Faster Feedback with AI? A Test Prioritization Study

Toni Mattis
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
toni.mattis@hpi.uni-potsdam.de

Lukas Böhme
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
lukas.boehme@hpi.uni-potsdam.de

Eva Krebs
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
eva.krebs@hpi.uni-potsdam.de

Martin C. Rinard
Massachusetts Institute of Technology
Cambridge, USA
rinard@csail.mit.edu

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
robert.hirschfeld@uni-potsdam.de

ABSTRACT

Feedback during programming is desirable, but its usefulness depends on immediacy and relevance to the task. Unit and regression testing are practices to ensure programmers can obtain feedback on their changes; however, running a large test suite is rarely fast, and only a few results are relevant.

Identifying tests relevant to a change can help programmers in two ways: upcoming issues can be detected earlier during programming, and relevant tests can serve as examples to help programmers understand the code they are editing.

In this work, we describe an approach to evaluate how well large language models (LLMs) and embedding models can judge the relevance of a test to a change. We construct a dataset by applying faulty variations of real-world code changes and measuring whether the model could nominate the failing tests beforehand.

We found that, while embedding models perform best on such a task, even simple information retrieval models are surprisingly competitive. In contrast, pre-trained LLMs are of limited use as they focus on confounding aspects like coding styles.

We argue that the high computational cost of AI models is not always justified, and tool developers should also consider non-AI models for code-related retrieval and recommendation tasks. Lastly, we generalize from unit tests to live examples and outline how our approach can benefit live programming environments.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; Software version control; • **Computing methodologies → Natural language processing**.

KEYWORDS

generative ai, large language models, embedding models, testing, test prioritization

ACM Reference Format:
Toni Mattis, Lukas Böhme, Eva Krebs, Martin C. Rinard, and Robert Hirschfeld. 2024. Faster Feedback with AI? A Test Prioritization Study. In *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming (‹Programming› Companion '24), March 11–15, 2024, Lund, Sweden.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3660829.3660837

1 INTRODUCTION

Immediate feedback during programming activities helps programmers catch errors earlier and allows them to quickly validate and iterate on the behavior they have in mind while editing.

Automated testing is a practice that allows programmers to obtain such feedback by exercising the system under test with exemplary data. However, running a large test suite introduces delays in feedback, and critical tests that identify an issue may be hidden in the test suite.

In this study, we address the problem of ranking tests by their relevance to a program change, also known as Regression Test Prioritization (RTP). Apart from choosing the best tests to run for detecting defects earlier, presenting the most relevant test to a programmer can provide valuable context and examples of how the code under change is being used.

With the recent emergence of large language models (LLMs), increased focus has been on their capability to generate code. However, we will assess their capability to judge already existing code in the form of test cases.

While LLMs offer significant potential, they also have high computational costs. Using them as a service can make them more accessible but incurs a privacy risk – especially when working on closed-source programs, and local deployment typically requires specialized hardware, such as high-video-memory GPUs. Hence, we will measure and compare the performance of more conservative approaches in the RTP setting.

Our study compares a state-of-the-art code LLM, a code embedding model, and a widely used information retrieval algorithm on an RTP task on three open-source Python projects. We find that using the transformer-based embedding model results in the most effective prioritization, while the information retrieval model achieves competitive performance for its low complexity. We also find that the LLM performs poorly and subsequently investigate possible causes.

In this study, we focus on unit tests, yet our framework and findings can be generalized to other instances that provide example data or validation mechanics for a program, including documentation, AI-generated code, or Babylonian examples in live programming environments. In a future where an increasing amount of code is AI-generated, tools that make growing code bases more accessible are particularly valuable.

2 BACKGROUND

This section will review large language models (LLMs) and embedding models on a high level. We will then discuss current regression test prioritization (RTP) approaches and explain the Okapi BM25 information retrieval algorithm used as our non-AI baseline.

2.1 Language Models

A language model is a stateless function that, given a sequence $[t_1, \ldots, t_n]$ of tokens, computes probabilities $P(t_{n+1}|[t_1, \ldots, t_n])$ for the next token following that sequence. Tokens are independent of the language's grammar and represent the most common substrings from the training data.

Generative LLMs. This stateless function alone does not lend itself to generating code. It starts with an initial sequence (prompt) and repeatedly selects and appends one of the most probable subsequent tokens for the next iteration. Based on this, a theoretically infinite sequence (completion) can be generated until the process is stopped by special tokens (e.g., a line break or a special end-of-sequence "EOS" token) or a defined maximum size. In this work, we do not generate new code but use the underlying probabilities to calculate how likely it is that a given test would be generated based on a change.

State-of-the-art code-generating models such as *CodeLlama* [13] and *StableCode* [11] are based on the transformer architecture.

Embeddings. An embedding aims to produce vectors for (textual) data so that the proximity of two vectors measures the semantic similarity of their associated data.

While transformer-based LLMs focus on the next token, they capture semantic information in the vectors associated with each token by incorporating other tokens as context. The same architecture can be trained to predict the *current* tokens (e.g., giving a masked input to be reconstructed) so that these vectors become a robust "lossy compression" of the input string. By pooling the vectors for each token, a single vector can be produced that we refer to as embedding of the input. Examples of such models specialized in source code are *CodeBert* [3] and *UniXCoder* [4]. They typically have a smaller footprint than generative LLMs.

2.2 Regression Test Prioritization

Research on test relevance has been concerned with two major problems: Regression Test Prioritization (RTP), in which the objective is to rank all tests, and Regression Test Selection (RTS), where the best subset of tests is to be found within certain constraints (e.g., given number of tests or maximum execution time).

For the remainder of this paper, we will focus on RTP. RTP approaches can be subdivided depending on the input data used to rank tests. While many approaches take the whole program as input and aim to uncover defects anywhere as quickly as possible, we are only interested in a (small) *change* and a ranking that helps programmers assess their change as quickly as possible.

Evaluating RTP. An RTP approach should ideally be evaluated using realistic change and testing histories. Representative data is typically scarce as "broken" code changes are rarely published, and recent approaches to collect such a dataset are limited (e.g., RTPTorrent [8] only includes Java projects).

Mutation testing, i.e., systematically injecting defects into the program and running the test suite, is a way to obtain synthetic test logs quickly but has several disadvantages: The synthetic changes are not representative of real-world programming activity, and the test suite has evolved in response to natural defects and feedback requirements and might not provide sufficient granularity or coverage to distinguish or detect synthetic faults. In subsection 3.1, we describe an approach that combines the scalability of mutation testing while retaining realistic changes.

Currently effective strategies. Current approaches to RTP often use historical data or natural language features. Applying the *demonstrated fault effectiveness* [5] strategy that ranks tests by their most recent failure rates to a real-world dataset [8] revealed that naturally occurring test failures follow predictable patterns, i.e., the most recently failed test is the most likely to fail again.

In the absence of historical data, the *REPiR* system [14] demonstrated that a widely used information retrieval technique (*Okapi BM25*, described below) is highly effective at predicting test failures even without dynamic or static program information apart from the changed source code alone. This is more effective than using statically determined test coverage [9].

Mutation testing has been used to train models for RTP tasks. For example, *precision-based ranking* [7] uses shared vocabulary between mutants and test failures to determine which identifiers in the code are predictive of test failures when they change. Meier et al. [10] developed a real-time heuristic that pre-trains decision trees on mutation testing data to perform *continuous live prioritization* while navigating and changing code.

2.3 Information Retrieval in RTP

Information-retrieval-based RTP approaches like *REPiR* use the *Okapi BM25* model. BM25 computes the similarity of a query (a change) to a set of documents (the tests) by splitting both into individual terms (normalized words) and comparing term frequencies ("TF"). The relevance of each term is weighted inversely proportional to the percentage of documents it appears in (inverted document frequency, "IDF") to boost the weight of specific over common terms. As such, it is an instance of a TF-IDF model.

The similarity of a test T to the change C is computed by representing both as vocabulary-sized vectors for n different terms in the vocabulary $(\mathrm{tf}(t_1)\,\mathrm{idf}(t_1), \ldots, \mathrm{tf}(t_n)\,\mathrm{idf}(t_n))$ and taking their dot product. We only sum over terms t appearing in the change since the other vector components are 0:

$$Prio(C,T) = \sum_{t \in C} \mathrm{tf}_C(t)\,\mathrm{tf}_T(t)(\mathrm{idf}(t))^2 \tag{1}$$

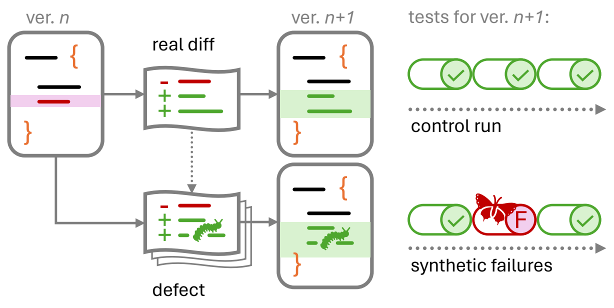

Figure 1: Change-based mutation testing: A defect is introduced into a real change between two versions. If the defect manifests during test runs, the additional test failures serve as ground truth for evaluating RTP approaches with the original change as input.

According to Okapi BM25 and its use in REPiR [14], the following definitions for tf and idf are used:

$$\mathrm{tf}_T(t) = \frac{k_1 n_t}{n_t + k_1(1 - b + b(N_T/\hat{N}))} \quad (2)$$

$$\mathrm{tf}_C(t) = \frac{k_2 n_t}{n_t + k_2} \quad (3)$$

$$\mathrm{idf}(t) = log\frac{D+1}{d_t+1} \quad (4)$$

Here, n_t is the number of times term t appears in the respective test or change, N_T is the size (total number of terms) of the test, $\hat{N}$ the average size of a test, D the number of tests ("documents") and d_t the number of tests in which t appears at least once. We have free parameters k_1 and k_2 that influence how slowly a term "saturates" in tests and changes, and b determines how much length penalizes term frequencies. REPiR uses the values $k_1 = 1, k_2 = 1000$, and $b = 0.3$, but we found setting $k_1 = k_2 = 10$ yields better results since related work might have used much larger changes, and thus a larger k_2.

3 APPROACH

Our study aims to determine how well LLMs and embeddings prioritize tests of a test suite given a change.

Before comparing different prioritization strategies, we need ground truth data that establishes which tests are relevant to a given change by observing them *fail* when that change introduces a defect. This section describes our approach to *change-based mutation testing* and two different methods to rank tests using AI models.

3.1 Change-based Mutation Testing

Change-based mutation testing is a technique that generates synthetic test logs by breaking *real* changes. It requires a fine-grained version history of the code and, for each version except the first, applies defective patches from a previous version before running its current tests as illustrated in Figure 1.

While the defects introduced into the program do not represent human errors, their distribution follows actual programming activity (e.g., no defects in never-edited code). Since we later use the whole real-world change (not just the mutation) as input to the RTP algorithm, we can emulate a realistic scenario where the fault location is not precisely known, only that it had to be introduced by the most recent change. The failing tests strongly suggest they would have been "relevant" to the change.

In the following, we will explain our implementation of change-based mutation testing in Python. It is an updated variation based on a previous implementation [7].

Change selection. For a program under test, we considered all Git commits in reverse chronological order. We compute the diff to their parent and skip changes that do not modify code (`.py` files) or change more than 50 lines of code at once. Large changes rarely reflect scenarios in which fast feedback is helpful or attainable.

Since changes can go back several years, we parse project metadata (`requiremens.txt`, `tox.ini` or `pyproject.toml`) and install the required version of the dependencies at the time of the commit through the package manager (`pip`).

Control run. We run all tests once before "breaking" the change and collect the results. If the control run fails entirely, we omit this change – this is mostly the case when the commit includes wrong dependencies or is old enough to use deprecated language features. We terminate change collection for a project once any commit (sorted descendingly by timestamp) starts to use a deprecated language feature.

Mutation runs. We use four mutation operators on the changes. They are inspired by *Stryker*[1] and have been empirically selected to apply relatively uniformly across Python code:

Binary This operator identifies binary operators in source code and replaces them with their counterparts (e.g., "+" with "-" or "|" with "&"). We exclude the modulo operator (%) for its extensive use in string formatting.

Strings This operator identifies strings introduced or changed and replaces them with the empty string.

Numbers This operator identifies numbers introduced or changed and replaces them with 42.

Conditions This operator identifies comparisons in a conditional role (e.g., as `if` condition) and yields two candidates per location, replacing them with `True` and `False` successively.

Each mutation operator identifies a set of mutation sites within the diff to the previous commit. We skip arithmetic mutations inside (algebraic) type annotations as they cause import-time crashes before any tests can be run and conditions inside while-loops as they likely cause infinite loops, delaying the testing process.

For each mutation site, we check out the commit into a separate working directory, apply one mutation, and run all tests except those that did not already succeed in the control run. Not all test runs finish since some mutations crash the test runner.

[1] https://stryker-mutator.io/ (last accessed 2024-02-29)

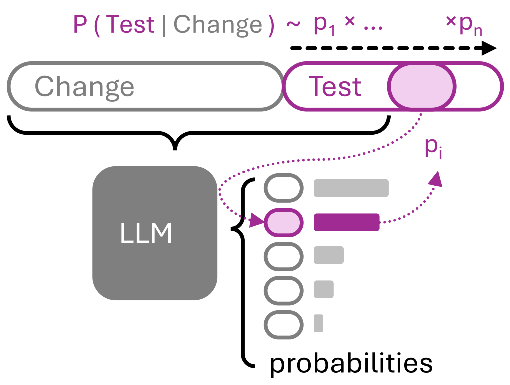

Figure 2: Computing the relevance of a test to a change by accumulating the probabilities output by an LLM as if it was generating the test.

At the end of the process, we have one control run and several mutation test runs per commit, associating at least one "defective" test run with each change.

3.2 LLM-based Test Prioritization

Our LLM-based approach uses the capability of a code-generating LLM to generate tests. However, instead of sampling a new test from the model, we use the probability that the LLM would have generated a given test as the test's priority. The process is illustrated in Figure 2.

More precisely, we use the change as a prompt, and rather than sampling the next token for a test, we look at the probability of the actual token in the test and then proceed to the next token as if the previous one had been generated. The transformer architecture of modern LLMs can compute the probabilities for all tokens in parallel. Since the test probability (product of token probabilities) would make longer tests less probable, we compute their geometric mean to account for varying lengths.

Formally, we compute the average probability $\hat{P}$ for a test T (consisting of tokens $t_1 \ldots t_n$) given a change C as:

$$\hat{P}(T|C) = \sqrt[n]{LLM(t_1|C) \ldots LLM(t_n|C, t_1, \ldots, t_{n-1})} \tag{5}$$

Change formatting. A relevant variation point is how we format the change as an LLM prompt. This process is also known as prompt engineering. Including only the changed lines omits relevant context and might be syntactically incorrect (e.g., if the prompt leaves an open bracket after a function call, the LLM might compute the probability that a test appears as a function parameter, which is generally very low and not a useful metric).

We parse all files involved in the change to build a syntactically correct context. For each contiguous chunk of a (possibly scattered) change, we compute the smallest set of complete statements covering the chunk. Then, we follow the lexical scope upwards and prepend all scope definitions (such as functions, classes, and eventually the file name as a comment). The code replaced in this change is commented out.

Listing 1 illustrates the context built around a single-line change replacing a function parameter. First, the full statement (function call) is identified, followed by the surrounding condition, method definition, and class definition. A prompt is appended instructing the LLM to continue with a test.

Listing 1: LLM prompt with lexical context around a single-line change replacing the second parameter in a call formatted over multiple lines. The red passage marks the removed line, the green line the inserted code.

```
# file: package/module/submodule.py
class AClass:
  def method(self, args):
    if condition:
      call(
        param1=value1,
        # changed:
        # param2=value2
        param2=new_value2
      )
# A test validating this change:
```

Optimizations. In our implementation, we use the arithmetic mean of logarithms of probabilities to avoid floating point underflows while multiplying many small numbers. Moreover, since we compute the probabilities over many tests while keeping the prompt constant, we process the prompt once, snapshot the transformer state, and re-use it for all tests.

Model selection. For this study, we use the *StableCode-3B* [11] model, which is the newest open-source LLM for code at the time of writing. Moreover, with less than 3 billion parameters, it is relatively small and performs similarly to previous models in the 7 - 15 billion parameter range.

3.3 Embedding-based Test Prioritization

Our embedding-based strategy uses two different variations to prioritize tests. Both strategies first compute a vector representation of the source code of each test but differ in how they process composite changes that consist of several disjoint chunks:

Whole-change embedding computes the average embedding of all chunks of the change. The test priority is the cosine similarity to this vector: $Prio(T|C) = e(T) \cdot e(C)$ for the embedding function e, change C and test T (see Figure 3).

Chunked embedding computes a separate vector for each contiguous part of the change. The test priority is the cosine similarity to the *closest* chunk vector: $Prio(T|C) = max_i(e(T) \cdot e(c_i))$ for chunks c_i in change C (see Figure 4).

We expect that if a change covers multiple locations, its average vector is closest to the tests relevant to most locations simultaneously. In contrast, the rationale behind chunked embedding is that different specific tests are relevant to different locations. Evaluation

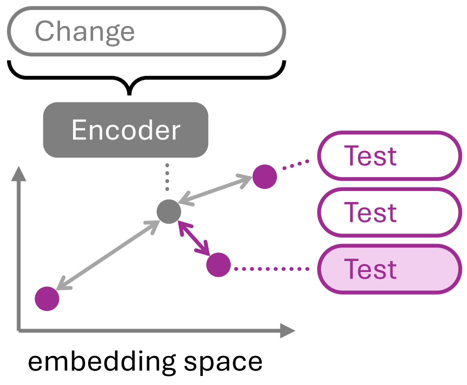

Figure 3: Test prioritization by embedding the whole change in the same vector space as individual tests. Tests are ranked by proximity (cosine similarity) in the vector space.

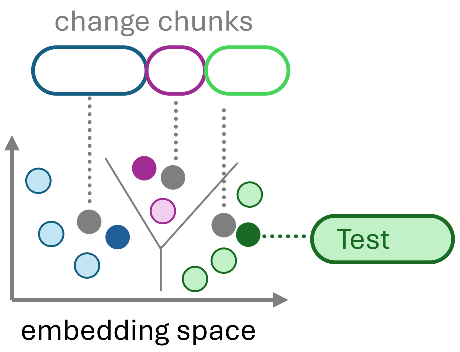

Figure 4: Test prioritization by chunked embedding. Tests are ranked by proximity to their closest chunk, clustering around the parts of a cross-cutting change rather than a single mean vector.

will show whether retrieving tests relevant to the change as a whole or specifically to one of its parts is more effective.

We format individual chunks as in the LLM prompt, i.e., include the lexical scope. We omit the final prompt line asking to generate a test since embedding models are not trained with regard to what comes "after" the input.

For this study, we use the *UniXCoder* model [4].

4 EVALUATION

To compare test prioritization models, we perform *change-based mutation testing* (see subsection 3.1) on open-source Python projects to generate evaluation data. We will then compare the performance of an LLM, the embedding strategies, and a baseline taken from the information retrieval field. The latter is interesting since we expect it to set a high bar, which should prompt a discussion of whether much more computation-intensive procedures are required.

4.1 Data

For this study, we selected three Python projects from the top non-educational Python projects ranked by GitHub at the time of writing that did not have compiled dependencies, ruling out course material and machine learning libraries. All projects are well-tested and have a fine-grained version history:

Flask a popular web framework
Requests an HTTP client library
Jinja a templating engine

We summarized the number of commits, extracted test runs, and their statistical properties in Table 1. The table shows the cleaned dataset: We discarded commits over 50 LOC, commits with crashes or incompatibilities, and mutants without test failures. The average number of tests executed per run is higher than that of unique tests since parametrized tests can be executed multiple times with different inputs and pass or fail independently. The reported test size excludes empty lines but includes comments, and the reported runtime does not include shared setup code or fixtures, only the isolated test time.

4.2 Metrics and Baselines

Test prioritization effectiveness is typically evaluated using *average percentage of faults detected* (APFD). The APFD metric computes the area under the curve that plots the percentage of uncovered faults so far (y-axis) over the percentage of already executed tests (x-axis):

$$APFD = 1 - \frac{\sum_i^{n_f} TF_i}{n_f \times n_t} + \frac{1}{2 \times n_t} \tag{6}$$

where n_t and n_f denote the number of tests and faults and TF_i the position of the earliest test that uncovered the i-th fault.

In our experiments, we are not only interested in the performance over all defects (we will call this the *macro-APFD*) but also whether individual test runs are ranked in a way that failures appear first. Thus, we will define the *micro-APFD* over a single test run by equating a fault with a test failure in the above formula, i.e., we compute the area under the curve that plots the percentage of *failures* over the percentage of tests executed. Figure 5 illustrates this procedure.

Baselines. We compare prioritization performance in terms of APFD to the original (default) ordering chosen by the test runner, a random order, and the *Okapi BM25* information retrieval method described in subsection 2.3.

4.3 Results

Our results in Table 2 show that *chunked embedding* outperforms other strategies in two of the three projects but is outperformed by the *BM25 baseline* in the Flask project. We hypothesize that the larger tests (13 vs. 8 − 9 LOC) in Flask contribute to better performance of document-oriented information retrieval while "confusing" the embedding model. When comparing chunked with whole-change embeddings, the chunked embedding is consistently the

Project	Commits	Unique Tests	Ø Tests per run	Mutants	Ø LOC changed	Ø Test LOC	Median runtime (ms)	Ø Failure rate (%)
Flask	159	390	442	726	12.5	13.4	1.6	1.1
Requests	43	314	557	188	13.8	8.0	2.0	4.4
Jinja	68	655	828	420	15.2	8.6	14.0	14.8

Table 1: Statistics per project. "Ø" denotes averages. The number of mutants corresponds to the number of test runs used for our study. The number of actual tests per run is higher than the number of unique tests because parametrized tests are run multiple times with different inputs and can pass/fail independently.

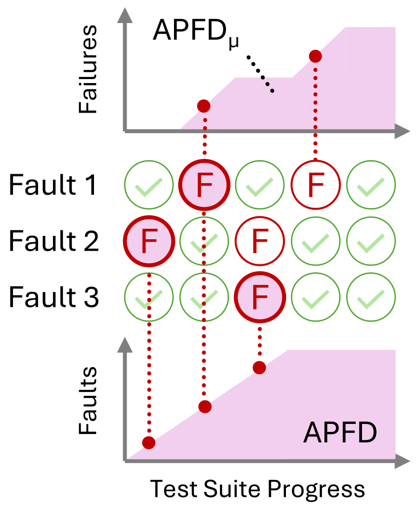

Figure 5: Visualization of micro and macro-APFD. Micro-APFD (top) measures the area under the curve obtained by plotting the percentage of failures over the percentage of executed tests. Macro-APFD (bottom) plots the percentage of overall faults detected and only considers the earliest failing test uncovering the fault.

better choice, suggesting that embedding smaller code snippets better captures semantic relationships.

Unexpectedly, the LLM does not perform well compared to embeddings, although the model was trained on vastly more data and should be able to deal better with context. Upon manual inspection of individual token probabilities in a sample of worst-ranked tests, we notice that the LLM might be "surprised" by complex test logic (such as monkey-patching) and extensive comments while assigning high probabilities to tokens that repeat previous patterns or simple statements. Long tests cause the context (change) to have lower weight as it resorts to judging the test's internal consistency by assigning high probabilities to repetitive code passages encountered earlier in the test.

The curve showing how fast faults (mutants) are detected throughout the running test suite in Figure 6 (appendix) further illustrates the fault-detection characteristics of the individual strategies. It is worth mentioning that all strategies cross the 50% threshold within the first 16 or fewer tests.

The violin plots in Figure 6 show the variation across *micro-APFD*. The high variability indicates that, although the strategies find a fault relatively fast with one test, several tests fail much later.

Strategy	Project APFD Flask	Requests	Jinja
LLM	0.890	0.973	0.874
embedding (chunked)	0.923	**0.980**	**0.934**
embeding (whole change)	0.922	0.960	0.896
Okapi BM25	**0.931**	0.944	0.863
random	0.709	0.861	0.817
default (unchanged)	0.629	0.781	0.750

Table 2: Macro-APFD results per strategy and project. Larger numbers are better.

5 DISCUSSION AND FUTURE WORK

5.1 Limitations

Although our study is limited to synthetic defects, small changes, and only three Python projects with a relatively clean code base, it illustrates some important learnings:

(1) Simple information retrieval models like BM25 can still be highly competitive for code retrieval and search tasks. Especially without specialized hardware like GPUs, they might be a better choice than models from the AI field.
(2) Embeddings benefit from small contexts and are prone to averaging out nuances. This has consequences for the design of RTP and other retrieval or recommendation systems. Breaking up the search corpus and the query and matching embeddings of their parts should be considered an alternative to retrieving or ranking context-heavy data or queries.
(3) Pre-trained LLMs, when not fine-tuned, are "opinionated" in a way that distracts them from the original context. Instead of rewarding a relevant test, the LLM considers best practices, internal consistency, and simplicity.

Moreover, our LLM-based probability computation takes 15 milliseconds per test on a high-end consumer GPU. Since most unit tests run faster (see Table 1), this approach does not scale favorably and will likely not yield faster feedback on test suites like those we studied. However, smaller models make embeddings relatively fast to compute on a GPU. Vectors for tests can be cached until the test is modified, resulting in a significant speed-up.

5.2 Beyond Tests: Live Examples

RTP is a benchmark representative for several scenarios: Techniques that work well for predicting test failures can be used as recommender systems and select tests that best illustrate how the code under change is being used. Tests and examples do not need to originate from the same program: The recommender can pull in relevant code from other code bases, documentation, or discussions, alerting programmers that their change might affect how others use their software or inform them about new usage examples.

In modern live programming environments, locating the proper tests can quickly set up *live context* by running them to provide example data. In example-based live programming, live examples can take the role of tests and bring examples much closer to the code - examples of such systems include Newspeak's Exemplars [1], Example-Centric Programming [2], and Babylonian programming [12]. If many live examples exist, an RTP-style recommender can select those relevant to the current context.

Prioritizing run-time objects beyond those created by test code might bring additional opportunities for program comprehension and debugging but also new challenges: Current LLMs work on source code and require fine-tuning on (serialized) live objects to become functional.

5.3 Beyond Pre-trained LLMs: Fine-tuning

Our study is limited to pre-trained models. Although they have "seen" a lot of source code, they are not adjusted to the specific task.

A next step is to fine-tune the used models. Fine-tuning can teach multiple aspects to an LLM or embedding model by providing example data to show the syntax we use to denote changes and which tests are failing and passing in our generated (or in real-world) data sets. Also, the model can "familiarize itself" with a project's coding style and best practices.

Embedding models can further be trained with a contrastive objective by moving vectors away from negative examples and towards positive examples. Our generated data contains both failing and passing tests and is thus an ideal candidate to fine-tune an RTP-specific embedding.

5.4 Beyond Changes: Context Engineering and Generation

We only tested with a simple variant of constructing the context used by the LLMs to judge a test. Using too much context introduces noise and can confuse models while using too little context limits opportunities to link tests semantically. This problem has no obvious answer and demands further experimentation.

Candidate objects to include in the context are:

- Surrounding lines of code
- Nearby code comments
- Dependencies and program slices (e.g., definitions of the variables used in the change or statements affecting the changed code)

However, with generative models, we are not limited to existing context: The models could generate context themselves. This leads to other approaches: (1.) Let the LLM propose candidate tests and use their embeddings to choose among the most similar real tests. (2.) Use an LLM to summarize a much larger context and use the embedded summary to query the embedding space. Such procedures are known as *generation-augmented retrieval* (GAR) [6]. This could combine the effectiveness of embeddings we demonstrated in our study with the capabilities of an LLM.

CONCLUSION

In this work, we studied how *large language models (LLMs)* and *embedding models* specialized in source code can help programmers obtain faster feedback using *regression test prioritization (RTP)* as an example benchmark. We constructed a Python-based dataset by combining real changes and test suites with synthetic defects to overcome the lack of public data on failing test runs. Subsequently, we evaluated how fast each approach would detect these defects when given the chance to re-rank all tests before they run.

Based on how well state-of-the-art LLMs can generate tests, we expected them to recognize relevant tests when presented with an existing test suite. Surprisingly, we found that LLMs focus on irrelevant aspects of the test, such as its coding style, thereby confounding their judgment. We hypothesize that fine-tuning can solve this problem but need to point out that minor improvements might not justify the high computational cost of an LLM.

Embedding models, in contrast, are effective and efficient at predicting test failures, and we could demonstrate trade-offs that tool developers using them need to consider. We also showed that simple information-retrieval techniques are competitive and can even outperform embedding models in some cases. Unless code generation is needed, it appears worthwhile to carefully consider and benchmark a simple model before integrating AI into a software engineering tool.

Our findings have future applications in the program comprehension and live programming domains where they can help programmers find and run other *live examples* besides tests.

ACKNOWLEDGMENTS

This work is supported by the HPI-MIT "Designing for Sustainability" research program[2]. We also gratefully acknowledge the financial support of HPI's Research School.

REFERENCES

[1] Gilad Bracha. 2021. Enhancing Liveness with Exemplars in the Newspeak IDE. https://newspeaklanguage.org/pubs/newspeak-exemplars.pdf.
[2] Jonathan Edwards. 2004. Example Centric Programming. *ACM SIGPLAN Notices* 39, 12 (Dec. 2004), 84–91. https://doi.org/10.1145/1052883.1052894
[3] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[4] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).
[5] Jung-Min Kim and Adam Porter. 2002. A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments. In *Proceedings of the 24th International Conference on Software Engineering (ICSE*

[2]https://hpi.de/en/research/cooperations-partners/research-program-designing-for-sustainability.html

'02). Association for Computing Machinery, New York, NY, USA, 119–129. https://doi.org/10.1145/581339.581357

[6] Yuning Mao, Pengcheng He, Xiaodong Liu, Yelong Shen, Jianfeng Gao, Jiawei Han, and Weizhu Chen. 2021. Generation-Augmented Retrieval for Open-domain Question Answering. https://doi.org/10.48550/arXiv.2009.08553 arXiv:2009.08553 [cs]

[7] Toni Mattis and Robert Hirschfeld. 2020. Lightweight Lexical Test Prioritization for Immediate Feedback. *The Art, Science, and Engineering of Programming* 4, 3 (Feb. 2020), 12:1–12:32. https://doi.org/10.22152/programming-journal.org/2020/4/12

[8] Toni Mattis, Patrick Rein, Falco Dürsch, and Robert Hirschfeld. 2020. RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*. Association for Computing Machinery, New York, NY, USA, 385–396. https://doi.org/10.1145/3379597.3387458

[9] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. 2012. A Static Approach to Prioritizing JUnit Test Cases. *IEEE Transactions on Software Engineering* 38, 6 (Nov. 2012), 1258–1275. https://doi.org/10.1109/TSE.2011.106

[10] Dominik Meier, Toni Mattis, and Robert Hirschfeld. 2021. Toward Exploratory Understanding of Software Using Test Suites. In *Companion Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming (Programming '21)*. Association for Computing Machinery, New York, NY, USA, 60–67. https://doi.org/10.1145/3464432.3464438

[11] Nikhil Pinnaparaju, Reshinth Adithyan, Duy Phung, Jonathan Tow, James Baicoianu, , and Nathan Cooper. [n. d.]. Stable Code 3B. https://huggingface.co/stabilityai/stable-code-3b

[12] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming - Design and Implementation of an Integration of Live Examples Into General-purpose Source Code. *Art Sci. Eng. Program.* 3, 3 (2019), 9. https://doi.org/10.22152/programming-journal.org/2019/3/9

[13] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. https://doi.org/10.48550/arXiv.2308.12950 arXiv:2308.12950 [cs]

[14] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. 2015. An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 268–279.

Received 2024-02-29; accepted 2024-03-11

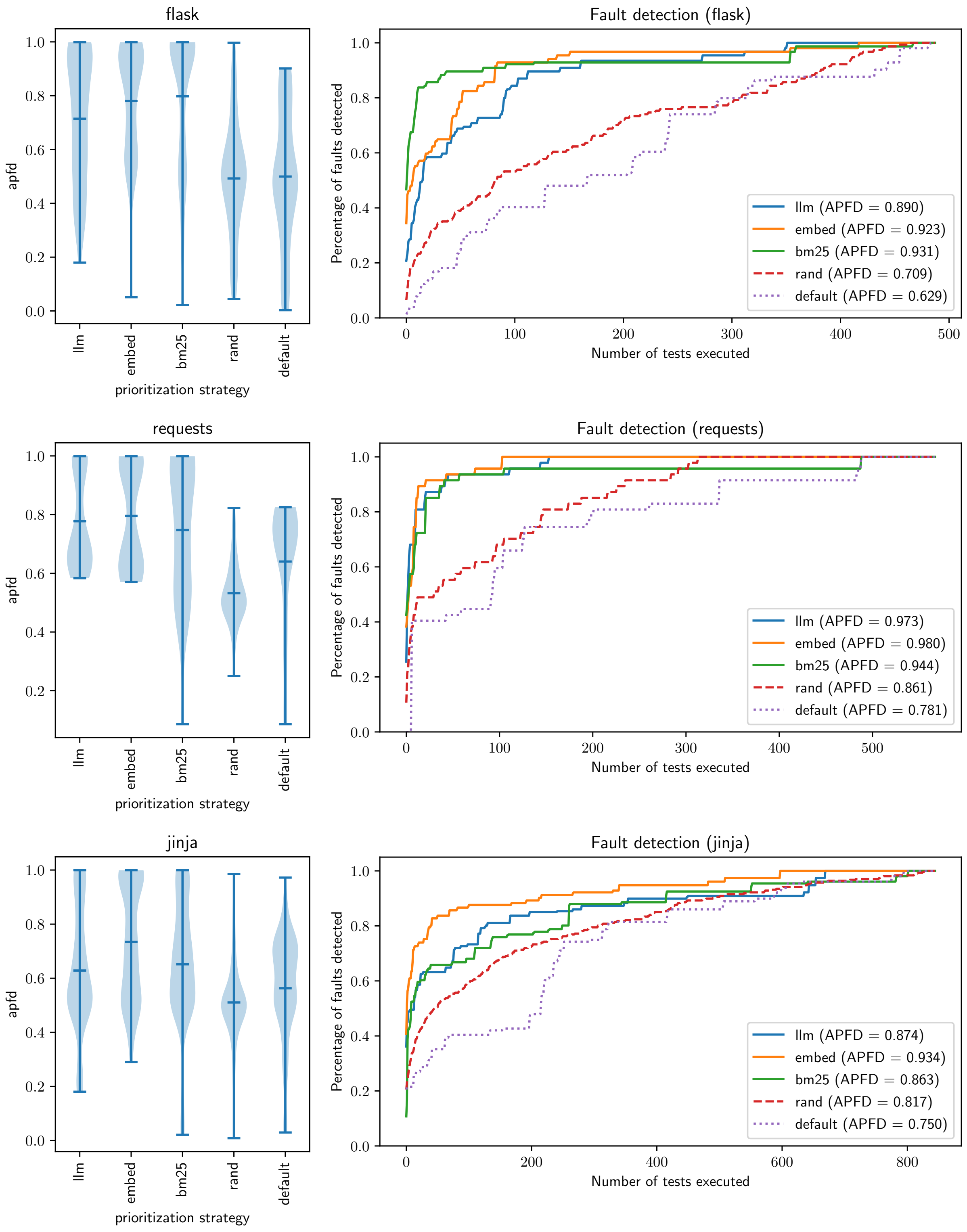

Figure 6: The distribution over micro-APFDs (left) and the fault detection over test runs including macro-APFD (right). Embedding results are only reported for the chunked embedding.