Benchmarking of LLM generated unit-tests

Goal and motivation

Many developers rely on LLMs to generate unit tests. How to benchmark models and extend the scope of explainability into the world of machine generated code? Classic test adequacy metrics such as code coverage and assertion density. However, these often fall short in measuring behavioral correctness. Is LLM good in detecting faults, or is it just pointlessly maximizing coverage? Some may manually review tests, but this is impossible given a large dataset of generated code. Task becomes even more heavier considering every test needs an oracle to decide whether it should pass or fail. No widely adopted tool currently integrates failure-accepting mutation testing for LLM-generated test code. As AI-generated code becomes more prevalent, ensuring its correctness is critical for software reliability and customer trust. Developing tools that better benchmark test quality can improve internal LLM models in AI-assisted development tools. This supports objectives around product excellence, developer experience, and long-term scalability. This project introduces a benchmark framework based on Mutation Testing (MT) that accepts test failures and uses a novel cosine similarity-based test differentiator. This framework evaluates AI-generated tests by their ability to detect artificially injected faults, even when the expected output is unknown. A modified fork of the MutPy library includes per-test mutation scores and RAPFD metrics for prioritization, utilizing the cost effectiveness with real world limits.

Mutation Analysis

Every test suite is a potential candidate to be mutation analysed. If a candidate is identified as invalid (MA cannot be performed), MA is skipped and the result counted into MA statistics. If candidate breaks in any way MA pipeline (due to resource leaks), the sample (of all three test suites per each model; regardless which one specifically caused the shutdown) is discarded from the run and is not counted in the final statistics.

Given specific SUT t, mutation is observed (i.e. killed by t) whenever:

- t passes against OI and t fails on MI
- t fails against OI and fails against MI and the output similarity of these two results is less then θ [1]

Given specific SUT t, mutation is not observed (i.e. survives t) whenever:

- t passes against OI and passes against MI
- t fails against OI and t fails against MI and the output similarity of these two results is greater than or equals to θ

Following cases are ignored, as they are assumed to occur randomly or in small numbers:

- t fails against OI and passes on MI

[1] θ is passed as a parameter to modified MutPy fork

- t times out either for OI or MI
- MI is incompetent

Mutation Testing

By default, MutPy generates all possible mutants based on the available mutation operators. In order to reduce sample space, second order HOM strategy `BETWEEN_OPERATORS` (BTO) is used. Derezinska and Halas [1] implements BTO strategy as a combination of two not-used FOMs. The first FOM is taken from the top of the list. Second FOM is the first FOM that has been generated with a different mutation operator that the first selected mutant. If these FOMs have direct relation in code AST (parent - child or vice versa) or they are the same mutant operators, they are considered indifferent and not selected for BTO.

Metrics

For every TS, metrics described in following sections are calculated using modified MutPy fork.

Mutation Score

Denote test suite $TS \in T$ [3], its generated mutants as M_{TS} and the killed mutants as K_{TS}, $K_{TS} \in M_{TS}$. Notice that $M, K \in \mathcal{M}$

For every generated test suite (per each inspected LLM) generated for Rosetta task implementation, Mutation Score is computed

$$\text{MS}(TS) := \frac{K_{TS}}{|M_{TS}|} \tag{0.1}$$

Per-Test Mutation Score

To allow automatized granular evaluation for every unit test case in generated test suite, modified version of mutation score called p̈er-testm̈utation score is computed for each unit test t_i

$$\text{PTMS}(TS, t_i) := \frac{K_{t_i}}{|M_{TS}|}, t_i \in TS \tag{0.2}$$

Note that K_{t_i} are not mutually exclusive; therefore, the "per-test" scores do not sum up to 1.

Relative weighted Average Percentage of Faults Detected Score (RAPFD)

Regarding test prioritization evaluation, two RAPFD scores for testing contsraint $m = 5$ are computed for each generated test suite:

- real RAPFD of the actual ordering in generated TS
- randomized RAPFD

To explain how RAPFD formula is computed, for start APFD is described. Although Rothermel, Untch, Chu, and Harrold [2] first proposed the APFD metric, we will use notation from [5] as a base intuition, which will be later extended to RAPFD. [2]

$$\text{APFD}(TS) = 1 - \frac{\sum_{\phi \in \Phi} \text{TF}(\phi, TS)}{|TS||\Phi|} + \frac{1}{2|TS|}.$$

where

- $\phi \in Phi$ set of faults in software
- $\text{TF}(\phi, TS)$ index of test $t \in TS$ that **first** detected ϕ

Wang, Fang, Chen, and Zhang [5] then defines RAPFD as follows:

$$\text{RAPFD}(TS, \gamma) = p(\gamma) - \frac{\sum_{\phi \in \Phi} \text{RTF}(\phi, TS, \gamma)}{\gamma \times |\Phi|} \tag{0.3}$$

where

- $\gamma \in \mathbb{N}$ is a test resource constraint (i.e. how many unit tests are allowed to be run due to limited resources)[3][4]
- RFT is constrained TF from previous APFD,

$$\text{RTF}(\phi, TS, \gamma) = \begin{cases} \text{TF}(\phi, TS) & \text{if } \gamma \geq \text{TF}(\phi, TS), \\ 0 & \text{else} \end{cases} \tag{0.4}$$

- $p(\gamma)$ discrete cumulative step function of detected faults
  i.e. number of faults detected by first m to the overall number of faults in Φ

$$p(\gamma) = \frac{|\{\phi \in \Phi \mid \text{RTF}(\phi, TS, \gamma) \neq 0\}|}{|\Phi|}. \tag{0.5}$$

For the comparison study, the metrics $\text{RAPFD}(TS_i, 5)$ and $\text{RAPFD}(TS'_i, 5)$ will be computed for each test suite TS_i and its corresponding reordered version TS'_i.

Test Differentiator

Test diffentiator is computed with parameter $\theta = 0.95$ as

$$d(t, p_x, p_y) = \begin{cases} \mathit{false}, & \text{if } p_x == p_y \\ \text{cos\_sim}(p_x, p_y) \leq \theta, & \text{otherwise} \end{cases}$$

[2]Note that TS is used to denote test suite instead of original σ from mentioned research

[3]Originally Wang, Fang, Chen, and Zhang [5] uses m to denote this bound-type constraint

[4]often chosen as $\gamma = \frac{\sum_{i=1}^{n} |TS_i| * \delta}{n}$ where $delta$ specifies reduction rate $\delta = (0, 1)$.

LLM Score Rankings

Based on computed metrics from and subsequent insights from exploratory data analysis, score ranks are created for each of the inspected LLM. Depending on the metric or insight indicator, ranking will be assigned as follows:

- from 1 (best performing) to 3 (worst performing) if metric/insight has **positive** indicator, i.e. less score points indicate better qualitative outcome
- from 1 (least negative quality) to 3 (most negative quality) if metric/insight has **negative** indicator, i.e. higher score points indicates worse qualitative outcome

All of the scores are Borda counted into a final one, and since inverted counting system is used, models are ranked best to worst based on least to most scored points. If two final rank scores result in the same value, lexicographic Tie-Break (TB) is used for resolution. Winner is selected as the entry with most top positions wins. If top positions wins are of equal amount, second position is compared, and so on.

Implementation

MutPy module source code was modified:

- `mutpy/commandline.py` additional parameters for CLI were implemented (RAPFD constraint, eda folder location, theta factor)
- `mutpy/views.py` custom print functions for per-test metrics and per-mutant stast were added
- `mutpy/test_runners/base.py` unittest run modified to accept both multiple (original) test failures and original passed as mutation killers
- `mutpy/test_runners/unittest_runner.py` support test order for RAPFD metrics
- `mutpy/controller.py` - main implemented logic, accept test failures, compute RAPFD and APFD, test output comparator using spacy cosine similarity, eda statistics data handling, modified mutation score for per-test and test failures

Docker Setup

1. `docker build -t my-mutation-test .`
2. `docker run -env-file .env my-mutation-test`
   Run Docker image. You can modify MA/MT run parameters by modifying the attached `.env` file.
3. `docker ps -a`
   Find container ID.
4. `docker cp <container_id>:/app ./app_output`
   Copy script output to `app_output` or other desired location.

Results

In Docker output (copied to `app_output`), there is a duplicated project along with extracted data under the `eda` folder. For the original project Evaluation run, see results in `Evaluation/eda` folder in the `Evaluation` directory. You can compare these with yours in `app_output/eda`.

Mutation Analysis Results

- `eda/MT_EDA_output.ipynb`: Mutation Testing postprocessing, EDA and evaluation
- `eda/MA_EDA_output.ipynb`: Mutation Analysis run analysis
- `eda/mutpy_results.csv`: Data for all files subjected to MA
- `eda/per_suite.csv`: MT metrics per suite
- `eda/per_test.csv`: MT metrics per test
- `eda/per_mutant.csv`: Mutant generation data

For the MA/MT inspected dataset (with `reference_data/small_data` set as default), output files are generated for each LLM. For example:

`/home/xtuchyna/git/gen-test-bench-simulation/reference_data/small_data/animation/test_deepseek_coder_an`

These output files are already available for the original Evaluation run, but will be overwritten by your run.

Note: There are also two clean Jupyter notebooks: `eda/MT_EDA.ipynb` and `eda/MA_EDA.ipynb`.

Visualizations

`eda/images` contains all visualizations mentioned in the project.

Test Data

You can perform MA on test data first or a small subset. Modify the `BASE_FOLDER` variable in the `.env` file (commented samples are provided).

Experimental MutPy Attached

An experimental fork of MutPy is included in the `mutpy` folder. Pipenv uses it directly for pip installation.

See: https://github.com/xtuchyna/mutpy

Pull request: https://github.com/xtuchyna/mutpy/pull/1

Important Notes

Resource Hogs

Several scripts from the original Rosetta dataset may cause memory leaks or other issues. These are stored in the `reference_data/resource_hogs` folder.

Python Version Requirement

Experimental MA is supported only for Python 3.12 and above due to the use of `addDuration()` in unittest runs. Docker is pre-configured with Python 3.12.

Common Issues

Spacy Download Error During Build:

```
ERROR: failed to solve: process "/bin/sh -c pipenv run python3.12 -m spacy download
    en_core_web_sm" did not complete successfully: exit code: 1
```

Solution: Re-run the build command.

Evaluation

Metrics Correlations

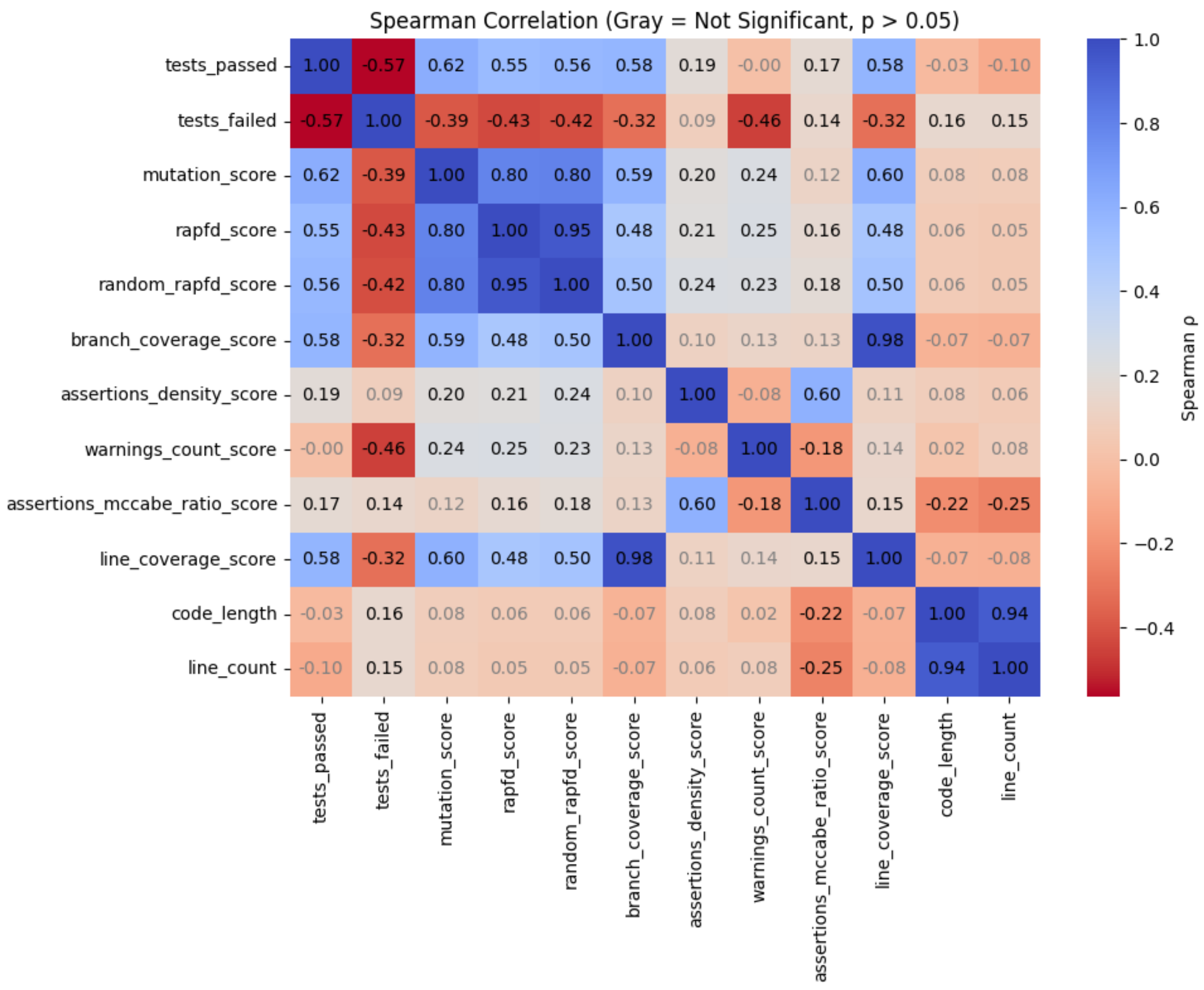

Figure 1: Spearman correlation heatmap

New introduced metrics (Mutation Score and RAPFD) show positive moderate correlation to classic code coverage metrics as Branch and Line coverage. Weak positive correlation is found with respect to assertions density score and warnings count score. From previously computed test adequacy metrics [4], Assertions McCabe Ratio Score asserts the weakest positive correlation (although it is statistically insignificant for Mutation Score). Code length and line count attributes of OI have strong non-monotonic correlations, however statistically insignificant. Due to its design, RAPFD score depends heavily on Mutation Score and therefore strong monotonic relationship is present. Clearly, passed and failed tests indicate strongly whether tests are of good quality or not, specifically for failed tests. All of the resulting correlations can be seen in 1

Mutation Analysis

As shown in 2, MA was partially successful on selected Python dataset, with most test suites (per each LLM) being able to be Mutation Tested. With respect to unsuccessful MA attempts, following culprits are observed:

- User input dependent programs
- Resource hogs that killed the MA script
- Python errors that prevented a `unittest` run (such as `ModuleImport` errors due to incorrect filename and etc.)

Based on overall number of results, each model is ranked in 1 with respect to each attribute indicator (positive result - ranked most to least; negative result - ranked least to most).

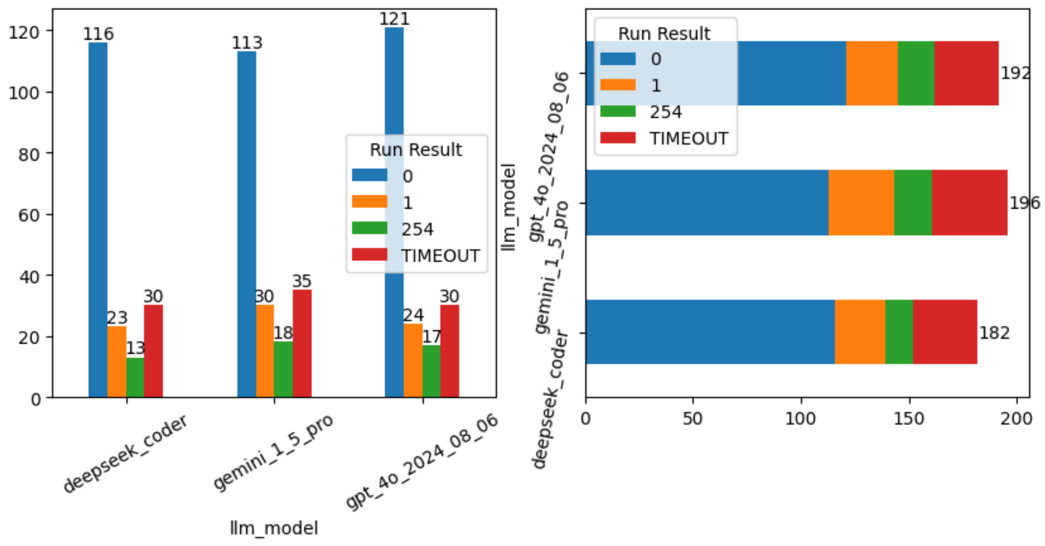

Figure 2: MutPy run statistics per each model.
0: Success
1: Error
254: Error
TIMEOUT: Per-File MA exceeded time limit

LLM	**MA Success (S1)**	**MA Timeout (S2)**	**MA Errors (S3)**
GPT	121 (1)	30 (1)	41 (2)
Gemini	113 (3)	35 (2)	48 (3)
DeepSeek	116 (2)	30 (1)	36 (1)

Table 1: MA results with corresponding S1,S2 and S3 ranks.

Mutation Testing

For all successful MA attempts (i.e. all valid test suite files), experimental MT provided multiple insights per each investigated model. 3 shows overall aggregation of passed and failed unit tests against OIs. Since this project uses test failures to its advantage (they are understood to be specifications of original program set to fail), overall executed tests **volume** is ranked in 2, its indicator being set to positive.

LLM	**OI Passed+Failed tests (S4)**
GPT	614 (1)
Gemini	606 (2)
DeepSeek	599 (3)

Table 2: Score table **S2**.

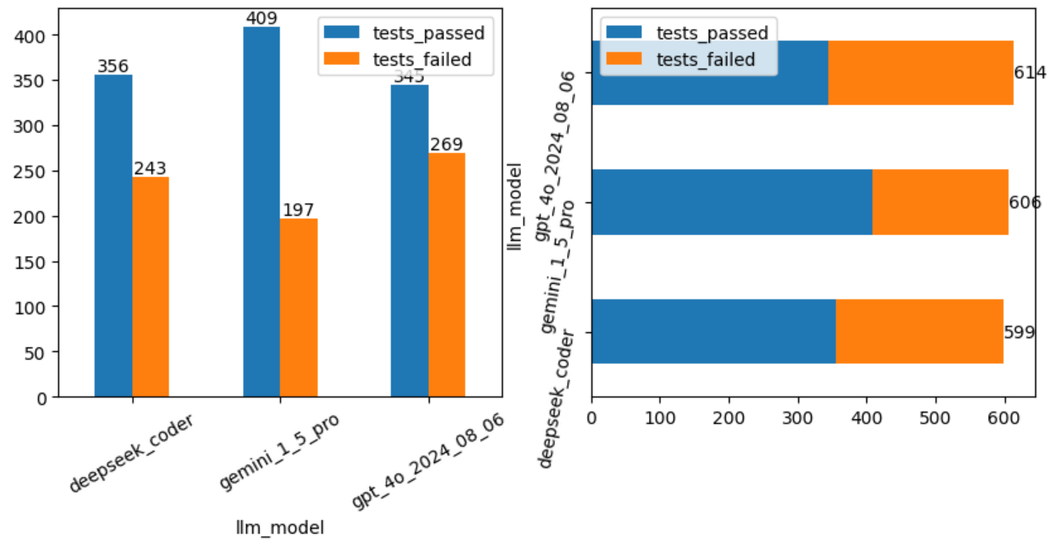

Figure 3: Overall test passed and failed per each model

Overall MT outcomes per each model is shown in 4. Since the most meaningful information is the number of killed and survived mutants, LLM models are ranked based on killed and survived percentages with respect to all mutants generated of each LLM.

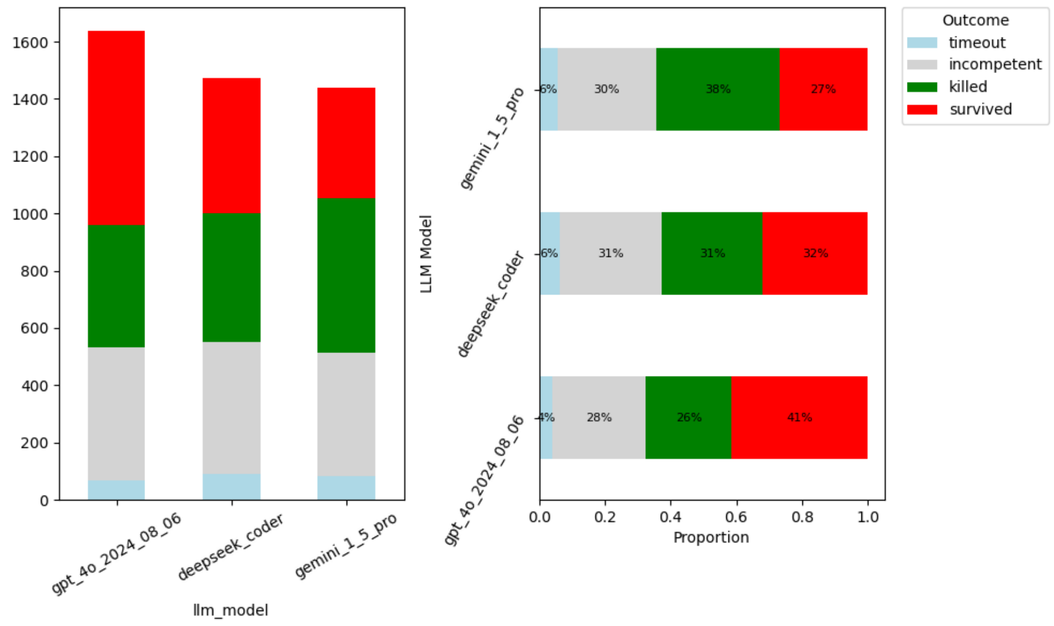

Figure 4: Overall mutants generated and their ratios per each LLM model

LLM	Killed Mutants (S5)	Survived Mutants (S6)
GPT	26% (3)	41% (3)
Gemini	38% (1)	27% (1)
DeepSeek	31% (2)	32% (2)

Table 3: Mutant Testing outcomes with models ranked

Distribution of the mutation scores is shown in 5, one boxplot per each investigated model. Although Gemini and DeepSeek medians are close to each other, the lower IQR for DeepSeek indicates worse performance. Models are ranked in 4

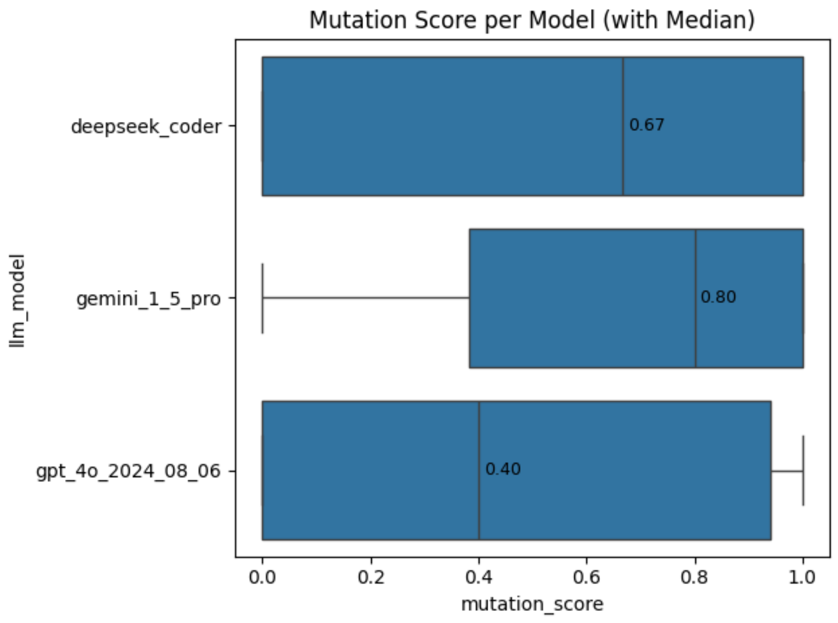

Figure 5: Boxplot showing the distribution of mutation scores per model. The box represents the interquartile range (IQR), the line inside is the median, and whiskers extend to non-outlier min/-max values.

LLM	Achieved Mutation Score Median (S7)
GPT	40% (3)
Gemini	80% (1)
DeepSeek	67% (2)

Table 4: Mutation Score with ranking **S7**.

Distributions of real and random RAPFD scores can be seen in 7. Randomized variant always performs slightly better than the real ordering. Since RAPFD metric depends heavily on Mutation Score, each model is also ranked in 6 based on its own RAPFD performance against randomized version, shown in 7. Distance of random outcomes is computed with respect to their real values and proportions for above (better than real RAPFD) and below (worse than real RAPFD) are calculated. Better random RAPFD outcomes are therefore treated as a negative indicator.

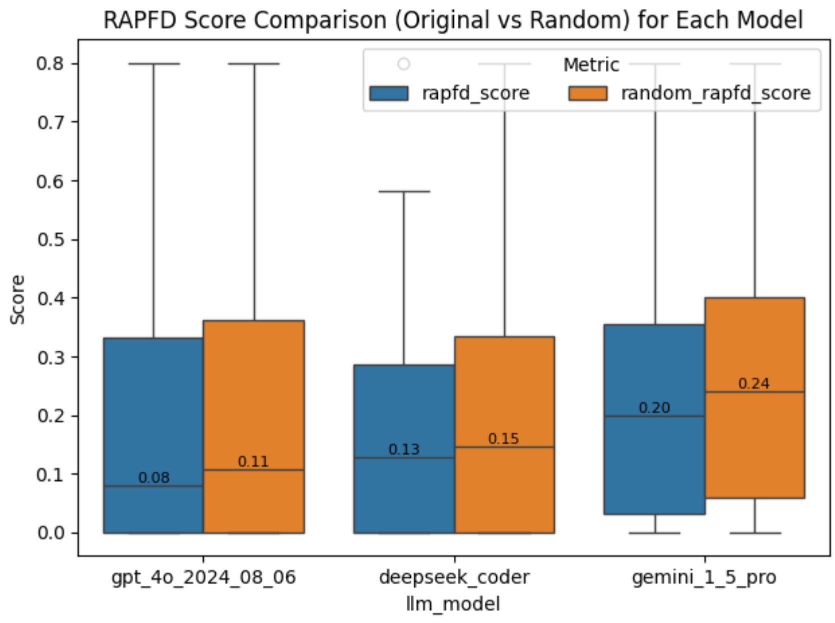

Figure 6: Boxplots showing RAPFD vs Randomized RAPFD per model.

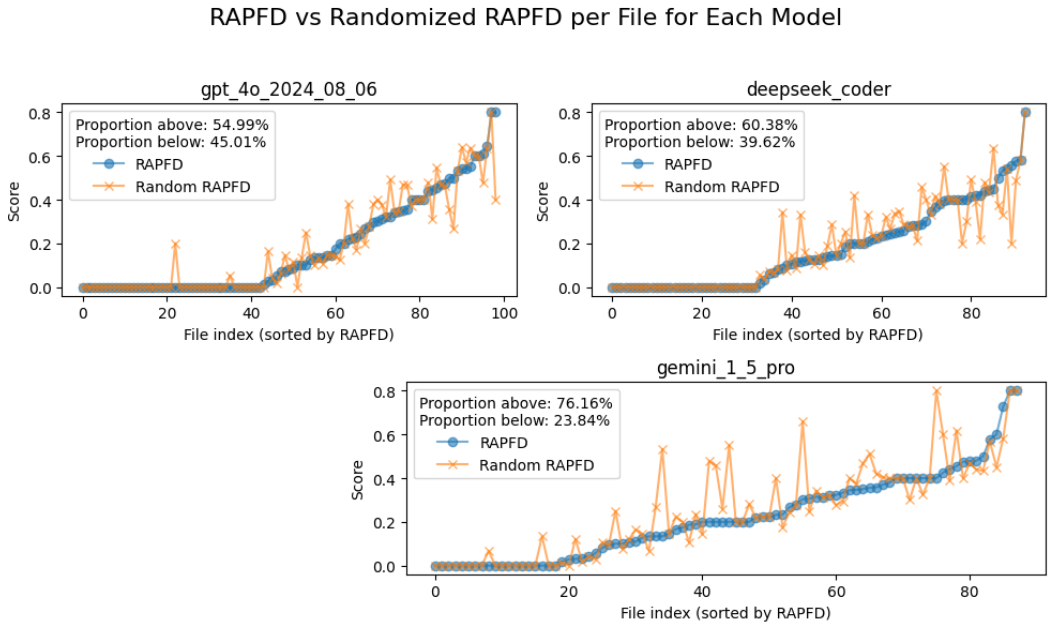

Figure 7: Random RAPFD with respect to its original RAPFD (sorted).

LLM	**Real RAPFD Median (S8)**	**Better Random RAPFD outcomes (S9)**
GPT	8% (3)	55% (1)
Gemini	20% (1)	76% (3)
DeepSeek	13% (2)	60% (2)

Table 5: RAPFD related scores with rankings S8 and S9.

Overall Model Ranking

Borda count was calculated based on all aggregated scores and final ranks evaluated GPT to have best overall performance, DeepSeek being the second best overall performance and Gemini to be the last one.

LLM	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**	$\sum$	**Final Rank**
GPT	1	1	2	1	3	3	3	3	1	18	**3**
Gemini	3	2	3	2	1	1	1	1	3	17	(TB) **1**
DeepSeek	2	1	1	3	2	2	2	2	2	17	**2**

Table 6: Final (Borda counted) LLM Rank. Gemini wins by Tie-Break

Results and impact

This project explored the feasibility of applying automated Mutation Anal- ysis (MA) to LLM-generated test code, highlighting its advantages over traditional test adequacy metrics. While LLM models can generate plau- sible test code that achieves reasonable results on conventional adequacy measures, their practical fault-revealing potential is not guaranteed, especially considering physical limitations and the availability of real test oracles. Best performing overall was Gemini, which had also the highest mutation score achieved - 80%. Worst overall was GPT, even though it seemingly generated the most runtime-adequate tests. DeepSeek emerges as an alternative in the middle. Furthermore, this project experimented with novel approach of using test failures as program specifications rather than unwanted errors to its advantage, which positively impacts the analysis as broader scope of data that can be included, explored and analyzed. Experimental implementation was successful and evaluation without known test oracles was possible. Modified MutPy fork is available for python installation along with data and artificial testing examples to verify the experimental setup using test differentiator (cosine similarity) enabling test failures to be counted in Mutation Testing. New additional metrics were implemented as Per-Test and RAPFD scores enabling testing benchmark to be extended. Per-Mutant statistics aggregation was also implemented

Bibliography

1. A. Derezinska and K. Halas. “Experimental Evaluation of Mutation Testing Approaches to Python Programs”. In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. 2014, pp. 156–164. DOI: `10.1109/ICSTW.2014.24`.
2. G. Rothermel, R. Untch, C. Chu, and M. Harrold. “Prioritizing test cases for regression testing”. *IEEE Transactions on Software Engineering* 27:10, 2001, pp. 929–948. ISSN: 1939-3520. DOI: `10.1109/32.962562`.
3. D. Shin and D.-H. Bae. *A Theoretical Framework for Understanding Mutation-Based Testing Methods*. 2016. arXiv: `1601.06466 [cs.SE]`. URL: `https://arxiv.org/abs/1601.06466`.
4. A. Skysľaková. “Generating unit tests using LLM [online]”. SUPERVISOR : Marek Grác. Diplomová práce. Masarykova univerzita, Fakulta informatiky, Brno, 2025 [cit. 2025-04-20]. URL: `https://is.muni.cz/th/p4zjp/`.
5. Z. Wang, C. Fang, L. Chen, and Z. Zhang. “A Revisit of Metrics for Test Case Prioritization Problems”. *International Journal of Software Engineering and Knowledge Engineering* 30:08, 2020, pp. 1139–1167. DOI: `10.1142/S0218194020500291`. eprint: `https://doi.org/10.1142/S0218194020500291`. URL: `https://doi.org/10.1142/S0218194020500291`.