

Reducing Wasteful CI/CD Testing with LLM-Based Regression Test Selection

CPSC 502.06 Research Proposal

Muhammad Ahmad

muham.ahmad@ucalgary.ca

Undergraduate Honours Computer Science Student
University of Calgary
Calgary, Canada

Dr. Mahmoud Alfadel

mahmoud.alfadel@ucalgary.ca

Assistant Professor, Faculty of Science
University of Calgary
Calgary, Canada

Abstract

Large software projects usually run thousands of tests after every code change commit. This may give confidence, but it can also waste time and computing power when most of these tests are unrelated to the change itself. We propose an LLM-based regression test selector that reads a commit diff and chooses a smaller set of existing tests to run. The main goal is not only to save time. The selected set must still include the tests that reveal real regressions. We plan to evaluate the method on reproducible Java and Python CI cases, using BugSwarm as our main dataset. The full-suite results will act as the answer key. We will also compare the LLM with simpler text-ranking methods and measure failing-test recall, missed-failure rate, test reduction, run-time savings, and LLM cost. This study will show whether this method offers a useful safety-efficiency trade-off and whether or not the same idea works across two programming languages.

CCS Concepts

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

Keywords

regression test selection, test case prioritization, large language models, CI/CD, software testing

1 Introduction and Motivation

Firstly, continuous integration and continuous delivery (CI/CD) pipelines test every new change before the commit is merged or released. As a project grows, its test suite also grows. Running the full suite after every single commit can become slow and expensive. Next, it uses developer time, CI machines, energy, and money even when a change affects only a small part of the overall system. Slow feedback also makes it harder for developers to fix problems quickly.

Regression test selection (RTS) is a solution which tries to reduce this waste. It chooses which existing tests should run after a change. Traditional methods often use code coverage, dependency graphs, or call graphs. These methods can work well, but they may require language-specific tools and accurate build information. They can also be difficult to maintain across different projects.

Large language models (LLMs) may offer another way to make this choice. An LLM can read the meaning of a code change and the purpose of a test. This may help it connect a test to a change even when the link is not obvious from file names alone. However,

skipping a useful test is also risky. A fast pipeline is not valuable if it misses the test that would have revealed a regression. Hence, the project must treat safety as the first goal and efficiency as the second.

The central research question is: Can an LLM-based regression test selector reduce wasteful test execution in a CI/CD pipeline without missing tests that reveal regressions?

2 Background and Related Work

Currently, existing work shows that code meaning can help with test selection and prioritization. Yim proposes asking an LLM to choose tests from descriptions of a code change and labelled tests; however, the work does not report a full empirical evaluation [8]. TestPrune uses an LLM to find a very small set of old tests for software-repair tasks. It reports large reductions in test count and run time, but it starts from issue descriptions rather than completed commit diffs [2]. AgentTCP uses several agents, coverage, history, and code-change information to rank Java tests, but it only prioritizes tests rather than deciding which tests are safe to skip [1].

Two more studies may especially be useful for the design of this project. Mattis et al. compare an LLM with BM25 and code embeddings on Python changes. Simple ranking methods often perform as well as or better than the pretrained LLM. They also find that splitting a change into smaller chunks can improve the ranking [5]. Mahmoud et al. use LLM-generated commit descriptions as input into machine-learning test prioritization. Their results suggest that a clearer change description can help, but the effect varies by method [4]. These findings mean that an LLM should not be tested without strong, cheaper baselines.

Test case prioritization and regression test selection are related, but they solve different problems. Prioritization orders all tests so that the useful tests run earlier. Selection makes a direct run-or-skip choice. Our project uses prioritization as an intermediate step, where the model gives each test a relevance score, and a cut-off turns that ranking into a selected subset.

The main research gap, however, is a direct and safety-focused test of run-or-skip decisions from a real commit diff. Much of the closest work only ranks tests, uses issue text, studies only one language, or does not measure the failures that selection misses. We will address this gap by using the commit diff itself, comparing the selected tests with the complete full-suite result, and reporting Java and Python results separately. The intended contribution is a safety-focused evaluation of direct test selection that measures missed failing tests, net CI savings, and cost using the same method across both languages.

3 Research Questions and Hypotheses

The main question is then further supported by three sub-questions:

- **SQ1 — Safety:** What proportion of the failing tests found by the full suite are included in the LLM-selected subset, and how often does that subset detect the regression at all?
- **SQ2 — Efficiency and cost:** How many tests and how much CI run time can be saved after the LLM’s own time and estimated price are included?
- **SQ3 — Comparison and generalization:** How does the LLM selector compare with simpler baselines, and are the results similar for Java and Python projects?

H1: The LLM selector will reduce the number of tests and total pipeline time while keeping a high failing-test recall.

H2 (optional): A hybrid method that combines an LLM’s relevance with simpler code or coverage information will miss fewer failing tests than an LLM-only method at the same test budget.

4 Proposed Method

4.1 Dataset and Case Selection

The main dataset will be BugSwarm. It contains real failed and passed CI builds from open-source Java and Python projects, and many cases include a preserved container environment [6]. The two languages appear in different repositories; a single repository is not expected to contain both. This is still useful because we can test the same selection method on projects from both language groups.

For each retained BugSwarm case, we will use the change associated with the failing build and the tests present at that failing revision. Cases where the link between the change and the test failure is unclear will be excluded.

First, we will run a small feasibility check, rather than settling on a larger sample, before we know which old builds still work. A case will be kept only when:

- the environment can be reproduced;
- the failure comes from a test rather than a build or dependency error;
- the failing tests can be named;
- the relevant code change is clear; and
- the full suite finishes within a practical time.

Flaky or unclear cases will also be removed. The exact final sample size will be reported and justified from this check.

Each case will use the tests that existed at that specific historical version. We will not take today’s test suite and run it on old code, because newer tests may just fail simply because their features did not exist yet. The selector will receive the commit diff and information from the matching test suite. It will not receive the pass/fail labels. The full-suite labels are kept only for evaluation.

If BugSwarm has only a few cases, we will use Defects4J for Java and BugsInPy for Python as backup datasets [3, 7]. These datasets have cleaner known bugs, but their diffs usually describe a bug fix rather than the change that introduced a regression. For this reason, they are backups rather than the first choice. A small change-based mutation set from Mattis et al. is a later option if needed, since it fits the commit-diff setting but uses artificial faults [5].

4.2 Selector Design

For each case, the system reads the commit diff and the existing tests. If the diff is large, it will be split by file, function, or diff hunk. The LLM will make a short summary of each part of the change. It will then compare the change with each test and return a relevance score and a short reason. The tests are ranked by this score. A test-count or time cut-off creates the final selected set. Testing several cut-offs, or budgets, will show how safety changes as more tests are skipped. Figure 1 summarizes this process.

The first version will use a fixed prompt and a low model temperature so that repeated runs are more stable. The model name, version, prompt, input size, output size, selection time, and estimated price will be recorded. The model will see only information that would be available when the CI job starts.

4.3 Comparison Methods

The evaluation will include some or all of the following comparisons:

- **Full suite:** the safest result and the answer key, but with no test reduction.
- **Random selection:** a basic check using the same number of selected tests.
- **BM25 text matching (optional):** ranks tests by shared words with the commit diff.
- **Embedding similarity (optional):** ranks tests by similarity between vector forms of the change and test text.
- **Direct LLM selection:** the main proposed method.
- **Hybrid selection, if feasible (optional):** combines the LLM score with available coverage or dependency information.

The random, BM25, embedding, and LLM methods will be compared at the same test budgets. This avoids giving one method an unfair advantage by allowing it to run more tests. Also, the full suite remains as a reference point rather than a competing reduced method.

5 Evaluation Plan

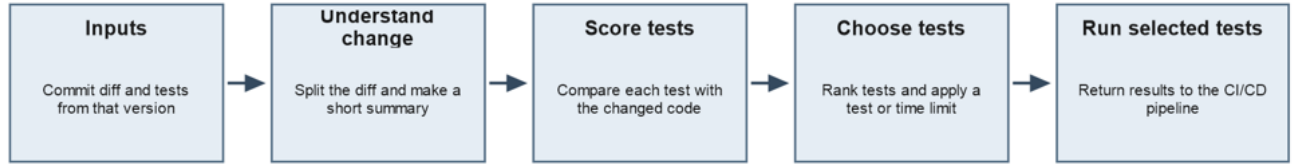
For each case, let (T) be all tests in that version, (F) be the tests that fail in the full suite, and (S) be the tests chosen by a selection method. Figure 2 shows how the two runs are compared.

Table 1 defines the main measures.

For example, suppose the full suite has 20 tests and 11 fail. The LLM chooses 11 tests: 9 failing and 2 passing. Failing-test recall is 9/11, or 81.8%. The missed-failure rate is 2/11, or 18.2%. Test reduction is 9/20, or 45%, because 9 of the 20 tests were skipped. The commit is still detected because at least one chosen test failed.

Failing-test recall is the main safety measure. Commit detection rate is also important, but it is less strict, as finding one failing test may detect the regression while still missing other failures. Another thing to note is that a singular regression may fail multiple tests, so a singular detection can still be useful, hence the recording of the commit detection rate. We will report both so that a high detection rate cannot hide weak recall. Efficiency will be judged only after LLM overhead is added. This prevents a method from appearing useful when the selection step takes longer than the tests it saves.

Proposed LLM-based test selection workflow



The model only ranks and selects tests. Full-suite results stay hidden and are used later as the answer key.

Figure 1: Proposed workflow from a commit diff to a selected set of tests.

How each historical case will be evaluated

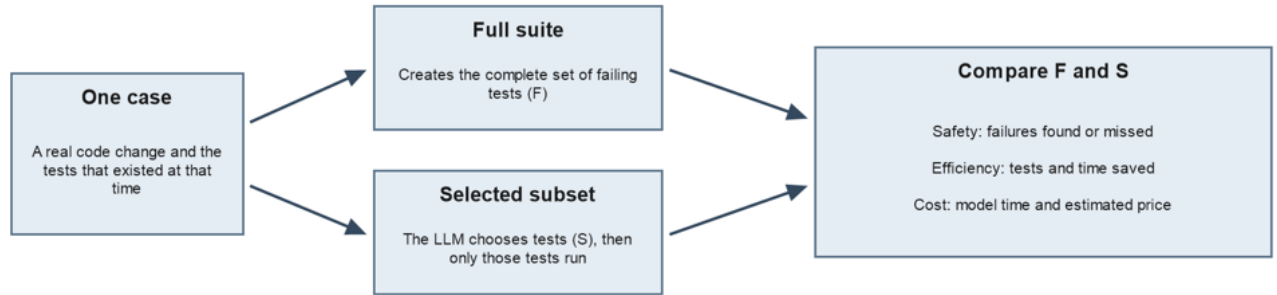


Figure 2: Full-suite results provide the answer key for evaluating the selected subset.

Measure	Calculation	Meaning
Failing-test recall	$ S \cap F / F $	Share of all failing tests that the selector included.
Missed-failure rate	$ F \setminus S / F $	Share of failing tests that the selector skipped.
Commit detection rate	Cases where $S \cap F \neq \emptyset$ / all cases	How often the selected set finds at least one failure.
Test reduction	$1 - S / T $	Share of tests that do not need to run.
Test-time savings	$1 - \text{selected-test time} / \text{full-suite time}$	Execution time saved after selection.
Net pipeline savings	$1 - (\text{LLM time} + \text{selected-test time}) / \text{full-suite time}$	Real saving after selection overhead is included.

Table 1: Measures used to evaluate safety, efficiency, and total pipeline cost.

We will report results for several test budgets rather than choosing only one cut-off. This will create a safety-efficiency curve. We will report the median and spread across cases, not only one average. Java and Python results will first be shown separately and then combined.

A method will be considered promising if it keeps high failing-test recall, detects nearly all faulty cases, and gives positive net pipeline savings. There is no universal safe recall threshold, so we will not claim that one cut-off fits every project. Instead, the results will show the cost of each level of safety and identify settings where the LLM is or is not useful.

6 Feasibility, Risks, and Limitations

The project is designed in stages so that it can still produce a useful result if one part is harder than expected. The minimum complete study is a reproducible dataset, the full-suite answer key, one LLM method, and the safety and efficiency measures above. Random and BM25 baselines, embeddings, and a hybrid method are extensions after this core study works.

The largest main risk is old CI cases that no longer build. The feasibility check and backup datasets are included to reduce this risk. Another risk is flaky tests, which can make the answer key unreliable. Cases with changing results will be removed or clearly marked. Some public repositories may have also appeared in an LLM's training data. This cannot be ruled out, so it will be reported

as a limitation. The model’s exact version and date will also be recorded.

BugSwarm cases may not represent every modern CI/CD pipeline. The study will also use a limited number of projects because each full-suite run can be expensive. The results will therefore show evidence for the tested cases, not proof for all software. Testing Java and Python separately improves the evidence for generalization, but it does not cover all languages or build systems.

7 Timeline

Table 2 gives the planned 32-week schedule.

8 Expected Outcomes and Impact

The main outcome will be evidence about whether an LLM can make useful run-or-skip decisions from a commit diff. The project will produce a selector prototype, a clear evaluation method, and a comparison against cheaper baselines. It will also show the trade-off between failures found and CI work saved at several selection sizes.

A positive result would support a practical way to shorten feedback in large CI/CD pipelines. A negative result would still be useful. It could show that simple text methods are safer or cheaper, that LLM overhead removes the time saving, or that the method works in one language but not the other. These findings can help future work avoid relying on LLM claims without measuring missed failures and total cost.

References

- [1] Junyu Chen, Xiuhua Li, Jinping Wang, Huan Xie, Chunyan Liu, Zhenyu Wu, and Yan Lei. 2025. AgentTCP: An LLM-Based Multi-Agent Framework for Test Case Prioritization. In *Proceedings of the 2025 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Los Alamitos, CA, USA, 634–645. doi:10.1109/APSEC66846.2025.00066
- [2] Yang Chen, Toufique Ahmed, Reyhaneh Jabbarvand, and Martin Hirzel. 2026. Can Old Tests Do New Tricks for Resolving SWE Issues? *Proceedings of the ACM on Software Engineering* 3, FSE (2026), 22 pages. Article FSE141. doi:10.1145/3808148
- [3] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*. Association for Computing Machinery, New York, NY, USA, 437–440. doi:10.1145/2610384.2628055
- [4] Yara Mahmoud et al. 2025. Enhancing Test Case Prioritization with LLM-Generated Commit Messages. In *Proceedings of the 21st International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. Association for Computing Machinery, New York, NY, USA, 45–54. doi:10.1145/3727582.3728681
- [5] Toni Mattis, Lukas Böhme, Eva Krebs, Martin C. Rinard, and Robert Hirschfeld. 2024. Faster Feedback with AI? A Comparative Evaluation of LLMs, Embeddings, and BM25 for Test Case Prioritization. In *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming*. Association for Computing Machinery, New York, NY, USA, 32–40. doi:10.1145/3660829.3660837
- [6] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE Press, Piscataway, NJ, USA, 339–349. doi:10.1109/ICSE.2019.00048
- [7] Ratnadira Widyasari et al. 2020. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Association for Computing Machinery, New York, NY, USA, 1556–1560. <https://github.com/soarsmu/BugsInPy>
- [8] Keun Soo Yim. 2024. Automatic Test Case Selection by Prompting a Large Language Model with Code Descriptions and Labeled Test Cases. Technical Disclosure Commons. https://www.tdcommons.org/dpubs_series/7296/

Weeks	Main work	Output
1–4	Check BugSwarm cases and confirm that tests, diffs, and environments can be reproduced.	Feasible case list and final scope.
5–8	Prepare the version-specific test information and full-suite answer keys.	Clean evaluation cases.
9–12	Build the LLM prompt, diff chunking, ranking, and selection steps.	Working LLM selector.
13–16	Add simpler comparisons and choose test-count or time budgets.	Comparable experiment settings.
17–20	Run the experiments and record test results, time, and model cost.	Complete raw results.
21–24	Calculate metrics and compare Java and Python results.	Tables, curves, and findings.
25–28	Write the report and review threats to validity.	Full draft.
29–32	Revise, check reproducibility notes, and prepare the final submission.	Final paper and project materials.

Table 2: Proposed project timeline.