 525–534. doi:10.1109/ICSME52107.2021.00053.

[23] F. S. Ahmed, A. Majeed, T. A. Khan, S. N. Bhatti, Value-based cost-cognizant test case prioritization for regression testing, PLOS ONE 17 (2022) e0264972. doi:10.1371/journal.pone.0264972.

[24] S. Elbaum, A. G. Malishevsky, G. Rothermel, Test case prioritization: A family of empirical studies, IEEE Transactions on Software Engineering 28 (2002) 159–182. doi:10.1109/32.988497.

[25] H. Hemmati, Z. Fang, M. V. Mäntylä, Prioritizing manual test cases in traditional and rapid release environments, in: Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST), 2015, pp. 1–10. doi:10.1109/ICST.2015.7102602.

[26] H. Hemmati, Z. Fang, M. V. Mäntylä, B. Adams, Prioritizing manual test cases in rapid release environments, Software Testing, Verification and Reliability 27 (2017) e1609. doi:10.1002/stvr.1609.

[27] H. Kaur, H. S. Pannu, A. K. Malhi, A systematic review on imbalanced data challenges in machine learning: Applications and solutions, ACM Computing Surveys 52 (2019) 1–36. doi:10.1145/3343440.