DARTS: A Lightweight Extractor of Test Artifacts from Natural Text

Giulia Duarte
Universidade Federal do Amazonas
Manaus, Amazonas, Brasil
giulia.duarte@icomp.ufam.edu.br

Igor Lima
Universidade Federal do Amazonas
Manaus, Amazonas, Brasil
igor.lima@icomp.ufam.edu.br

Yan Soares
Universidade Federal do Amazonas
Manaus, Amazonas, Brasil
yan.soares@icomp.ufam.edu.br

Pablo Fonseca
Instituto de Desenvolvimento Tecnológico
Brasília, Distrito Federal, Brasil
pablo.fonseca@indt.org.br

Bruno Gadelha
Universidade Federal do Amazonas
Manaus, Amazonas, Brasil
bruno@icomp.ufam.edu.br

André Carvalho
Universidade Federal do Amazonas
Manaus, Amazonas, Brasil
andre@icomp.ufam.edu.br

Abstract

Exploratory testing is guided by test charters: short, freeform sentences that tell a tester what to investigate. A charter says what to look at, not how; the how is left to the tester's memory of the product. When the catalog, that is the project's living set of documented use cases, holds thousands of entries across dozens of components, even experienced testers miss components and explore unevenly, and LLM-based test generators downstream waste tokens (or hallucinate) when the catalog is not narrowed first. We propose DARTS (Discriminative Artifact Routing for Test Suites): a simple retriever that, given a charter, returns the components most likely to hold the use cases the tester needs. DARTS uses a class-based term-weighting scheme over the catalog, a lightweight statistical model of which words discriminate one component from the others, and scores each charter by lexical similarity to the per-component vocabularies. We evaluated DARTS on 150 charters over a 4,453-use-case catalog. Preliminary results show that DARTS retrieves at least one relevant component in its top-3 for 94.67% of the charters (MRR 0.7311), against 79.33% (MRR 0.6356) for a hybrid retriever and 83.33% (MRR 0.5900) for a siamese ranker on the same workload, and runs end-to-end about 4× faster than the hybrid baseline. The same primitive can aid the generation of test cases with LLMs, help plan coverage, and remind a tester of options they were about to skip. In settings where catalogs evolve and labeled data is scarce, DARTS shows that a careful lexical fingerprint can carry retrieval most of the way, leaving learned methods to be brought in only where they pay off.

Keywords

Test Artifact, Charter, Exploratory Testing, Information Retrieval, Component Extraction, Mobile Testing

1 Introduction

A mobile software project under test grows, over time, a body of test artifacts written in natural language by QA engineers. The most common kinds are *exploratory test charters* [1, 17], manual test cases [3], and regression scripts [12]. Following the session-based testing tradition [1, 17], we define a *charter* as a one-sentence test mission that tells a tester what to investigate during a session. All such artifacts reference, implicitly, parts of the product such as the launcher, the camera, settings, parental controls, or accessibility, which we call the *components* of the product. A charter such as *"Remove the screen time when the time is running"* does not state which components are involved, but a tester familiar with the product knows it lives in the parental-controls and settings components. In a mature industrial project, the *catalog* (the project's living set of documented *use cases*, the small atomic actions a tester or an automation can perform, such as *"open the wallpaper picker"*) easily reaches several thousand entries across dozens of components, and the catalog grows continuously as new features are released or existing ones are refined.

This is where exploratory testing shows a well-known weakness: the charter states only the goal, and the tester has to remember on the fly which use cases are worth performing. With thousands of use cases in the catalog, even experienced testers forget options and coverage becomes uneven. The same problem affects automated approaches: LLM-based generators of exploratory test cases [5, 14] either drown in irrelevant context when given the whole catalog, or hallucinate steps when given nothing. The catalog therefore has to be narrowed to the components that matter for a charter before anything else runs, and cheaply enough to be invoked at every step downstream. We ask whether such narrowing can be done without the apparatus of modern retrieval: *can a method that uses no embeddings, no language model, and no labeled training data return, for a given charter, a top-3 list of components that contains at least one functionally relevant component for the great majority of charters, and do so faster than embedding-based or learned alternatives?*

The natural reflex today is to deploy an embedding-based model to solve this problem, either through hybrid retrieval (BM25 [11] combined with dense Sentence-BERT [10] scores) or through siamese networks [4, 13] trained on related pairs. These methods are accurate where data and compute are abundant, but in our setting they hit three walls. They are slow at query time, since dense lookups and rerankers add hundreds of milliseconds per query and the cost compounds when each charter triggers several lookups inside an LLM pipeline. They need labeled *(charter, component)* pairs that the project may not have. And they drift as the catalog evolves, forcing index rebuilds and, for the siamese network, retraining. Related work confirms this trade-off [6, 8, 9, 13, 15]: when the catalog changes continuously and there is no labeled training set, these solutions may add many costs.

We propose DARTS, which rests on a single observation: although each charter is short and noisy, the aggregated text of all use cases belonging to a component forms an internally coherent vocabulary that distinguishes that component from every other. DARTS turns this observation into a per-component "lexical fingerprint" by weighting terms according to how characteristic they are of a component relative to the rest of the catalog, and scoring a charter against each fingerprint by cosine similarity. The catalog itself plays the role of supervision, so DARTS dispenses with the training stage that the embedding-based baselines require. We evaluate DARTS on 150 real exploratory charters from an industrial Android project over a 4,453-use-case catalog spanning 33 components, comparing against two baselines previously deployed in the same project on the same task and the same dataset: a Hybrid RAG pipeline and a Siamese Network. The contributions are a formal specification of DARTS (Section 3), a controlled comparison on four IR metrics and end-to-end wall time (Section 4), a qualitative error analysis identifying recurring failure modes of the baselines that the lexical fingerprint sidesteps (Section 5), and a discussion of the cases where all three methods fail and of the structural biases of DARTS (Section 6).

2 Background and Related Work

Test artifacts in the form of charters, test cases, scripts, and plans are the documents through which a project's testing knowledge accumulates. In the exploratory-testing tradition of Bach [1], Whittaker [17], and Itkonen and Mäntylä [3], the charter is the central artifact: a short statement of intent that orients a session. In scripted testing, the manual test case and the automation script play analogous roles. In every case the artifact is written in natural language and references product components implicitly through its vocabulary.

Knowing which components are involved in an artifact is useful for several downstream tasks. Coverage-based prioritization [6, 12] can order the execution so that failures are revealed early when artifacts are labeled by component. Suite minimization [8, 9] removes redundant artifacts while preserving coverage, and component overlap is a natural redundancy signal. Identifying similar test cases written in natural language [15] is closely related, since two artifacts that involve the same components are good candidates for clustering or deduplication. LLM-based test-case generation [5] can restrict its candidate space to the components flagged by an upstream retriever, which reduces hallucination. Generating test cases from requirements [2, 16] starts with mapping a requirement to the parts of the product it affects. The mapping from natural-language artifact to components is therefore a widely useful primitive, and the goal of this paper is to build it cheaply enough to keep up to date as the catalog changes.

Recent work on test selection, prioritization, and minimization relies heavily on vector representations of artifacts, from language-model embeddings for similarity and suite size [8, 9] to clustering of redundant cases [15], scalable similarity-based prioritization [6], vector-space near-duplicate detection [13], and the combination of textual similarity with failure history [7]. These works represent the state of the art when there are enough resources to train or host neural models. DARTS targets the complementary case: projects where the catalog changes continuously, computational resources are limited, and labeled training data is unavailable.

3 Method

DARTS rests on a simple principle: the aggregated vocabulary of a component is a class signal strong enough for retrieval. The method takes as input a catalog of product use cases labeled by component and a test artifact written in natural language, and returns a ranked list of the top-N components most likely to contain the use cases needed to satisfy the artifact. Internally, it builds one lexical fingerprint per component once per catalog, and at query time it measures the similarity between the artifact and each fingerprint. The name reflects the two ideas behind the method: *discriminative*, because the per-component weighting brings out the terms that distinguish one component from the others, and *routing*, because the output is used to route a charter to the components a downstream agent should focus on.

As a concrete illustration, given the charter *"Set a bonus time when the timer is running"*, DARTS scores all K components and returns the top-3 `[parental-controls, Settings, Clock]` (Figure 1). The parental-controls component ranks first because the bigrams "screen time" and "bonus time" appear almost exclusively in its class document; *Settings* ranks second because it shares some configuration vocabulary with the charter; *Clock* appears third due to the literal token "timer". The downstream consumer (a human tester or an LLM-based test-case generator) then restricts its candidate use cases to those belonging to the three returned components.

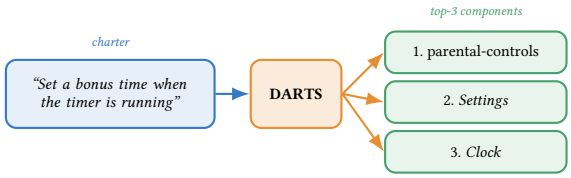

Figure 1: Illustrative DARTS execution for one charter. Component names follow the anonymization described in Section 4.

Formally, let $C = \{c_1, \ldots, c_K\}$ be the set of components in the catalog and $\mathcal{D}$ the set of documented use cases. For each component c_k we build a *class document* d_k by concatenating, over every use case $a \in \mathcal{D}$ that belongs to c_k, the textual content of five fields: `area`, a functional grouping internal to the component; `title`, the short name of the use case; `action`, the imperative phrase performed; `description`, a one-line explanation of the expected effect; and `screen_id`, the dotted identifier of the UI screen, which encodes its position in the application hierarchy. Fields encoding navigation and implementation detail, such as the target screen and the internal function name, are deliberately left out. This aggregation step is what makes DARTS different from a document-level retriever applied to the same catalog: instead of treating every use case as a separate document and asking which use case is most similar to the charter, we collapse all use cases of a component into a single class document and ask which *component* as a whole is most lexically aligned with the charter. The signal we exploit is collective rather than individual.

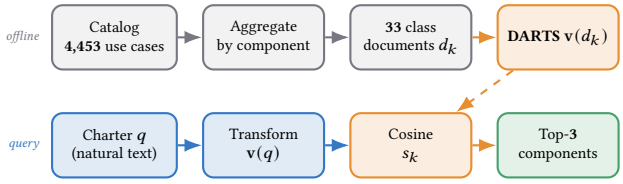

Figure 2: The DARTS pipeline. Offline (top row), the catalog of 4,453 use cases is aggregated by component into 33 class documents, and DARTS fits one lexical fingerprint $\mathbf{v}(d_k)$ per component. At query time (bottom row), a charter is mapped into the same vector space and scored against every fingerprint by cosine similarity, returning the top-3 components.

On the set $\{d_1, \ldots, d_K\}$ we fit a per-component weighting vectorizer with English stop-words removed, n-gram range $(1, 3)$, and a feature ceiling of 5,000. With the $(1, 3)$ range the full vocabulary comprises 62,273 distinct terms, of which the ceiling retains the 5,000 most frequent, covering 57% of all term occurrences and the entire unigram vocabulary of 2,751 terms. The ceiling is a parameter rather than a property of the method: the matrix is sparse and has only K rows, so a larger catalog can raise it proportionally at negligible cost, though it is likely to require a larger ceiling rather than the same one. The vectorizer assigns each term t in class document d_k the weight

$$w(t, k) = \mathrm{tf}(t, d_k) \cdot \left(\log \frac{1 + K}{1 + \mathrm{df}(t)} + 1 \right), \tag{1}$$

where $\mathrm{tf}(t, d_k)$ is the raw frequency of t in d_k and $\mathrm{df}(t)$ is the *class-document frequency*, that is, the number of class documents in which t appears at least once. Because each class document concatenates the entire text of one component, $\mathrm{df}(t)$ effectively counts how many components share the term, so terms frequent inside one component but rare across components receive a high weight, while terms shared by many components are attenuated. The L2-normalized rows of the resulting matrix are the lexical fingerprints $\mathbf{v}(d_k)$ of the K components. Given an artifact q in natural language, we transform q with the same vectorizer (so it is mapped into the same vocabulary and weighting learned from the class documents) and rank components by

$$s_k = \cos\big(\mathbf{v}(q), \mathbf{v}(d_k)\big). \tag{2}$$

The output of DARTS is the list of top-N components ordered by s_k descending (Figure 2). As a safeguard against degenerate queries with no lexical overlap with the catalog, we apply a low threshold τ and emit the sentinel `ALL_COMPONENTS_FALLBACK` whenever the top-1 score s_1 does not reach τ, signaling the downstream consumer to fall back to the whole catalog. We set $\tau = 0.02$, a value low enough that it only fires on the 4 charters in our set whose s_1 is exactly zero, so τ should be read as a documented floor rather than as a discriminative knob.

The output of DARTS can be used in at least four ways inside an industrial pipeline: as a *retrieval* filter that restricts the candidate space shown to a downstream agent (a human tester or an LLM) to the use cases of the top-N components, shrinking the space by roughly an order of magnitude; as a *ranking* signal where the scores s_k themselves order the catalog by relevance; as a primitive for *coverage-based suite selection*, maximizing component diversity or minimizing redundancy between artifacts that hit the same components; and as a *planning* aid, where the distribution of components across the artifacts of a sprint informs allocation of testers and time estimates. We use $N = 3$ and $\tau = 0.02$ in every experiment, with no later tuning. The value $N = 3$ was fixed before the experiments on pragmatic grounds. It is the smallest cut that consistently surfaced a relevant component in preliminary inspection and it matches both the cognitive bandwidth of a human tester scanning a suggestion list and the prompt budget of a downstream LLM, for which each additional component multiplies the number of use cases injected into the context. Section 6 returns to this choice and to the absence of a systematic sweep over N. The vectorizer is fit once over the catalog at a cost of a few seconds for thousands of use cases, and at query time scoring an artifact against all K class documents is a single sparse matrix-vector product followed by a top-N selection, running in milliseconds on commodity CPU.

4 Evaluation

We evaluated DARTS on 150 exploratory charters drawn from an industrial Android project. The project catalog contains 4,453 documented use cases distributed across 33 components. The distribution is strongly skewed: *Settings* alone holds 2,068 use cases or 46% of the catalog, the median component holds 31, and ten components hold ten or fewer. The five largest account for 70%, leaving a long tail of 28 sparsely documented components, which is what makes the size bias of Section 6 concrete and the aggregation step nontrivial. The charters cover screen-time management, connectivity, wallpaper customization, launcher behavior, font and accessibility settings, camera and photos, and other areas. To preserve project confidentiality, the textual examples shown throughout this paper are representative paraphrases of the kind of intent seen in the set rather than literal reproductions of any charter, and the names of OEM-specific components have been replaced by neutral descriptors (for example, “the parental-controls component”, “the screen-saver component”), while standard Android components such as Settings, Clock, Calendar, Status Bar, and News are kept under their usual names.

The 150 charters were written by QA engineers of the project in the normal course of their work, before and independently of this study. They were not authored, edited, or selected for this evaluation and the catalog is likewise the project’s operational artifact. The ground truth by contrast was produced for this study. One of the authors inspected each charter against the list of 33 components and recorded the components a tester would consider correct, up to three per charter. This annotation was completed before the DARTS implementation existed, so no output of any method could influence the labels, and the same labels are used to score all three methods. We nevertheless record it as a threat, since the annotator is not independent of the proposed approach and expectation bias cannot be excluded on the strength of the annotation order alone. Our primary metric is Hit@3 (functional), a manual evaluation in which a reviewer marks YES whenever at least one of the top-3 predicted components is functionally relevant to the charter, even if its name does not match the ground-truth label exactly (for example, accepting *Status Bar* for a SIM-related

Table 1: Ranking and retrieval metrics on 150 charters. Hit@3 (functional), in the last row, is a manual evaluation: YES if at least one top-3 component is functionally relevant to the charter.

Metric	DARTS	Hybrid RAG	Siamese Net
MRR	**0.7311**	0.6356	0.5900
P@1 (%)	**59.33**	52.00	50.67
P@3 (%)	**42.22**	37.78	29.78
R@1 (%)	**22.22**	18.33	18.56
R@3 (%)	**47.33**	40.33	32.56
Hit@3 strict (%)	**88.67**	78.00	70.00
Hit@3 (functional) (%)	**94.67**	79.33	83.33

charter). We chose this metric as primary because it captures what a downstream consumer actually needs: a short list that contains something usable. The criterion was written before any output was inspected, and a single reviewer evaluated the top-3 of all three methods in the same session, with the method identifier hidden during the rating, to limit unconscious bias. Borderline cases (cases where the reviewer initially hesitated) accounted for 9 charters out of 150 and were decided by a written tie-breaker rule that requires the component to host at least one use case whose action verb matches the charter intent.

We complement Hit@3 (functional) with four standard information-retrieval metrics computed against the strict ground truth: Mean Reciprocal Rank (MRR), the reciprocal of the rank at which the first strictly correct component appears, averaged over the 150 charters; Precision@k, the fraction of the top-k predicted components that are in the ground truth; Recall@k, the fraction of ground-truth components that appear in the top-k; and Hit@k (strict), a binary indicator that is 1 if at least one of the top-k predictions matches the ground truth exactly. We compare DARTS to two methods previously deployed on the same project, on the same catalog, against the same 150 charters: a Hybrid RAG pipeline that combines sparse BM25 scoring [11] with dense Sentence-BERT embeddings [10] and issues two RAG calls per atomic action, and a Siamese Network [4, 13] trained on pairs of artifacts and catalog use cases, returning the top-N components by similarity in the learned space. The three methods perform the same task on the same dataset: each is a pre-filter that, given a charter, narrows the catalog down to the top-N components for a downstream exploratory test-case generation pipeline. The baselines were not re-tuned for this study; their configuration is the one previously selected by the team for the same pre-filter role.

Table 1 summarizes the four information-retrieval metrics plus Hit@3 (functional). DARTS leads on every metric. The MRR gap is the cleanest signal: DARTS places the first correct component, on average, at a higher position in the ranking than either baseline, and at $k = 3$ both Precision and Recall remain higher than the baselines'. Hit@3 (functional) is the most practically meaningful number, since it answers the question a downstream consumer cares about: does the top-3 list contain something I can use? At 94.67%, DARTS does, for almost every charter.

Figure 3 plots Precision@k and Recall@k for $k \in \{1, 2, 3\}$. Precision decreases with k because more noise enters the top-k, while

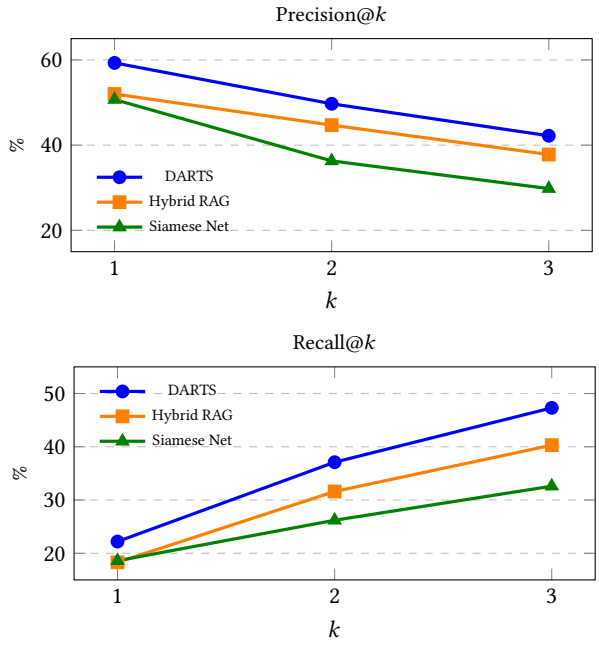

Figure 3: Precision@k (top) and Recall@k (bottom). DARTS keeps the lead in both curves across every cut.

Table 2: End-to-end wall time on 150 charters (hh:mm:ss).

Method	Index	Query	Total
DARTS	00:00:11	**00:07:31**	**00:07:43**
Hybrid RAG	00:00:18	00:31:00	00:31:19
Siamese Net	00:00:16	00:16:32	00:16:49

recall increases because more chances become available to find the correct components. DARTS keeps the lead on both axes at every k. Figure 4 shows Precision against Recall, with each marker corresponding to a top-k cut. The closer to the upper-right corner, the better, and DARTS sits closest to it at every cut. Beyond ranking quality, end-to-end runtime matters when the extractor sits at the front of a longer pipeline. Table 2 reports indexing and query times measured end to end on the 150 charters. "Indexing" includes loading the embedding models, building the indexes and, for the Siamese, training the head. "Query" covers the full charter → component → atomic action → direct action loop, which all three pipelines share and which calls a local LLM for charter decomposition, scenario generation, and per-action judging; differences across methods therefore reflect the retrieval substrate, not the LLM cost, which is constant. DARTS runs roughly 4× faster than Hybrid RAG and roughly 2× faster than the Siamese Network. Across both phases, DARTS is the cheapest.

The empirical distribution of the top-1 score across the 150 charters has a heavy tail (Figure 5). The median is 0.084 and the maximum observed is 0.454, with 24% of charters scoring below 0.05. The shape is effectively bimodal, with a high-confidence regime

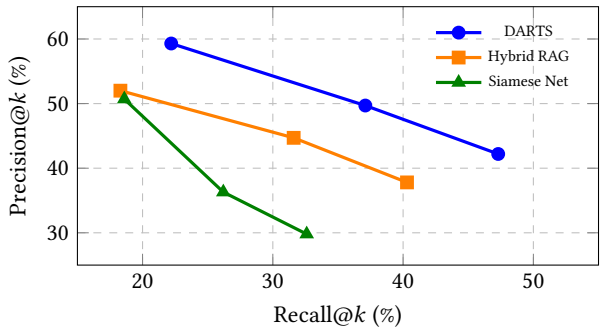

Figure 4: Precision vs. Recall trade-off. Each marker is one value of $k \in \{1, 2, 3\}$. DARTS sits closest to the upper-right.

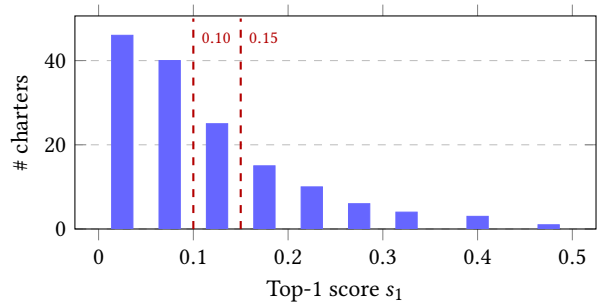

Figure 5: Empirical distribution of s_1 across 150 charters. Dashed lines at 0.10 and 0.15 mark the low- and high-confidence regimes.

above $s_1 \approx 0.15$ in which the leading component is lexically diagnostic enough to be used alone, and a low-confidence regime below $s_1 \approx 0.10$ in which the top-3 list is the meaningful unit and $N = 1$ would routinely miss the relevant component. The threshold $\tau = 0.02$ used as a degenerate-query floor only fires on 4 charters whose s_1 is exactly zero (no lexical overlap with the catalog), confirming that τ is not a discriminative knob but a safeguard.

5 Qualitative Error Analysis

To understand where each method wins and loses, we compared the three methods, charter by charter, under the functional criterion. The pattern is informative. On 106 charters, all three methods return a functionally relevant component and we call these charters easy in the sense that they are decided identically by every method and therefore carry no discriminative information about the comparison; on 11 charters only DARTS hits a correct component, and these reveal the structural advantage of the method; on 3 charters only DARTS fails while the baselines succeed; and 4 charters defeat all three methods.

The 11 cases where only DARTS hits fall into two recurring patterns. The first one shows up in parental-control charters about screen time and bonus time, such as *"Set a bonus time when the timer is running"*, *"Set a bonus time when the timer is exhausted"*, *"Verify if the bonus time is reflected"*, and *"Try set a Bonus time that exceeds hours"*. The dense methods systematically route these to *Clock*, *Calendar*, or other time-related components, because the tokens "timer", "bonus", and "hours" have strong ties to time-keeping components in the embedding space. Here lexical matching beats distributional similarity: dense methods generalize "timer" to clock-related components, whereas DARTS learns from the catalog that "screen time" and "bonus time" co-occur almost exclusively in the parental-controls component, and weights them accordingly.

The second pattern is charters that live under *Settings*, such as *"Change the screen timeout"*, *"Enable screen saver"*, *"Check Data usage"*, and *"Change configurations of a quick-dial prefix"*. The ground truth for these is *Settings*, and the *Settings* component contains 2,068 use cases out of 4,453, nearly half the catalog. Its aggregated text is long and diverse, which makes its dense centroid diffuse and far from any specific charter, so the dense methods end up preferring narrower, more visually compact components such as the parental-controls component or the system-utilities component. DARTS, by contrast, ranks *Settings* high whenever the charter contains tokens that are weighted heavily for that component and rarely appear elsewhere.

Table 3: Charters that all three methods fail.

Charter (paraphrased)	Diagnosis
Add [a contact] with no space	Ambiguous, no explicit object
Check In-Pocket detection	Proprietary jargon, thin coverage
Interact with Screencast (Chromecast)	No *Cast* component in dataset
Create, Edit, Delete APN	Technical, sparsely documented

At the other end of the spectrum, four charters defeat every method (Table 3). They share a structural weakness: the catalog does not contain enough vocabulary to anchor an answer, and proprietary or technical terms such as "In-Pocket", "Chromecast/Screencast", or "APN" have textual coverage in the catalog that is too thin for any retrieval method, lexical or dense. We read these failures as dataset limits, not method limits: improving DARTS on these charters means enriching the catalog, not the algorithm.

The error analysis also clarifies where each method is structurally weak. DARTS is strong on features with distinctive vocabulary ("screen time", "ambient display", "wallpaper"), deterministic, and cheap, but it is vulnerable to synonyms: a charter that says "Talkback" against a catalog that says "accessibility reader" is not matched, because the surface forms differ. Hybrid RAG is better at paraphrase (for instance, "Curated content" is correctly routed to *News*, and "sound options" to the audio-effects component), but the long aggregated representation of large components dilutes the dense vector and reduces discrimination, and the runtime cost during query is not compensated by an equivalent gain in quality. The Siamese Network biases its predictions toward components with thinner aggregated text (for example, the parental-controls and onboarding components); a likely contributing factor is that the negatives selected by the standard "most opposite component" rule are easy in a catalog where most components share generic Android vocabulary, leaving the network with little pressure to learn the harder boundaries between semantically adjacent components.

6 Discussion

DARTS is lexical by construction, and it inherits two structural biases. The first is a component-size bias: components with more catalog entries contribute more terms to the fingerprint and naturally appear more often in top-3 lists. In our experiment, *Settings* concentrates 46% of the catalog but appears in only 38% of the top-3 lists, so the bias is real (Settings shows up disproportionately often relative to other components) but bounded (it does not dominate beyond its share). The second is a diagnostic-vocabulary bias: charters with rare, distinctive tokens such as "wallpaper" are routed confidently, while charters with generic vocabulary fall into the low-confidence regime of the score distribution, which is the regime where lexical methods are limited in principle. Stated positively, these biases delimit the conditions under which DARTS is expected to work. The method assumes that components possess distinctive vocabulary, meaning that the terms characteristic of one component are rare in the others, and that the catalog documents each component densely enough for that vocabulary to surface. Our catalog satisfies the first condition well, since Android feature names are largely non-overlapping, and it satisfies the second unevenly, which is exactly what the four universal failures of Table 3 expose. Those are the charters whose target components are thin in the catalog. A catalog whose components share generic vocabulary, or one documented at a coarser granularity, would degrade the fingerprint regardless of the algorithm, and in such a setting a distributional method would be expected to recover part of the gap. A natural objection is that a supervised classifier should beat any unsupervised lexical method. There are two practical reasons why this is not currently within reach for the project: there is no large labeled set of (charter, component) pairs available (the 150 charters reported here were the bottleneck, not the catalog), and a learned classifier ages with the catalog and has to be retrained whenever components are added or renamed, whereas DARTS refits from the catalog itself in seconds. A learned reranker over the top-3 of DARTS is a natural hybrid to investigate.

6.1 Threats to Validity

Internal validity rests chiefly on how the ground truth was produced. It was annotated by one of the authors, who is not independent of the proposed method, although the annotation preceded the implementation and the same labels score all three methods, so no method is advantaged by it. Expectation bias is not excluded by these measures. The Hit@3 functional rating was performed by a single reviewer with the method identifier hidden and borderline cases resolved by a written rule, but blinding mitigates rather than measures annotator bias, and Cohen's κ was not computed. A second independent annotator is the first item of the extended study. The baselines were not re-tuned, so we cannot rule out that a dedicated tuning round would narrow part of the effectiveness gap, though the runtime advantage is structural and no hyperparameter choice removes the cost of embedding lookup and reranking.

External validity is limited to one Android catalog and 150 charters. Whether the fingerprint transfers to non-Android catalogs is open, and so is the more immediate question of whether it transfers to other Android catalogs, where conventions are shared but the product, the component decomposition, and the documentation style differ.

On construct validity, Hit@3 functional encodes a judgment that admits alternative definitions, and a stricter or looser construct would move all three methods, though not necessarily by the same amount. The strict IR metrics avoid that judgment but count a functionally reasonable component as wrong when it is absent from the ground truth, which is why we report both. The value $N = 3$ was fixed a priori, so the numbers characterize one operating point rather than a curve.

7 Conclusion

We presented DARTS, a lightweight retriever that maps natural-language test artifacts to product components. The method scores a charter against per-component lexical fingerprints derived from the catalog, dispensing with the LLM, the embedding model, and the labeled data that the embedding-based baselines require, and it ranks the top-N most relevant components for a charter in milliseconds per query. Preliminary results on 150 industrial exploratory charters point to a Hit@3 (functional) rate of 94.67% (MRR 0.7311), with an advantage over Hybrid RAG and Siamese baselines on the same project and at a fraction of the compute cost. Whether the lexical fingerprint generalizes to non-Android catalogs and to non-exploratory artifacts is the question we plan to answer next.

Artifact Availability

The catalog, charters, ground truth, and method implementations used in this study are derived from internal artifacts of an industrial mobile-testing project under non-disclosure agreement. They contain proprietary feature names, internal screen identifiers, and platform-specific commands, and the DARTS implementation embeds field names that would let a reader reconstruct the catalog schema. For these reasons, we are unable to release artifacts at this time. We plan to revisit this decision once the underlying catalog can be anonymized at the schema level.

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) through projects CNPq - 313137/2025-0; and CNPq 406506/2025-6, and Amazonas State Research Support Foundation- FAPEAM- through the POS-GRAD 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.

Use of AI Tools

The authors used Anthropic's Claude as an editing assistant during the preparation of this manuscript: to suggest rewordings of prose passages already drafted by the authors and to help fit text within the page budget. Claude was not used to generate technical content, equations, experimental results, citations, or analysis. The authors reviewed every suggestion before incorporating it, and any factual claim, reference, or numerical result in the paper was verified by the authors against the underlying data and original sources.

References

[1] J. Bach. Session-Based Test Management. *Software Testing and Quality Engineering (STQE)*, 2(6), 2000.

[2] J. Frattini, J. Fischbach, and A. Bauer. CiRA: An Open-Source Python Package for Automated Generation of Test Case Descriptions from Natural Language Requirements. In *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*, pages 68–71, 2023.

[3] J. Itkonen and M. V. Mäntylä. Are test cases needed? Replicated comparison between exploratory and test-case-based software testing. *Empirical Software Engineering*, 19(2):303–342, 2014.

[4] G. Koch, R. Zemel, and R. Salakhutdinov. Siamese Neural Networks for One-Shot Image Recognition. In *ICML Deep Learning Workshop*, 2015.

[5] I. Lima, A. Carvalho, Y. Soares, G. Pacheco, A. Peralta, H. Melo, N. Ferreira, and B. Costa. Automated Generation of Exploratory Test Cases Using Prompt Chaining and Reflective Evaluation. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software (SBES)*, pages 825–831. SBC, 2025. doi:10.5753/sbes.2025.11599.

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

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

[8] R. Pan, T. A. Ghaleb, and L. C. Briand. LTM: Scalable and Black-Box Similarity-Based Test Suite Minimization Based on Language Models. *IEEE Transactions on Software Engineering*, 50(11):3053–3070, 2024. doi:10.1109/TSE.2024.3469582.

[9] R. Pan, F. Niu, L. C. Briand, and H. Hu. Requirements Coverage-Guided Minimization for Natural Language Test Cases. *ACM Transactions on Software Engineering and Methodology*, 1(1), January 2026, 25 pages. doi:10.1145/3798164.

[10] N. Reimers and I. Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, pages 3982–3992. ACL, 2019.

[11] S. Robertson and H. Zaragoza. The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389, 2009.

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

[13] R. Yandrapally, A. Stocco, and A. Mesbah. Near-Duplicate Detection in Web App Model Inference. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 186–197. ACM, 2020.

[14] Y. Su, D. Liao, Z. Xing, Q. Huang, M. Xie, Q. Lu, and X. Xu. Enhancing Exploratory Testing by Large Language Model and Knowledge Graph. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, Lisbon, Portugal. ACM, 2024.

[15] M. Viggiato, D. Paas, C. Buzon, and C.-P. Bezemer. Identifying similar test cases that are specified in natural language. *IEEE Transactions on Software Engineering*, 49(3):1027–1043, 2022.

[16] C. Wang, F. Pastore, A. Goknil, and L. C. Briand. Automatic generation of acceptance test cases from use case specifications: an NLP-based approach. *IEEE Transactions on Software Engineering*, 48(2):585–616, 2022.

[17] J. A. Whittaker. *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. Addison-Wesley Professional, 2009.