On the Limits of Text-Only LLM-Based Test Case Prioritization in Industrial Black-Box Regression Testing

Alexandre Lima
Federal University of Amazonas
Manaus, Brazil
alexandre.lima@icomp.ufam.edu.br

Ronaldo Soares
Federal University of Amazonas
Manaus, Brazil
ronaldo.soares@icomp.ufam.edu.br

Yan Soares
Federal University of Amazonas
Manaus, Brazil
yan.soares@icomp.ufam.edu.br

Acauan Ribeiro
Federal University of Amazonas
Manaus, Brazil
acauan.ribeiro@ufrr.br

José Nascimento
Federal University of Amazonas
Manaus, Brazil
jcrn@icomp.ufam.edu.br

Eulanda Santos
Federal University of Amazonas
Manaus, Brazil
emsantos@icomp.ufam.edu.br

Bruno Gadelha
Federal University of Amazonas
Manaus, Brazil
bruno@ufam.icomp.edu.br

André Carvalho
Federal University of Amazonas
Manaus, Brazil
andre@ufam.icomp.edu.br

Abstract

Regression Test Case Prioritization (TCP) is critical in industrial black-box continuous software engineering, where large regression suites must be ordered under tight time budgets without source code or detailed execution traces. Pairwise ranking success with LLMs in information retrieval makes this setting look promising: perhaps test descriptions and commit messages are enough for useful prioritization. We test that transfer directly on 277 failing builds from a proprietary industrial dataset. The main evaluation is deliberately history-free and restricts the model to current-build test descriptions and commit messages. Within that text-only regime, we compare pointwise and pairwise LLM rankers, including a two-stage structured risk scorer, against lightweight text baselines, with TF-IDF as the primary reference and BM25 as a companion lexical baseline. In that regime, the best text-only LLM reaches mean APFD 0.5250, below TF-IDF (0.5410) and BM25 (0.5273) and only marginally above random, at substantially higher cost. Adding historical execution evidence improves the best LLM configuration to 0.6007, but simple history-only baselines such as LatestFail (0.6214) and MostFail (0.6098), as well as published history-aware predictors, remain stronger on the same benchmark. The contribution is therefore delimiting rather than promotional: recent LLM gains in textual ranking do not transfer directly to this tested industrial black-box TCP setting. Without history, the tested text-only LLM configurations are not competitive with cheap lexical baselines; with history, they improve, but fit better as auxiliary components than as primary prioritizers.

Keywords

Black-Box Testing, Test Case Prioritization, Large Language Models, Regression Testing

1 Introduction

Regression testing is a fundamental practice in modern software engineering because it provides confidence that recent changes have not broken existing functionality [1, 2]. Under continuous integration and frequent release cycles, however, executing every test in every cycle is often impractical. Test Case Prioritization (TCP) addresses this problem by reordering tests so that failures are revealed as early as possible [3]. The challenge is particularly acute in industrial environments with large system- and UI-level suites, where test execution is expensive and rapid feedback is operationally important.

In many such environments, TCP must be performed under black-box constraints [4]. Source code and execution traces may be unavailable, inaccessible, or too costly to integrate into the prioritization pipeline, leaving engineers with external artifacts such as test descriptions, commit messages, and sometimes historical execution data [5, 6]. Prior work has shown that these artifacts can support effective TCP, especially when methods can exploit richer historical behavior [6]. Yet those stronger signals are not always available or desirable to rely on, which makes the history-free setting an important and practically relevant regime to test explicitly.

That is why LLMs look attractive here. They can work directly over natural language and do not require task-specific supervised training. The appeal grew after Qin et al. [7] reported strong pairwise ranking results on textual documents. At first glance, TCP seems like a natural extension: the model could either score each test on its own or compare tests in pairs using the current-build evidence.

The analogy is loose, though. In document ranking, relevance is tied to an explicit textual query, and the useful signal is often visible in the text itself [7, 22]. Here the target is the chance that a test fails in the current build. That depends on runtime behavior, change impact, system state, and project-specific failure patterns that are only partly visible in test steps and commit messages. Two tests may read similarly and still exercise very different behaviors at runtime. Commit messages may also be too terse to say which tests are really at risk.

That mismatch matters in practice. If test descriptions and commit messages do not carry enough information about failure risk, better prompts or longer context will only go so far. Cost, latency,

reproducibility, and ease of integration matter too. For that reason, text-only LLM-based TCP in this setting needs to be judged against cheap lexical baselines—especially TF-IDF and BM25 in our study—and, when history is allowed, against stronger non-LLM methods that use historical behavior directly.

Throughout the paper, the measurable target is the observed *Pass*/*Fail* outcome. In practice, we ask whether a ranking moves failing executions earlier in the current build. We do not measure how early the ranking finds unique defect instances, because the dataset does not provide fault identifiers. We use *history-free* to mean that no prior execution outcomes are visible to the ranker, and *text-only* to mean that the available current-build evidence is limited to textual artifacts such as test descriptions and commit messages. A history-augmented setting may still use text, but it additionally exposes prior execution behavior.

The question in this paper is narrower than whether LLMs can rank tests at all. We ask whether text-only LLM rankers can serve as good primary prioritizers for industrial black-box regression TCP when restricted to current-build test descriptions and commit messages. To answer that, the main evaluation stays deliberately history-free: the model only sees test descriptions and commit messages. Within that setting, we compare pointwise and pairwise formulations and several evidence-construction strategies—RAG, Closest Cluster, All-Commits, and Changelog—against random and lightweight text baselines. TF-IDF is the main history-free reference, BM25 is the companion lexical retrieval baseline, and LSI/LDA are retained only as auxiliary text-only references. We then add a history-augmented analysis to see whether the missing signal comes from the ranking formulation or from the evidence itself.

The results are fairly direct. In the history-free setting, the best text-only LLM stays below TF-IDF and BM25; LSI, retained only as an auxiliary text-only reference, points in the same direction. Once historical execution evidence is added, the best LLM result rises to 0.6007 with the two-stage risk-scoring history-augmented RAG variant; the best direct pointwise history-augmented RAG run reaches 0.5949. That jump matters because it shows that behavior-oriented information is doing real work. Even so, simple history-only baselines from the companion re-analysis—LatestFail at 0.6214 and MostFail at 0.6098—already sit in the same range or above, and published learned or specialized predictors still sit higher, in the 0.64–0.69 range. Pointwise also ends up as the only practical LLM formulation among the configurations tested in this benchmark: it gives the best LLM results and is far cheaper than pairwise prompting.

To structure the empirical study, we ask three research questions:
RQ1. In a history-free industrial black-box setting, how competitive are LLM rankers relative to canonical lexical baselines such as TF-IDF and BM25?
RQ2. Still in the history-free regime, do ranking formulation (pointwise versus pairwise) and text-only evidence construction (RAG, Closest Cluster, All-Commits, and Changelog) materially change the outcome?
RQ3. Once historical execution evidence is available, do LLMs offer a practical advantage over simple history-based baselines and stronger specialized predictors?

The contribution of the paper is empirical rather than algorithmic. We do not introduce a new best-performing prioritizer. Instead, we show where the tested text-only LLM-based TCP setup is unconvincing and where history-augmented variants become more plausible. More specifically, we: (i) compare pointwise and pairwise LLM prioritization in a history-free industrial black-box setting where the model is restricted to current-build test descriptions and commit messages; (ii) show that stronger textual evidence engineering alone does not make these text-only LLM configurations competitive with lightweight lexical baselines such as TF-IDF and BM25; (iii) show that historical evidence helps, but also shifts the relevant comparison toward stronger non-LLM methods; and (iv) treat cost as part of the result rather than as an afterthought.

2 Related Work

Black-box TCP has long been motivated by the practical need to prioritize tests when source code and detailed runtime information are unavailable. A first relevant line of work studies history-free or weak-signal prioritization from non-code artifacts. String-based similarity over test descriptions showed early that useful signal can be extracted from textual representations alone [5]. FAST later emphasized that scalable similarity-based prioritization can remain effective without deep structural information [11]. These studies matter for our paper because they justify using cheap text baselines as the first comparison point in a history-free industrial setting.

A second line of work relies on historical behavior once it is available. Noor and Hemmati [9] prioritize tests through similarity over historical failure data, while Magalhães et al. [10] use explicit historical criteria such as past failures, number of executions, and recency in industrial UI testing. Cheng et al. [8] likewise show that lightweight history-aware rules remain strong baselines on long-running suites. Taken together, this line of work suggests that once historical behavior is available, it often carries signal that simple text alone does not.

A third line of work uses learned or specialized predictors over richer historical or structural representations. Beyond the industrial SentenceBERT-plus-classifier baseline of Hernandes et al. [6], prior TCP literature includes learning-to-rank and ranking-to-learn strategies for continuous integration [12], reinforcement-learning prioritization in RETECS [13], historical-feature neural models such as DeepOrder [14], end-to-end neural prioritization in TCP-Net [15], and specialized graph-based ranking in NodeRank [16]. Although these methods differ substantially, they share a common pattern: they move beyond current-build text and exploit richer learned, historical, or structural evidence. In our paper, they therefore serve as contextual references once history is allowed, rather than as like-for-like competitors in the history-free regime.

Our study is also motivated by recent evidence that LLMs can be strong rankers in text-centric tasks. Qin et al. [7] show that pairwise ranking prompting enables LLMs to perform well in document ranking. That result provides a legitimate starting point for our initial hypothesis. At the same time, it highlights the transfer risk: document ranking targets explicit textual relevance, whereas our benchmark observes the risk that a test fails in the current build. The target in TCP is therefore more latent, more behavior-dependent, and only partially visible in test descriptions and commit messages.

Relative to prior work, our contribution is one of delimitation. We do not present LLMs as replacements for history-aware heuristics or specialized predictors, and we do not assume that pairwise textual ranking transfers automatically from information retrieval to regression testing. Instead, we study a deliberately history-free regime, use TF-IDF and BM25 as the canonical low-cost baselines to beat, retain LSI/LDA as auxiliary text-only references, and then use a history-augmented variant as diagnostic evidence about where the missing signal actually resides.

3 Data Definition

The experiments use build-level records extracted from private repositories, and all textual artifacts are in English. Intuitively, each build is treated as one ranking problem: we have the commits associated with that build and the black-box tests executed for that build. Formally, for each build b, we keep two main inputs: the ordered list of commit messages

$$\mathcal{M}_b = (c_1, c_2, \ldots, c_m) \tag{1}$$

and the set of executed black-box test cases

$$\mathcal{T}_b = \{tc_1, tc_2, \ldots, tc_n\}. \tag{2}$$

Each test case is represented as $tc = \langle key, steps, result \rangle$, where *key* uniquely identifies the test case within the build, *steps* is the textual execution description concatenated into a single string, and $result \in \{\text{Pass}, \text{Fail}\}$ is the observed execution outcome. A build identifier links the commits and test executions that belong to the same product version.

The *steps* field comes from the industrial partner's operational black-box test artifacts, but the dataset available for this study does not include per-test provenance about how each description was originally authored or revised. We therefore treat these descriptions as the textual information available to an automated prioritizer in that environment, not as controlled natural-language specifications. Current-build commits provide the textual change evidence, test steps provide the black-box description of each candidate test, and execution outcomes are used only for evaluation or, in history-augmented references, as historical signals from earlier builds.

4 LLM Ranking Framework

Figure 1 shows the main design choice behind the framework: evidence construction and ranking are treated as separate modules. The evidence module decides what information about the current build is shown to the model, whereas the ranking module decides how the model output is converted into a final test order. This separation matters because the same evidence can be reused with different ranking formulations, and historical information enters the framework as additional evidence rather than as a different ranking algorithm.

Across all experiments, the pipeline has three common stages: (1) text pre-processing and codification, (2) evidence construction, and (3) LLM-based scoring or comparison. The first stage consolidates commits and test cases into a unified corpus. These texts are cleaned and transformed into dense vectors using sentence embeddings so that commits and test cases can be compared in a shared semantic space. The evidence-construction module then associates each test case with a contextual view of the build, and this evidence is combined with the test text to build an LLM prompt. In the text-only regime, we study four main evidence strategies, summarized in Figure 2:

- **RAG:** retrieves the top-k most similar commits to the test case;
- **Changelog:** generates a summarized changelog text from clusters of commits using LLMs;
- **Closest Cluster:** selects commits from clusters that are semantically closest to the test case;
- **All-Commits:** uses a build-level block of raw commit messages without retrieval or clustering.

In the final stage, the model either scores each test independently (pointwise) or compares pairs of tests (pairwise). We use these two formulations to separate two questions that are often conflated: whether textual evidence carries enough signal for TCP, and whether the extra cost of pairwise prompting is justified once operational constraints are considered.

4.1 Pre-processing and Codification

This step has two simple goals: clean the text and convert it into vectors that can be compared efficiently. First, the commit messages are cleaned using heuristics to remove irrelevant tokens (e.g., hashes, dates, URLs) while preserving semantic meaning.

Next, each commit message in $\mathcal{M}_b$ and each test case textual description are embedded into a shared 768-dimensional vector space using **all-MPNet-Base-V2**[1][17]. In practical terms, semantically similar commit messages and test descriptions are mapped to nearby points in the same space. The resulting vectors are then normalized for efficient comparison.

Embeddings from each build are indexed independently using FAISS (Equation 3). In practice, this lets us quickly retrieve the commits that are most semantically related to a given test while keeping memory use manageable.

$$C_b = \{\, \vec{c}_1, \vec{c}_2, \ldots, \vec{c}_m \,\} \longrightarrow \text{FAISSIndex}(C_b) \tag{3}$$

In this way, the vector representation acts as a bridge between maintenance artifacts and test cases, enabling semantic retrieval and clustering even in the absence of source code.

4.2 Evidence Construction

Given a test case tc represented by its embedding and the set of commit embeddings $C_b = \{\vec{c}_1, \ldots, \vec{c}_m\}$ for the current build, the goal of this phase is to decide which information should be shown to the LLM as context. In the history-free regime, this evidence is built only from current-build commits. In the history-augmented regime, the same current-build evidence is supplemented with summaries of earlier executions of the same test and of semantically similar tests, always restricted to builds prior to the current one. Throughout the paper, we refer to this configuration as *history-augmented RAG*: the augmentation is the addition of historical behavioral summaries on top of ordinary current-build RAG evidence, not a second retrieval mechanism over the same commit corpus.

[1] https://huggingface.co/sentence-transformers/all-mpnet-base-v2

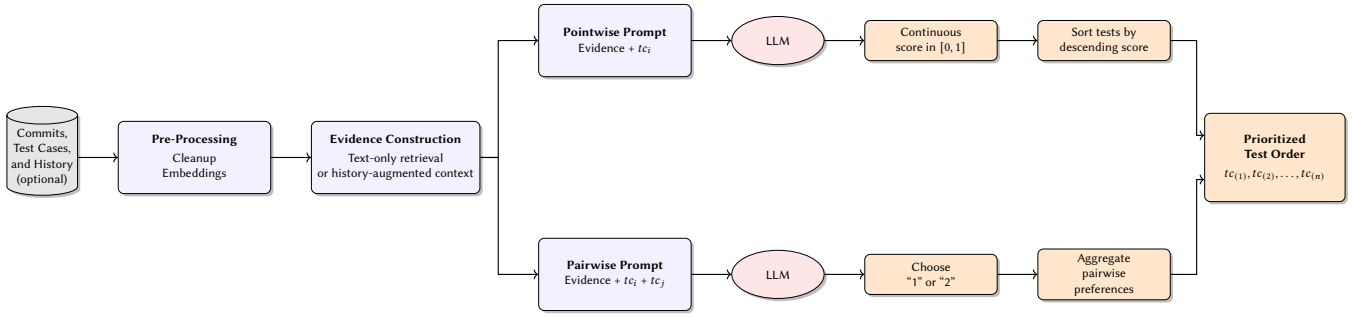

Figure 1: Unified workflow of the pointwise and pairwise LLM ranking formulations. The pre-processing and evidence-construction stages are shared, while prompt structure and ranking aggregation differ in the final stage.

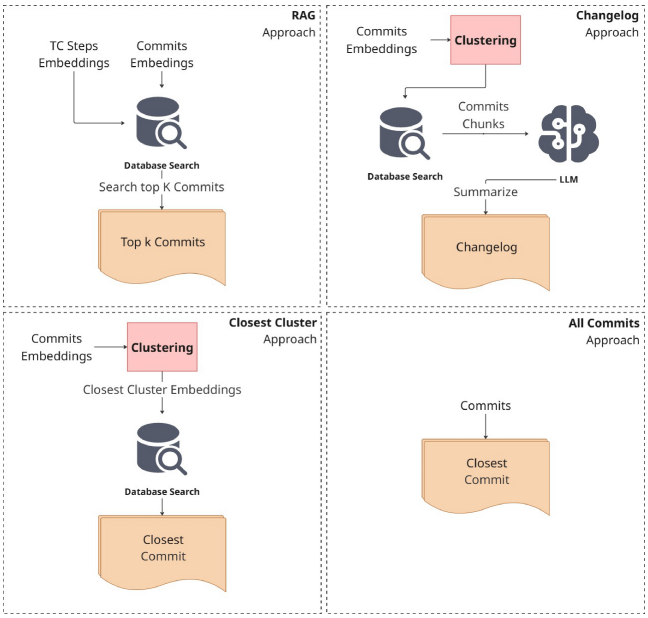

Figure 2: Evidence-selection strategies used in the text-only regime

The four base strategies differ mainly in how they construct the context block. RAG and Closest Cluster build context specifically for the focal test, whereas Changelog and All-Commits reuse a build-level context across tests. This difference changes what the model reads, but not how the ranking step operates: the same evidence can later be consumed by either pointwise or pairwise ranking.

4.2.1 Selection by RAG. RAG [18] is the simplest strategy for building test-specific context. For each test case, it retrieves only the subset of commit messages that is most semantically related to that test, instead of exposing the model to the whole build history.

Given a test case embedding $\vec{t}_i$ and the current build b, we query the FAISS [19] index built over C_b to retrieve the top-k commit embeddings closest in semantic space. Formally, this search returns the ordered list:

$$\left(\vec{c}_{(1)}, \ldots, \vec{c}_{(k)}\right) = \text{FAISS\_search}\left(\vec{t}_i,\ k\right) \tag{4}$$

where each $\vec{c}_{(j)}$ denotes the j-th nearest commit embedding to $\vec{t}_i$ among the commits of build b. We then retrieve the corresponding commit texts, remove duplicates, and concatenate the resulting messages into the prompt. Thus, the LLM receives the textual commit evidence itself, not the embeddings.

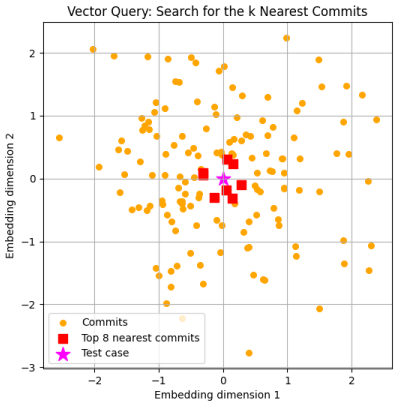

Figure 3: Example of the k nearest neighboring search performed in RAG for $k = 8$.

Figure 3 illustrates how only the commits that lie in the immediate semantic neighborhood of a test case are selected in the shared embedding space. This retrieval is performed independently for each test, so the prompt context changes with the focal test case. The value of k was tuned on the validation split. In practice, very small values often missed relevant context, whereas much larger values increased prompt cost without clear benefit.

4.2.2 Closest Cluster. Closest Cluster is another strategy for building test-specific context. Instead of retrieving a fixed number of commits, it first groups the build commits with DBSCAN [20]. For each test case, we then select the cluster whose centroid is closest to the test embedding and use all commits in that cluster as evidence. Intuitively, this strategy tries to preserve a coherent local neighborhood of related changes instead of forcing the context into a fixed top-k budget.

4.2.3 Changelog. Changelog is a build-level summarization strategy. Instead of selecting commits separately for each test, it produces a single build-level summary that is reused across the ranking process. To do so, we first cluster the commit messages by semantic similarity and then ask an LLM to summarize each cluster in natural language. These summaries are concatenated into one markdown-formatted changelog block that captures the main changes in the build while keeping the context shorter than the raw commit list. This strategy therefore has an extra build-level summarization step before ranking; when summaries are regenerated rather than reused from cache, this cost is separate from the per-test ranking calls. One synthetic example of the changelog is provided below.

```
**Consolidated Summary**

The changes focus on integrating new features,
↪ optimizing existing modules, and fixing known
↪ issues.

* **Component Integration**:
...
* **Scheduler Adjustments**:
...
* **Permission Checks**:
...
* **Feature Enhancements**:
...
These updates are intended to improve ...
---
**Key Changes Highlighted:**
...
**Relevant Context:**
...
**Additional Notes**
* Corrected ...
* Updated ...
* Refined ...
```

During ranking, this same changelog block is shown together with the focal test case in pointwise mode or with the focal test pair in pairwise mode. The main benefit is efficiency: the model sees a compact description of the build instead of a long list of raw commits. A second benefit is consistency, because every test in the same build is judged against the same high-level summary of recent changes.

4.2.4 All-Commits Baseline. All-Commits is the simplest build-level baseline. It provides one build-level block containing the raw commit messages that fit the prompt budget, without retrieval or clustering. It therefore acts as a minimal-processing reference point for judging whether more selective evidence construction is actually helpful.

4.3 Ranking Formulations

Once evidence has been built, the ranking module converts the LLM output into an execution order. Pointwise and pairwise share the same evidence strategies; they differ only in the question asked to the model and in how the answers are aggregated. Within the pointwise family, our framework supports both a direct continuous-token scorer and a two-stage structured risk-scoring variant.

The history-augmented regime keeps the same ranking procedures and current-build evidence strategies, but appends two behavior-oriented signals to the prompt: a summary of earlier outcomes for the focal test case and a summary derived from historically observed, semantically related tests. In other words, commit retrieval still uses the same current-build RAG evidence as before; what changes is that the prompt is augmented with historical behavior summaries. We use this variant to test whether the missing signal lies in the absence of behavioral history, not to introduce a different ranking algorithm.

4.3.1 Pointwise Scoring. In the pointwise formulation, each test case is scored independently. Conceptually, the model is asked a simple question for each test: given this test and its context, how likely is it to fail in the current build? The model returns a risk score, which makes pointwise ranking the most direct and scalable alternative in our study.

Formally, let $tc_i \in \mathcal{T}_b$ be a test case in build b, and let $E(tc_i, b)$ denote the evidence attached to that test under the selected evidence-construction strategy. The pointwise scorer produces a continuous value

$$s(tc_i) = \text{score}(tc_i, E(tc_i, b)) \in [0, 1] \tag{5}$$

that we interpret as an estimated failure risk. The main reported pointwise runs use a continuous-token configuration. There, the prompt asks the model to produce a single next token corresponding to *Pass* or *Fail* and to return the top log-probabilities for that token. Informally, the score becomes larger when the model assigns more probability mass to *Fail*-like tokens than to *Pass*-like tokens. Let $\mathcal{V}_{\text{fail}}(tc_i)$ and $\mathcal{V}_{\text{pass}}(tc_i)$ denote the subsets of returned candidate tokens whose normalized surface form starts with "fail" and "pass", respectively, and let $\ell(v)$ be the returned log-probability of token v. The score is then computed as

$$s(tc_i) = \frac{\sum_{v \in \mathcal{V}_{\text{fail}}(tc_i)} e^{\ell(v)}}{\sum_{v \in \mathcal{V}_{\text{fail}}(tc_i)} e^{\ell(v)} + \sum_{v \in \mathcal{V}_{\text{pass}}(tc_i)} e^{\ell(v)}}. \tag{6}$$

This score therefore uses only the first generated token and its returned top candidates, not the probability of the full response string.

The framework also supports a more explicit pointwise variant, denoted *two-stage structured risk scoring*. Here the evidence builder first creates a base prompt $P(tc_i, b)$ for the focal test. A specialized scorer then transforms that prompt into a structured JSON response containing a failure-risk score in $[0, 1]$, a confidence value, a predicted *Pass*/*Fail* label, and a short rationale. The ranking score is obtained from the returned risk estimate after validation and normalization of the fields.

To reduce the influence of uncertain predictions, the implementation applies a confidence gate with threshold $\tau = 0.6$: when the reported confidence is below τ, the score is mapped to the neutral value 0.5 before ranking. If all tests in a build collapse to exactly the same score, the system issues an additional tie-break pass to request

more discriminative values. This variant preserves the same pointwise interface while replacing token-level probability extraction with an explicit structured risk estimate.

After scoring all tests, the final ranking is obtained by sorting them in descending order of this score:

$$\pi_b = \text{argsort}_{tc_i \in \mathcal{T}_b} s(tc_i). \tag{7}$$

Depending on the evidence strategy, pointwise may receive a context block tailored to the focal test or a single build-level context reused across tests. Operationally, pointwise requires one primary LLM call per test case and therefore scales linearly with suite size, with occasional extra calls only when the two-stage tie-break mechanism is needed. Thus, pointwise is not used merely as a cheaper proxy for pairwise comparison, but as a first-class alternative that requests a different signal from the model.

4.3.2 Pairwise Comparison. In the pairwise formulation, an LLM receives two test cases and decides which one is more likely to uncover a failure in the current build. This formulation complements pointwise scoring by eliciting a relative preference rather than an independent risk estimate. The prompt is composed of the steps corresponding to each tc and the evidence selected according to the methods detailed in the previous subsection.

Formally, let $tc_i, tc_j \in \mathcal{T}_b$ be two test cases from the same build. The pairwise decision function is implemented by an instruction-tuned LLM:

$$\text{prioritize}(tc_i, tc_j) = \begin{cases} 1, & \text{if } tc_i \text{ must be prioritized} \\ 2, & \text{if } tc_j \text{ must be prioritized} \end{cases} \tag{8}$$

As previously mentioned, the prompt structure depends on the chosen evidence strategy. With RAG and Closest Cluster, each candidate test is paired with its own test-specific commit context. With Changelog and All-Commits, both candidates share the same build-level context block. The model returns "1" or "2"; if an error or unexpected response occurs, the system retries up to four times with a short delay.

In the experiments reported here, pairwise ranking is aggregated with an all-vs-all scheme: every unordered pair of tests in the build is compared once, and each win contributes one point to the winning test. The final build-level order then ranks tests by number of pairwise wins. Because this requires repeated comparisons within the same build, pairwise prompting is operationally much more expensive than pointwise scoring. In this paper, we therefore use pairwise ranking mainly as a diagnostic formulation rather than as the main operational candidate. For the final benchmark, we do not pursue an exhaustive grid of pairwise variants. Instead, we keep one strong representative pairwise configuration—test-specific RAG evidence with all-vs-all aggregation—because the purpose of the pairwise arm is to test transfer of the ranking formulation itself from document ranking to black-box TCP. Once that representative configuration proved much more expensive and did not outperform lightweight lexical baselines, further expansion of the pairwise family on the final benchmark would have increased computational cost without changing the main conclusion of the study.

In the next section we describe the experimental design and present the results.

5 Experiments and Results

Our experiments were conducted in two phases. First, we used a separate validation partition to adjust evidence-selection parameters, refine prompts, and calibrate the ranking formulations. No final results reported in this paper come from that phase. Second, we performed the main evaluation on a chronologically later test split, using fixed settings.

5.1 Experimental Setup

The dataset used in our experiments was provided by an industrial partner and is proprietary. It contains English commit messages, build identifiers, and black-box test artifacts centered on test steps and execution outcomes. The chronologically earlier training split, used only by baselines that require history, contains 69,169 test-execution records, 3,187 builds, 1,654 failing executions, and 625 builds with at least one failure. The chronologically later test split contains 31,333 test-execution records, 1,365 builds, 924 failing executions, and 277 builds with at least one failure. All scripts rely on a common data loader that normalizes the commit field from string form to a list of commits and disambiguates repeated test-case keys within a build when necessary.

For the final benchmark used in this paper, we restricted the evaluation to the 277 builds in the test split that contain at least one failing test execution. This restriction is also operationally sensible: in cycles with no failing tests, running inference is unnecessary because there is no failure to reveal earlier, so the ranking does not change the APFD-based comparison among methods. This filtered benchmark contains 12,070 test executions and 1,708 distinct test cases, with an average of 43.6 executed tests per build and 49.6 distinct commits per build. Baselines and LLM methods also skip builds with fewer than two test cases.

When historical evidence is used, it is extracted only from data preceding the current build, thereby preserving temporal validity. In the history-free regime, the same benchmark is evaluated while deliberately hiding all historical execution information from the LLM.

Our primary experiments use a local quantized Gemma 3 27B instruction model [21]. We use a local LLM to keep proprietary industrial artifacts on-premises and to ensure controlled inference throughout the benchmark. On the separate validation partition, we also checked two additional local instruction-tuned models (Llama 3.1 8B Q8 and Qwen 3 14B Q3). These checks showed the same qualitative pattern as Gemma across the tested settings, but they were used only as validation-phase sanity checks rather than as a full quantitative model-family study. We therefore keep Gemma 3 27B as the representative configuration for the final benchmark and restrict the claims accordingly. Unless otherwise stated, the reported pointwise results correspond to the continuous-token setting, in which the model is queried for a single next token together with its top log-probabilities under fixed inference settings. The two-stage structured risk scorer uses a fixed structured-output scoring prompt with $temperature = 0.0$. Because local quantized inference

and serving libraries may still introduce implementation-level variability, we do not treat these settings as a guarantee of bit-for-bit deterministic reproduction. All experiments were executed on a local GPU workstation.

Because the industrial records are subject to confidentiality restrictions, they cannot be publicly released. The paper therefore documents the prompt structure, evidence construction, scoring procedures, temporal split, and inference settings needed to reimplement the study on compatible data.

5.2 Comparison Baselines and Reference Methods

Besides the LLM configurations, the evaluation includes several non-LLM references plus one human contextual reference. For clarity, we group them by the type of information they use. First, Random and Perfect Ranking serve as lower- and upper-bound anchors. The random baseline estimates expected APFD by averaging 100,000 random permutations per build.

Consistent with Section 3, we use *steps* throughout this section to denote the textual execution description of a test case.

Second, we keep two canonical lightweight history-free baselines based only on current-build text, following prior text-similarity TCP work [5, 11]: TF-IDF and BM25. In our adaptation, TF-IDF represents the aggregated commit text for the build and each test case's *steps* field in the same weighted term space, then ranks tests by textual similarity. BM25 [22] uses the same inputs but scores them with a lexical retrieval function; in the reproduced benchmark, BM25 indices were available for all 277 failing builds. These are the primary text-only baselines that the LLM runs must beat to justify their extra cost.

Third, we retain two additional text-only references from the earlier framing. LSI [23] compares tests and commits after projection into a reduced latent space, whereas LDA [24] compares the topic distribution of the aggregated commit query with that of each test case. We report them as auxiliary text-only references rather than as canonical baselines.

Fourth, when historical evidence is allowed, we report a Similarity History baseline inspired by Noor and Hemmati [9]. Each current test case is scored by the average Jaccard similarity between its *steps* text and the set of unique historically failing *steps* texts extracted from the training split; in the reproduced benchmark, this historical set contains 483 distinct historically failing test cases.

Fifth, once history is allowed, we also report a companion reanalysis of simple history-only rules on the same 277-build benchmark as contextual references. In line with the historical criteria studied by Magalhães et al. [10], MostFail ranks tests by the cumulative number of prior failures, LatestFail by the recency of the last prior failure, and LatestTrans by the recency of the last prior *Pass*/*Fail* transition. These baselines matter because they test a strong practical alternative to LLM augmentation: once temporal execution history is available, very simple recency- and frequency-based rules may already be highly competitive. For space, Table 1 highlights the strongest failure-history variant, LatestFail, while the text also discusses MostFail and LatestTrans.

Sixth, we report a local SBERT classifier baseline [6], which reaches mean APFD 0.5950 on the filtered benchmark, together with published trained or otherwise specialized predictive references that we use only as contextual comparison points once richer signals are allowed. RETECS [13] is a reinforcement-learning prioritizer for adaptive continuous-integration settings. DeepOrder [14] uses a lightweight neural model over historical execution features. TCP-Net [15] is an end-to-end deep neural prioritizer with periodic retraining and automated configuration. NodeRank [16] is a graph- and mutation-based ensemble ranker. These methods are not like-for-like baselines for our history-free setting; rather, they indicate the competitive level already reached by stronger learned or otherwise specialized alternatives. Table 1 reports the local SBERT classifier baseline explicitly and summarizes the published range induced by RETECS, NodeRank, TCP-Net, and DeepOrder.

Finally, we also report a manual ordering produced by human testers in the same operational setting. On the filtered 277-build benchmark used in this paper, that human-planned ordering reaches mean APFD 0.5698. We keep it as contextual information rather than as a direct automated baseline because it represents human test-plan construction and may reflect tacit system knowledge, verbal communication, or other operational signals unavailable to the automated history-free ranking methods.

5.3 Evaluation Metric

The main evaluation metric considered in this work was APFD [3]. After ordering $\mathcal{T}_b$, we obtain the vector of observed execution outcomes in the same order as the ranking:

$$\mathbf{r} = (r_1, \ldots, r_n), \quad r_k \in \{\text{Pass}, \text{Fail}\} \tag{9}$$

Because the dataset records execution outcomes but not unique fault identifiers, we operationalize APFD at the failing-execution level. Concretely, the implementation receives the ranked sequence of observed *Pass*/*Fail* labels and treats each *Fail* occurrence as one detected event in the ordering. Let F be the number of failing test executions in the ranked build and let $\text{pos}(f)$ be the position of the f-th failing execution in the ranking. Under this operational definition, *Average Percentage of Faults Detected (APFD)* is computed as:

$$\text{APFD} = 1 - \frac{\sum_{f=1}^{F} \text{pos}(f)}{F \cdot n} + \frac{1}{2n}. \tag{10}$$

Values of APFD closer to 1 indicate that failing executions are revealed earlier in the testing process. If multiple failing tests are caused by the same underlying defect, they are still counted separately because the dataset does not provide fault identifiers. We retain the standard name APFD for comparability with prior TCP literature, but in this benchmark the measured target is early exposure of failing executions rather than unique-defect detection.

In this study, APFD is computed independently for each build and the final score is reported as the simple mean over the 277 evaluated builds. Only *Pass* and *Fail* executions are considered in the ranking and metric calculation; other execution statuses are discarded. Builds with no failing tests are excluded from the APFD analysis because, in those cycles, no ranking can change how early failures are exposed.

5.4 Results

5.4.1 Overall Prioritization Performance. Table 1 groups methods by the evidence they use. The main like-for-like comparison is the history-free primary block, where the text-only LLM is compared with TF-IDF and BM25. LSI and LDA are auxiliary text-only references, while history-augmented methods, trained predictors, and manual ordering are contextual because they use additional behavioral, structural, or human planning information.

Table 1: Mean APFD on the 277-build benchmark, grouped by evidence regime and comparison role (higher is better)

Approach	Mean APFD
Anchors on the filtered benchmark	
Perfect Ranking	0.9550
Random Baseline	0.5000
History-free primary comparison	
TF-IDF	0.5410
BM25	0.5273
LLM: pointwise + Changelog (text-only)	0.5250
History-free auxiliary text references	
LSI	0.5407
LDA	0.4864
History-augmented diagnostic comparisons	
Similarity History	0.5156
LatestFail	0.6214
LLM: pointwise + History-augmented RAG	0.5949
LLM: two-stage risk score + History-augmented RAG	0.6007
History-augmented contextual references	
SBERT classifier baseline	0.5950
Manual ordering (human planners)	0.5698
Published trained/specialized predictors	0.6406–0.6890

The history-free primary block is the main like-for-like comparison. LSI/LDA are auxiliary text-only references; history-augmented rows are diagnostic or contextual once prior execution data is allowed. The published trained/specialized range summarizes RETECS, NodeRank, TCP-Net, and DeepOrder.

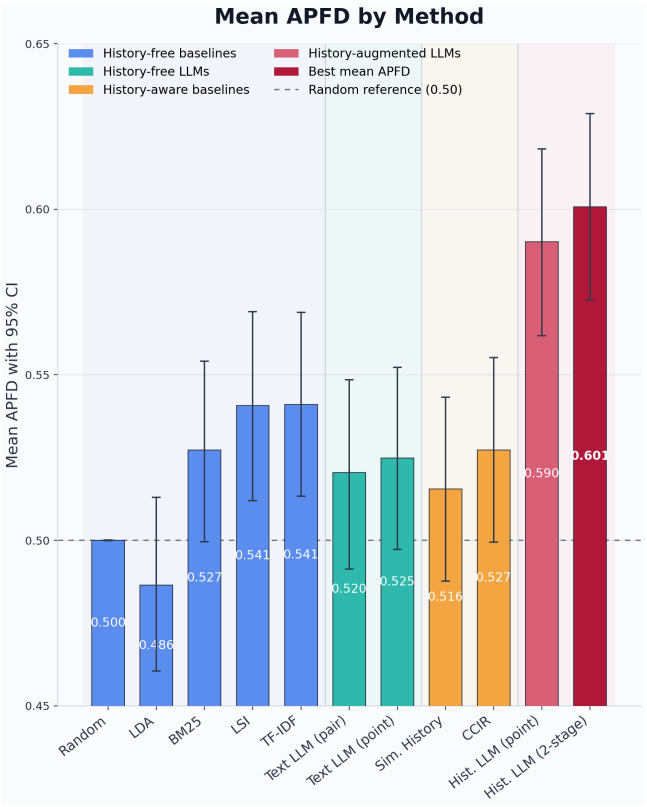

Figure 4: Mean APFD with bootstrap 95% confidence intervals for the selected methods with aligned per-build results on the 277-build benchmark.

In the history-free primary block, the best text-only LLM run is pointwise Changelog at 0.5250, below TF-IDF (0.5410) and BM25 (0.5273) and only modestly above random. The auxiliary text-only references point in the same direction: LSI is close to TF-IDF, while LDA falls below random. Thus, stronger text-only evidence construction does not make the tested LLM competitive. **Answer to RQ1:** in the history-free regime, LLM prioritization is not competitive with the canonical lexical baselines.

The history-augmented rows show that prior execution data helps: the two-stage history-augmented RAG run reaches 0.6007, and direct pointwise history-augmented RAG reaches 0.5949. However, that improvement does not create a practical advantage. LatestFail reaches 0.6214, and the companion re-analysis also reports MostFail at 0.6098 and LatestTrans at 0.6061, so simple recency and frequency rules match or exceed the best history-augmented LLM. Manual planning reaches 0.5698, and the published learned or specialized references remain higher, from 0.6406 to 0.6890. **Answer to RQ3:** historical evidence helps the LLM, but it does not create a practical advantage over simple history-based baselines, much less over stronger specialized predictors.

5.4.2 Paired Statistical Analysis. For the paired analysis, we aligned the same 277 builds across the selected methods and computed bootstrap 95% confidence intervals together with two-sided paired Wilcoxon signed-rank tests on per-build APFD. Figure 4 gives a compact family-level view, and Table 2 lists the key paired contrasts for the main history-free and strongest history-augmented configurations.

In the history-free setting, the intervals of TF-IDF, BM25, the auxiliary LSI reference, and the two text-only LLM variants overlap heavily. Against the two canonical lexical baselines, the best text-only pointwise LLM is not statistically better than TF-IDF ($\Delta = -0.016$, $p = 0.425$) or BM25 ($\Delta = -0.002$, $p = 0.756$) on the aligned builds. The auxiliary LSI comparison tells the same story ($\Delta = -0.016$, $p = 0.520$). Even against Random, the gain is small and not conventionally reliable ($\Delta = 0.025$, 95% CI $[-0.002, 0.052]$, $p = 0.079$). The effect sizes are similarly small, ranging roughly from -0.06 to 0.13.

The text-only pairwise run does not change that picture. It remains statistically indistinguishable from TF-IDF ($\Delta = -0.021$, $p = 0.320$) and BM25 ($\Delta = -0.007$, $p = 0.727$), while costing about 11.1× as much per build as the best text-only pointwise configuration. Once history is added, the strongest two-stage history-augmented RAG run clearly beats Similarity History with aligned per-build APFD. The mean paired gain is 0.085 (95% bootstrap CI $[0.045, 0.126]$, $p < 0.001$). At the same time, the same paired analysis shows no reliable gain over the simplest failure-history heuristics. Against LatestFail, the mean difference is -0.021 (95%

bootstrap CI $[-0.051, 0.010]$, $p = 0.195$); against MostFail, it is -0.009 (95% bootstrap CI $[-0.038, 0.020]$, $p = 0.466$); and against LatestTrans, it remains effectively tied at -0.005 (95% bootstrap CI $[-0.036, 0.026]$, $p = 0.836$). Historical evidence therefore helps the LLM, but not enough to open a clear paired-performance gap over simple recency/frequency rules.

Table 2: Selected paired statistical comparisons on APFD over the aligned 277-build benchmark

Comparison	Δ mean	95% CI	p	W/T/L
LLM text (pointwise) – TF-IDF	-0.016	[-0.056, 0.023]	0.425	109/50/118
LLM text (pointwise) – BM25	-0.002	[-0.039, 0.034]	0.756	107/49/121
LLM text (pointwise) – LSI	-0.016	[-0.054, 0.022]	0.520	110/50/117
LLM text (pointwise) – Random	0.025	[-0.002, 0.052]	0.079	138/29/110
LLM + history (two-stage) – LatestFail	-0.021	[-0.051, 0.010]	0.195	100/68/109
LLM + history (two-stage) – MostFail	-0.009	[-0.038, 0.020]	0.466	96/77/104
LLM + history (two-stage) – Sim. Hist.	0.085	[0.045, 0.126]	<0.001	145/43/89

Δ is the mean paired APFD difference (left minus right). Confidence intervals are bootstrap 95% intervals over builds, p-values come from two-sided paired Wilcoxon signed-rank tests, and W/T/L reports builds where the left method achieves higher APFD, equal APFD, or lower APFD.

5.4.3 Effect of Text-Only Evidence Strategies. Changing the text-only evidence builder does not change the story very much. Across RAG, Closest Cluster, All-Commits, and Changelog, the scores stay in a narrow band near random, below TF-IDF, and effectively tied with or below BM25. The auxiliary LSI reference also remains above them. The context builder clearly matters a little, but not enough to change the ranking among method families. **Answer to RQ2:** within the history-free regime, neither changing the ranking formulation (pointwise versus pairwise) nor changing the text-only evidence builder materially changes the conclusion.

Pointwise ends up being the only LLM formulation that looks operationally plausible in this benchmark. It produces the best LLM results and does so at far lower cost. Pairwise remains useful mainly as a diagnostic comparison.

5.4.4 Computational Cost. The cost numbers reinforce the same reading. Pairwise text-only ranking is by far the most expensive configuration at about 17.7 minutes per build and 81.8 hours over the full benchmark. Pointwise text-only ranking requires about 1.6 minutes per build and 7.4 hours in total, while the strongest pointwise + history configuration—the two-stage history-augmented RAG run—stays in the same operational range at about 1.3 minutes per build and 6.2 hours over the full benchmark. These values describe the measured ranking runs; Changelog-style evidence may also require an additional build-level summarization call when cached summaries are not already available. Whatever one thinks about the APFD gains, pointwise is the only LLM formulation here that looks easy to justify operationally.

5.5 Threats to Validity

Construct validity. APFD is computed over observed *Pass*/*Fail* executions rather than unique defect instances, because the dataset does not provide fault identifiers. The study therefore measures how early failing executions are surfaced, which may reward methods that move several correlated failures from the same underlying fault earlier in the order. Without fault identifiers, we cannot quantify this bias directly or report defect-level metrics such as APFD-c.

Table 3: Representative runtime cost of the main LLM settings

Setting	Avg. per build	Total runtime
Pairwise text-only	17.7 min	81.8 h
Pointwise text-only	1.6 min	7.4 h
Pointwise + history (best)	1.3 min	6.2 h

Internal validity. LLM results may depend on prompt wording, model family, quantization, inference settings, and evidence construction. We mitigate this risk by checking additional local instruction-tuned models on the validation partition and by comparing pointwise and pairwise formulations, multiple text-only evidence builders, and a separate history-augmented variant on the same benchmark, although a broader prompt/model grid, repeated inference runs, or frontier hosted models could still change absolute APFD values. The main negative result should therefore be read as evidence about the tested local text-only setup rather than as a universal statement about all LLMs.

Conclusion validity. Not all comparisons are like-for-like. The main claim is restricted to the history-free, text-only regime in which the model sees only current-build test descriptions and commit messages, whereas the history-augmented discussion broadens the comparison to methods that use richer signals.

External validity. The study uses one proprietary industrial black-box dataset with specific artifacts and temporal dynamics, and the primary experiments rely on one local Gemma-based model configuration. The relative ranking of methods may differ in projects with richer commit messages, different test descriptions, more informative historical labels, different CI/CD latency constraints, or different model families, so replication on additional industrial datasets is needed.

6 Discussion

Pairwise ranking prompting works well in information retrieval because the target is explicit textual relevance: candidate documents are compared against a query whose semantics are expressed in text [7]. Our benchmark asks a different question: which test is more likely to fail in the current build? That target depends on runtime behavior, change impact, system state, and project-specific failure dynamics that test descriptions and commit messages expose only partially. Semantically plausible comparisons therefore do not necessarily translate into better failure-first rankings.

The history-free results suggest that the limiting factor is evidence rather than ranking formulation. Pointwise and pairwise text-only variants remain close to one another and behind TF-IDF and BM25, while RAG, Closest Cluster, All-Commits, and Changelog move scores only modestly. One plausible explanation is that the available test descriptions and commit messages vary in granularity and may not consistently expose the behavioral links that determine failure risk. Thus, text-only evidence engineering by itself does not make the tested LLM configurations convincing primary prioritizers in this industrial setting.

The history-augmented runs reinforce that interpretation. Once execution history for the same and related tests is included, performance improves, showing that behavioral history contains signal that text alone does not expose. But allowing history also changes the relevant comparison: the LLM must then compete with history-aware methods, and simple rules such as LatestFail (0.6214), MostFail (0.6098), and LatestTrans (0.6061) already match or exceed the history-augmented LLM. Cost further narrows the recommendation. Pairwise prompting is too expensive for the observed gains, and even pointwise ranking remains costlier than lexical baselines that do better in the history-free regime. The most defensible role for LLMs here is therefore auxiliary—for explanation, evidence summarization, or selective escalation—rather than primary prioritization.

7 Conclusion

This paper asked whether text-only LLM rankers based on current-build test descriptions and commit messages can serve as primary prioritizers in industrial black-box regression testing. On 277 failing builds from a proprietary industrial dataset, the answer is negative for the history-free setting studied here.

Text-only LLM prioritization does not beat the canonical lightweight baselines TF-IDF and BM25, and the extra cost is substantial. The paired analysis supports the same conclusion: the text-only pointwise LLM shows no reliable gain over TF-IDF or BM25, the auxiliary LSI comparison points in the same direction, and the text-only pairwise variant offers no compensating benefit for its much higher runtime.

Historical execution evidence improves the LLM results: the strongest run reaches 0.6007 with the two-stage risk-scoring history-augmented RAG variant, and direct pointwise history-augmented RAG reaches 0.5949. That shows that behavioral history adds useful signal, but it does not make the tested LLM configurations the best option. Simple history-only baselines such as LatestFail (0.6214), MostFail (0.6098), and LatestTrans (0.6061) are already in the same range or above, and published learned or specialized predictors still sit higher.

The implication is not that LLMs are useless for TCP, but that success in text-centric ranking does not transfer automatically to black-box TCP when the model sees only current-build textual artifacts. In this benchmark, the model is estimating failure risk from incomplete evidence, not ranking explicit textual relevance. Practitioners should therefore benchmark LLM-based prioritization against cheap lexical baselines and stronger history-aware methods before treating it as a primary solution. Future work should test narrower hybrid uses, such as explanation, evidence summarization, or selective escalation when cheaper rankers are uncertain.

Artifact Availability

In compliance with confidentiality policies and data protection agreements, datasets used in this study cannot be publicly disclosed, as they contain sensitive information and internal processes, including mobile device validation flows, file naming conventions, and proprietary commands. Access to these datasets has been strictly limited to internal research and development purposes, in accordance with the terms set forth in confidentiality agreements (NDAs).

Acknowledgements

This work was supported by the following Brazilian agencies: the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior-Brasil (CAPES-PROEX)- Finance Code 001, the National Council for Scientific and Technological Development (CNPq), and Amazonas State Research Support Foundation- FAPEAM- through the POSGRAD project 2024/2025, and by Motorola Mobility Comércio de Produtos Eletrônicos Ltda, under the terms of Federal Law No. 8.387/1991, through agreement No. 02/2021 with ICOMP/UFAM.

References

[1] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. 2012. A static approach to prioritizing JUnit test cases. *IEEE Transactions on Software Engineering* 38, 6 (2012), 1258–1275.

[2] S. Wang, J. Nam, and L. Tan. 2017. QTEP: Quality-aware test case prioritization. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. 523–534.

[3] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 27, 10 (2001), 929–948.

[4] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon. 2016. Comparing white-box and black-box test prioritization. In *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. 523–534.

[5] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran. 2012. Prioritizing test cases with string distances. *Automated Software Engineering* 19 (2012), 65–95.

[6] V. Hernandes, A. Carvalho, E. Santos, Y. Soares, H. Oliveira, A. Barros, R. Soares, A. Lima, R. Ferreira, G. Martins, L. Carvalho, N. Assumpção, J. Nascimento, E. Collins, S. Ascate, and M. Souza. 2025. A method for regression testing plan ordering for non-automated executions in black-box testing. In *Anais do XXVIII Congresso Ibero-Americano em Engenharia de Software (CIbSE 2025)*. 120–134.

[7] Z. Qin, W. Wang, Y. Li, Y. Wang, M. Wang, and J. Wang. 2024. Large language models are effective text rankers with pairwise ranking prompting. In *Findings of the Association for Computational Linguistics: NAACL 2024*. 1504–1518.

[8] R. Cheng, S. Wang, R. Jabbarvand, and D. Marinov. 2024. Revisiting test-case prioritization on long-running test suites. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. 614–626.

[9] T. B. Noor and H. Hemmati. 2015. A similarity-based approach for test case prioritization using historical failure data. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE 2015)*. 58–68.

[10] C. Magalhães, A. Mota, and L. Momente. 2021. UI test case prioritization on an industrial setting: A search for the best criteria. *Software Quality Journal* 29 (2021), 381–403.

[11] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino. 2018. FAST approaches to scalable similarity-based test case prioritization. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*. 222–232.

[12] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo. 2020. Learning-to-rank vs. ranking-to-learn: Strategies for regression testing in continuous integration. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020)*. 1261–1272.

[13] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. 12–22.

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

[15] M. Abdelkarim and R. ElAdawi. 2022. TCP-Net: Test case prioritization using end-to-end deep neural networks. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2022)*. 122–129.

[16] Y. Li, X. Dang, W. Pian, A. Habib, J. Klein, and T. F. Bissyandé. 2024. Test input prioritization for graph neural networks. *IEEE Transactions on Software Engineering* 50, 6 (2024), 1396–1424.

[17] N. Reimers and I. Gurevych. 2019. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP 2019)*. 3982–3992.

[18] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*. 9459–9474.

[19] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-É. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou. 2024. The Faiss library. *arXiv*:2401.08281 (2024).

[20] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD 1996)*. 226–231.
[21] Gemma Team et al. 2025. Gemma 3 technical report. *arXiv*:2503.19786 (2025).
[22] S. Robertson and H. Zaragoza. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval* 3, 4 (2009), 333–389.
[23] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. 1990. Indexing by latent semantic analysis. *Journal of the American Society for Information Science* 41, 6 (1990), 391–407.
[24] D. M. Blei, A. Y. Ng, and M. I. Jordan. 2003. Latent Dirichlet allocation. *Journal of Machine Learning Research* 3 (2003), 993–1022.