pytest-ranking: A Regression Test Prioritization Tool for Python

Runxiang Cheng
University of Illinois, USA
rcheng12@illinois.edu

Kaiyao Ke
University of Illinois, USA
kaiyaok2@illinois.edu

Darko Marinov
University of Illinois, USA
marinov@illinois.edu

Abstract

Regression Test Prioritization (RTP) can find test failures quicker and provide faster feedback to developers to help in debugging. While RTP has been researched for almost three decades, with many research techniques proposed, practical tools and evaluations are sporadic. We present PYTEST-RANKING, a robust tool for Python and its most popular testing framework Pytest. We evaluate our tool on 4,308 builds for 14 open-source Python projects running on the GitHub Actions CI. Our experiments show that our tool integrates well with the Pytest ecosystem, has a low runtime overhead, and finds test failures faster than the default and random order baselines.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Software testing, regression testing, test prioritization, Python

ACM Reference Format:
Runxiang Cheng, Kaiyao Ke, and Darko Marinov. 2025. pytest-ranking: A Regression Test Prioritization Tool for Python. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728587

1 Introduction

Developers frequently run regression test suites in Continuous Integration (CI) to expose potential faults in their project changes. Delays in obtaining test results from CI slow down development cycles and prevent timely feedback to the developers. Researchers have proposed Regression Test Prioritization (RTP), which aims to expose potential faults sooner by reordering tests in the test suite such that the tests more likely to fail are run earlier [18, 57]. Many RTP techniques have been developed, e.g., using test costs [2, 59], past outcomes [22, 72], code coverage [57, 58], information retrieval [60], and machine learning [1, 63]. These RTP techniques were shown effective on CI build datasets of both open-source and proprietary software [7, 8, 18, 20, 25, 27, 29, 30, 70].

Although many RTP techniques and datasets exist, there is no readily usable, open-source tool for running RTP in CI. Several research RTP tools have been developed, but mostly for Java where widely used testing frameworks change slowly, e.g., the test ordering extension for Surefire, the default test runner for the Maven build system, has been pending for 2.5 years [65]. Some prior studies have released scripts for RTP [1, 4, 8, 9, 63] but limited to reproducing evaluation results on research datasets; they provide no interface to integrate with common testing frameworks [68] or CI services. Lack of a readily usable tool makes it difficult for practitioners to adopt RTP research results in their CI practices and for researchers to evaluate their RTP innovations in widely used CI systems. Thusly motivated, we develop an RTP tool for Python, because it is one of the most popular programming languages, and Pytest, its most popular testing framework, is often more open to accepting new contributions than Maven Surefire.

Moreover, most prior studies [4, 18, 20, 27, 30] have only evaluated RTP techniques by *simulating* RTP orders on historical builds—they reorder tests from the original test suite runs (TSRs) of the build but compute RTP effectiveness on the reordered test suite with test outcomes and durations *copied* from the original build, *without actually executing* the reordered test suite. Such simulations can miss issues in test execution, e.g., the reordered test suite can have different test outcomes or durations [68], reordering can violate test-order dependency [24], and test parallelism can impact RTP effectiveness [74]. These are important issues, and their influence on RTP effectiveness should be evaluated.

This paper makes the following contributions:

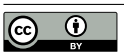
Core Tool: We develop PYTEST-RANKING, an RTP tool for Pytest, the most prominent testing framework for Python. We release its source on GitHub [13], binary on PyPI [38] for easy installation, and a demo video linked from https://zenodo.org/records/15149581.

Ecosystem Integration: We make PYTEST-RANKING easy to use as a Pytest plugin compatible with several other plugins for test selection, test parallelization, and test ordering. Moreover, PYTEST-RANKING is easily deployable, and we integrate it with 14 projects that use GitHub Actions, currently the most popular CI system.

Realistic Evaluation: We evaluate PYTEST-RANKING by actually rerunning historical builds on GitHub Actions, rather than just simulating the test outcomes and durations. Our analysis of test failures finds many flaky tests [26], which simulations would have missed. We also find PYTEST-RANKING effective, with a low overhead.

2 Implementation

We implement PYTEST-RANKING as a plugin for Pytest. Figure 1 illustrates how the components of PYTEST-RANKING interact with the core Pytest. PYTEST-RANKING has four main components: Ranker, Monitor, Extractor, and Reporter. When PYTEST-RANKING is enabled (①), Pytest provides the selected test suite to the Ranker (②) and receives a prioritized test suite to run (③). Monitor collects the relevant test data (test duration and test outcome) as each test finishes (④). Extractor processes the data when the entire test suite finishes and saves the data of the TSR into the cache directory provided by Pytest (⑤); note that Ranker will use the cached data to prioritize tests in subsequent TSRs (⑥). At the end of the TSR, Pytest reports a summary, accompanied with a summary from the Reporter (⑦).



FSE Companion '25, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728587

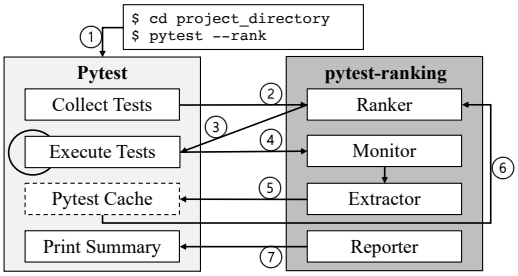


Figure 1: PYTEST-RANKING interaction with Pytest core.

2.1 Tool Components

To develop PYTEST-RANKING, we follow the best practices from the Pytest documentation and popular plugins built by Pytest developers [47, 48]. Specifically, we define an entry point for PYTEST-RANKING [56], so when Pytest is run, it can automatically discover and load the installed PYTEST-RANKING [45]. Once PYTEST-RANKING is loaded, it initializes a runner object of the class `RTPRunner` that contains custom implementations of Pytest hooks, which are Pytest APIs that a plugin can re-implement to change Pytest's default behavior [55]. PYTEST-RANKING then registers the runner to Pytest plugin manager [49], so that Pytest will execute `RTPRunner`'s implementations when running these hooks.

We next describe implementations of the main components.
Ranker re-implements a Pytest hook [50] that takes a list of test items from Pytest [43] and reorders it. Ranker can apply various RTP techniques for reordering. In Pytest, a *test item*, which we shorten to just *test*, is either a test function/method or a parametrized unit test (PUT) with a concrete parameter value. Each test has a unique ID based on its "collection address" [44], e.g.:

```
dir/test_module.py::test_class::test_function[value]
```

By default, Pytest orders the list hierarchically before passing it to Ranker: tests in the same file follow the definition order, and files follow the directory tree traversal order. Ranker sorts the tests based on their RTP score, as computed by various RTP techniques. Ranker uses a stable sort, i.e., it uses the initial order to break ties when multiple tests have the same RTP score.

From the myriad of prior RTP techniques [18], we implement three in Ranker—*QTF*, *RecentFail*, and *SimChgPath*—because recent research shows that simple RTP techniques often work better than more complex techniques, including machine-learning based techniques, especially when evaluated for realistic scenarios [4, 8, 32]. *QTF* prioritizes tests with a shorter runtime from their last runs; *RecentFail* prioritizes the recently-failed tests. *SimChgPath* prioritizes tests whose IDs are more textually similar to the paths of Python files changed since the last TSR; its change-awareness complements other techniques [32, 60]. PYTEST-RANKING tracks changed files between TSRs with file content checksums [17].

Ranker uses configurable weights to linearly combine values from all three RTP techniques into a test's overall RTP score, and then prioritizes tests by their scores. Linear combination implements a hybrid RTP that is more effective than any individual RTP technique [4, 32] and easier to scale with new techniques [64]. To combine different RTP techniques, Ranker normalizes the values to the $[0, 1]$ range for each technique [8, 70], ensuring that a higher value means a higher priority. For tests with no prior RTP data, i.e., likely new tests, Ranker assigns the highest priority.

Ranker allows the user to reorder tests at different granularities, from test values to functions, modules, or directories. Ranker is hierarchy-aware [24, 68], e.g., if a user runs module-level RTP, Ranker identifies a test's parent module m via its ID, computes the average score from tests in m as the RTP score of m, and prioritizes on modules; tests in each module follow the Pytest initial order.
Monitor re-implements a hook [51] to collect test execution data. After each test finishes, Pytest calls Monitor with a `TestReport` object [54] of this test as input. A `TestReport` has attributes such as execution duration and outcome. To minimize runtime overhead, Monitor only appends each `TestReport` to an in-memory list in `RTPRunner`; Extractor processes these reports when the TSR finishes.
Extractor saves the RTP data of the executed tests from `TestReports` to the Pytest cache [41]. The Pytest cache directory is in the target project's root [40]; Extractor creates a subdirectory to store mapping data for RTP techniques. For *QTF*, Extractor saves a key-value mapping from a test to its duration from the current TSR. For *RecentFail*, it saves a mapping from a test to how many times it has passed since its last failure, as computed from the previous cache value and the current outcome. *SimChgPath* data (file checksums) is handled by Ranker as change file set is only checked before a TSR. PYTEST-RANKING overwrites the previous cache with the new one, not saving data for historical TSRs to minimize cache usage.
Reporter adds a session to the Pytest terminal summary to report the configuration and overhead of PYTEST-RANKING [52].

3 Usage

PYTEST-RANKING is a Python package and can be installed and managed via `pip`. PYTEST-RANKING is invoked by setting Pytest's command-line options or configuration files [42]. For example, `pytest --rank` runs PYTEST-RANKING with the default configuration. PYTEST-RANKING has these configurable options:

- `--rank-weight` sets RTP technique weights (§2). If all weights are 0, PYTEST-RANKING randomly shuffles tests, i.e., *Random*.
- `--rank-level` sets the granularity level on which RTP is run in the test suite hierarchy [68]. Its value can be `put` (each PUT is treated as its own group), `function`, `module`, and `dir`. Test units in the same group follow the default order, while groups are reordered by the average RTP score.
- `--rank-hist-len` sets the largest value that can be recorded for each test for *RecentFail* [9].
- `--rank-seed` sets the random seed for *Random*.

If a CI workflow runs different builds in a fixed location, e.g., a project directory on a specific machine, users can deploy PYTEST-RANKING into CI without any additional setup. If the CI workflow always starts a new virtual machine to run a build, PYTEST-RANKING requires setting up the workflow to pass the Pytest cache data across builds. In our experience with GitHub Actions, the setup adds only 14 lines of YAML to a CI workflow file [10, 13].

PYTEST-RANKING works with test selection [47] and parallelization [39]. It also works with plugins for ordering tests [34, 35], by running ordered tests first in their declared order. Some Pytest options [53], or plugins that randomly order tests [37], can interfere with PYTEST-RANKING as they use the same reordering hook [55].

4 Experiment Setup

To evaluate PYTEST-RANKING, we collect and rerun builds from 14 projects that use Pytest and GitHub Actions.

4.1 Dataset Collection

Project Selection. We start from the 2,500 most downloaded projects in the year 2023 on PyPI [19, 33] and select candidate projects through metadata filtering and manual inspection.

First, we exclude projects that did not list a valid GitHub URL, a required Python version, or a required Pytest version in their PyPI metadata [19]—these steps yield 315 projects. Then, we exclude projects that did not run CI for their recent commits [15], and projects whose `git clone` takes over 1 minute (these are likely too large for our compute budget)—these steps yield 122 projects. To find popular projects with sufficient failures, we keep projects with over 1,000 GitHub stars, at least one failed TSR on the default branch—these steps yield 58 projects.

We inspect the top 40 out of the 58 most downloaded projects. We fork each project, find its latest CI build with passing TSR, rerun that build on the fork, and inspect the results. We include a project for evaluation if that build (1) ran tests on GitHub Actions (which provides uniform API access to test logs and artifacts [11]), (2) passed on `ubuntu-latest` and Python 3, and (3) has a TSR duration exceeding 45 seconds. Our inspection yields 12 projects.

We also select projects that already run tests randomly, as they likely have no test-order dependency [24, 73] and could be open to RTP. However, we found only 1 eligible candidate after applying the selection steps above to the 2,500 most-downloaded projects. To select more projects, we eventually searched the top 50,000 most-downloaded projects, found 91 projects that used related plugins [36, 37], excluded 72 after filtering, and selected 2 after inspection.

In total, we select 14 (12 + 2) projects to evaluate.

Build Collection. An RTP tool could be deployed to CI at a certain time and run on all builds from that time. Thus, for each project, we collect all builds [12] from 2024-01-01 to 2024-12-01; we order the builds chronologically by their start time [8, 66].

Some builds did not contain a TSR (e.g., package-release builds)—they cannot be used to evaluate RTP. We remove non-TSR builds by checking if the build is from a CI workflow file that runs tests: for each project, we manually find the CI workflow file that runs tests, collect the set of paths this file has historically taken via `git log`, and check if a build's workflow file path matches any of the collected paths. We only keep TSR builds that have a `completed` status, with a `success` or `failure` build conclusion [12]—we removed incomplete TSR builds (e.g., `cancelled`, `timed_out`, or `skipped` builds), because they are often ignored in development and would trigger reruns. We eventually collected 27,224 builds with TSRs, however, we did not evaluate them all to limit computation time (§4.2).

4.2 Experiment Procedure

Different from prior work, we do not only simulate RTP order on the historical builds (§1). Instead, we use PYTEST-RANKING to actually execute the RTP-ordered test suite of the historical build, on the code version that the build ran; then we study the test results (§5).

To rerun historical builds on a project, we first fork the project and set up PYTEST-RANKING in its GitHub Actions CI test run file (§3). For each build, we get the repository content archive at the commit of the build [14], overwrite the fork with the archive content except for the modified CI file, and push the code to GitHub. GitHub Actions automatically builds the pushed code with the modified CI file, and we finally download the produced TSR results. Rerunning historical builds could install later versions of the dependencies that are incompatible with the historical code version, and introduce unwanted failures. To eliminate such failures, we integrate the package manager UV with its `--exclude-newer` option to evaluated projects' CI workflows, so that UV installs dependency versions released before the start date of a given historical build [67].

To mimic how these projects would have run PYTEST-RANKING, we (1) make minimal change to their original CI file; (2) use the same GitHub Actions CI service; and (3) run builds with one environment they used, i.e., Ubuntu and Python 3. We also mimic build overlap: if build $i+1$ started after build i ended, we rerun $i+1$ after i finishes rerunning (so $i+1$ uses the cache updated by i); otherwise, we run both builds in parallel as $i+1$ would have used the cache prior to i.

For each build, we run 6 orders: 4 RTP orders (*QTF*, *RecentFail*, *SimChgPath*, and *Hybrid* that combines the prior three with equal weights) and 2 baselines (Pytest *Default* and *Random*). Each order runs as a separate CI workflow to avoid interference of Pytest cache. We run orders at the `function` granularity (§3): test functions are prioritized, but parameter values of the same PUT function follow the default order as they often have order dependency [46].

We do not rerun all 27,224 builds 6 times [16] but select all failed builds, and the first non-overlapping successful build before each failed build. The successful builds are used to create the right data cache for running RTP on the failed builds. For each project, we rerun its selected builds until reaching the last build or the 50^{th} build; we then rerun more builds until reaching at least 30 failed TSRs per project. We collected the TSR results as JSON-formatted test reports. We run each order once per build to (1) analyze test failures, and (2) compute effectiveness for all orders except *Random*. To compute effectiveness for *Random* [71], we rerun *Random* 10 additional times per originally failed build using rerun IDs as seeds.

5 Evaluation Results

Table 1 shows the statistics of our evaluation dataset. Following §4.2, we successfully reran 718 selected commits/builds from 14 projects and obtained 4,308 TSRs (6 TSRs per commit). Some selected commits failed to rerun, because they have errors that cause their test runs to exit early (e.g., commits from stale branches using unavailable, old dependencies). Of 4,308 TSRs, 1,121 had at least one test failure, 720 had at least one regression failure, and 660 had only regression failure. §5.1 presents our analysis of these failures. §5.2 discusses the effectiveness and overhead of PYTEST-RANKING.

5.1 Analysis of Test Failures

Flaky tests can pass or fail on the same code version, therefore they do not always indicate regression faults in CI [9, 19, 26, 31]. While research often uses "sanitized" datasets that carefully remove flaky tests, e.g., [21], real CI runs do have flaky failures as also reported by some prior work [4, 9, 32]. Our inspection finds three types of flaky tests: OD (order-dependent) [23], Concurrency (non-desirable worker interactions), and ND (non-deterministic outcome [5]).

Table 1: Evaluation dataset and results. Reg. means regression, RTP techniques are described in §2.1.

Project	Commits	TSRs			Failed tests		Test failures		Mean APFDc on regressions						Mean TSR size		Tool overhead	
		All	Failed	w/ Reg.	Flaky	Reg.	Flaky	Reg.	*Def.*	*Ran.*	*QTF*	*Rec.*	*Sim.*	*Hyb.*	Tests	Time (s)	Time (s)	Cache (KB)
aeon	49	294	121	120	9	341	21	3,307	0.48	0.50	0.72	0.72	0.48	0.70	32,295	2,202.2	0.21	538
ansible-lint	50	300	128	24	11	10	310	140	0.87	0.62	0.84	0.77	0.87	0.82	815	350.7	0.02	71
apscheduler	102	612	34	30	3	322	22	1,948	0.19	0.45	0.76	0.24	0.52	0.75	740	52.3	0.02	15
dask	51	306	65	48	9	118	32	943	0.44	0.52	0.80	0.48	0.41	0.67	12,613	802.1	0.07	216
dvc	49	294	86	30	9	5	76	42	0.54	0.53	0.70	0.70	0.53	0.75	2,693	306.1	0.02	81
ipython	42	252	171	42	83	2	2,257	42	0.58	0.44	0.99	0.87	1.00	1.00	1,422	91.2	0.08	37
librosa	32	192	30	30	0	10	0	90	0.25	0.48	0.77	0.63	0.45	0.83	13,752	450.9	0.42	138
molecule	50	300	54	54	0	7	0	66	0.67	0.48	0.71	0.76	0.76	0.88	458	182.3	0.07	35
networkx	50	300	84	36	6	123	103	750	0.61	0.49	0.97	0.81	0.93	0.99	6,073	125.7	0.18	155
pytest-django	51	306	56	36	2	21	25	138	0.57	0.52	0.60	0.59	0.73	0.68	219	59.1	0.01	7
pytest-xdist	34	204	105	102	1	12	3	150	0.43	0.49	0.55	0.88	0.47	0.92	208	105.4	0.01	7
pytorch-lightning	67	402	30	30	0	21	0	126	0.44	0.50	0.91	0.50	0.78	0.88	3,336	609.6	0.20	107
trimesh	49	294	78	78	0	26	0	204	0.29	0.49	0.86	0.57	0.54	0.91	599	325.4	0.02	22
ultralytics	42	252	79	60	13	17	240	357	0.27	0.50	0.75	0.53	0.82	0.80	73	524.7	0.02	6
Total or Mean	**718**	**4,308**	**1,121**	**720**	**146**	**1,035**	**3,089**	**8,303**	**0.47**	**0.50**	**0.78**	**0.65**	**0.66**	**0.83**	**5,378**	**442.0**	**0.10**	**103**

We semi-automatically examine all test failures. Tests that fail on some but not all orders for the same commit are potentially flaky. We rerun each such test on the same order multiple times in an attempt to reproduce the failure. If the failure is deterministically reproduced, it is likely OD, and we find the minimal test sequence that reveals OD flakiness [62]. If the test is not OD, we rerun it 1,000 times to check if it is ND. Other possibly flaky tests only fail with `pytest-xdist` [39], we run each such test concurrently with another test to confirm their Concurrency flakiness. For tests that fail on all orders for the same commit, we inspect the logs to determine whether the failures are due to regression changes.

In the 1,121 failed TSRs, we confirm 146 flaky tests and 1,035 regression tests. 10 projects have at least one flaky test. In Table 1, "Failed tests" shows the number of distinct tests; "Test failures" shows the number of test failures. Flaky tests are 7× fewer than regression tests but fail 2.6× more often per test, because flaky tests have a longer lifecycle, while regressions are detected deterministically and addressed more quickly [9]. 774 regression tests fail for only one commit. No regression test fails for over 9 commits. Only 9 flaky tests are fixed in any subsequent commit we examine.

Of all 146 flaky tests, 112 (77%) are OD. ODs are categorized into Victims (fail if run after *polluters*) and Brittles (pass only if run after *state-setters*) [19, 62, 69]. Of the 112 OD, 107 (95%) are Victims, and 5 (5%) are Brittles. The higher prevalence of Victims matches prior findings in Python and Java projects [19, 62]. Of the 60 polluter test sequences we confirm, 14 (23%) have more than one test. This percentage is greater than a prior study on Java (2%) [62]. Of the 34 non-OD, 6 are Concurrency and fail under `pytest-xdist`, a Pytest plugin that spawns multiple processes to run tests. This parallelism can lead to failures when, e.g., concurrently running tests try to use the same network port. The other 28 flaky tests are ND, due to non-deterministic behavior of certain APIs, e.g., random number generators [5]. The 10 additional *Random* reruns further expose 34 flaky tests (27 OD and 7 ND; 3 fixed) from 6 projects.

Implication. OD flaky test failures are not uncommon when running RTP in CI. Developers can effectively identify and annotate these tests [69, 73]. RTP tool can also help developers distinguish flaky tests when it is initially deployed or reporting test failures.

5.2 RTP Effectiveness

The most common RTP evaluation metrics are Average Percentage of Faults Detected per Cost (APFDc) and the simpler APFD [6, 18, 28, 72]. We use APFDc as it considers test runtime not just test count. APFDc is normalized to $[0, 1]$, so a 0.1 increase reduces the time to detect all faults by 10% of the TSR time. We compute APFDc with one-to-one and many-to-one failure-to-fault mappings [61]; they produce similar results, so we present only one-to-one.

Table 1 shows the mean APFDc across TSRs, considering only regressions from §5.1 as faults to be detected. Table 1 lists the name prefix of each technique: all RTP techniques (*QTF*, *RecentFail*, *SimChgPath*, *Hybrid*) outperform the baselines (*Default*, *Random*), being 38%–77% better than *Default*, and 30%–66% better than *Random*, on average across projects. *Hybrid* (§4.2) performs the best (0.83), 77% better than *Default* (0.47) and 66% better than *Random* (0.5). The same test suite can have different durations in different orders, which can affect RTP effectiveness [68]. We find that the magnitude of relative difference between the durations of a commit's *Default* TSR and its non-*Default* TSRs is 1%–10% (average 4%) across projects. TSR durations vary much less than individual test durations in the same commit, as individual test variations cancel each other out. Our additional results from simulations rank all evaluated RTP techniques in APFDc the same as Table 1 [3].

We also evaluate the overhead of PYTEST-RANKING. We collect its runtime from Reporter's terminal summary (§2), which includes the time for computing RTP data and reordering. The average runtime is 0.1 sec, or 0.03% of the TSR duration. We collect the Pytest cache sizes after the RTP-ordered TSRs finish, and compare the uncompressed sizes between the TSR cache and the CI log. The cache size is 6–538 KB, or 1%–9% (average 5%) of the CI log sizes.

Implication. PYTEST-RANKING finds faults faster with low overhead on Pytest test suites. Researchers may simulate RTP orders and should obtain similar RTP technique ranking in APFDc as reruns.

6 Conclusions

We presented PYTEST-RANKING, our RTP tool for Pytest test suites. Our evaluation on 4,308 GitHub Actions builds of 14 Python projects shows that our tool integrates well with the Pytest ecosystem, has a low runtime overhead, and finds test failures faster than baselines. We hope that these promising results will enable more research and eventually lead to adoption of RTP in practice.

Acknowledgments

NSF grant CCF-1956374 supported this work. We thank Sam Grayson and Erkai Yu for their help with UV.

References

[1] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-Rank vs Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration. In *ICSE*.

[2] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing Test Prioritization via Test Distribution Analysis. In *ESEC/FSE*.

[3] Runxiang Cheng. 2025. *Regression Test Prioritization for Modern Software*. Ph. D. Dissertation. University of Illinois Urbana-Champaign.

[4] Runxiang Cheng, Shuai Wang, Reyhaneh Jabbarvand, and Darko Marinov. 2024. Revisiting Test-Case Prioritization on Long-Running Test Suites. In *ISSTA*.

[5] Saikat Dutta, Anshul Arunachalam, and Sasa Misailovic. 2022. To Seed or Not to Seed? An Empirical Analysis of Usage of Seeds for Testing in Machine Learning Projects. In *ICST*.

[6] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. 2001. Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization. In *ICSE*.

[7] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *ESEC/FSE*.

[8] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration. In *ISSTA*.

[9] Emad Fallahzadeh and Peter C Rigby. 2022. The Impact of Flaky Tests on Historical Test Prioritization on Chrome. In *ICSE-SEIP*.

[10] GitHub 2025. Caching dependencies. https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/caching-dependencies-to-speed-up-workflows.

[11] GitHub 2025. GitHub actions. https://docs.github.com/en/actions.

[12] GitHub 2025. List workflow runs. https://docs.github.com/en/rest/actions/workflow-runs?apiVersion=2022-11-28#list-workflow-runs-for-a-repository.

[13] GitHub 2025. pytest-ranking. https://github.com/softwareTestingResearch/pytest-ranking.

[14] GitHub 2025. Repository archive. https://docs.github.com/en/rest/repos/contents?apiVersion=2022-11-28#download-a-repository-archive-zip.

[15] GitHub 2025. REST API endpoints for commit statuses. https://docs.github.com/en/rest/commits/statuses?apiVersion=2022-11-28#get-the-combined-status-for-a-specific-reference.

[16] GitHub 2025. Workflow usage limits. https://docs.github.com/en/actions/administering-github-actions/usage-limits-billing-and-administration.

[17] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight test selection. In *ICSE*.

[18] Renan Greca, Breno Miranda, and Antonia Bertolino. 2023. State of Practical Applicability of Regression Testing Research: A Live Systematic Literature Review. *Comput. Surveys* (2023).

[19] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An Empirical Study of Flaky Tests in Python. In *ICST*.

[20] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing White-box and Black-box Test Prioritization. In *ICSE*.

[21] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *ISSTA*.

[22] Jung-Min Kim and Adam Porter. 2002. A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments. In *ICSE*.

[23] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *ICST*.

[24] Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D Ernst, and Tao Xie. 2020. Dependent-Test-Aware Regression Testing Techniques. In *ISSTA*.

[25] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How Does Regression Test Prioritization Perform in Real-World Software Evolution?. In *ICSE*.

[26] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *FSE*.

[27] Qi Luo, Kevin Moran, and Denys Poshyvanyk. 2016. A Large-Scale Empirical Comparison of Static and Dynamic Test Case Prioritization Techniques. In *FSE*.

[28] Alexey G. Malishevsky, Joseph R Ruthruff, Gregg Rothermel, and Sebastian Elbaum. 2006. *Cost-Cognizant Test Case Prioritization*. Technical Report.

[29] Toni Mattis, Patrick Rein, Falco Dursch, and Robert Hirschfeld. 2020. RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization. In *MSR*.

[30] Rongqi Pan, Mojtaba Bagherzadeh, Taher A Ghaleb, and Lionel Briand. 2022. Test Case Selection and Prioritization Using Machine Learning: A Systematic Literature Review. *ESE* 27 (2022).

[31] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *TOSEM* 31 (2021).

[32] Qianyang Peng, August Shi, and Lingming Zhang. 2020. Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization. In *ISSTA*.

[33] PyPI 2025. BigQuery. https://docs.pypi.org/api/bigquery.

[34] PyPI 2025. pytest-dependency. https://pypi.org/project/pytest-dependency.

[35] PyPI 2025. pytest-order. https://pypi.org/project/pytest-order.

[36] PyPI 2025. pytest-random-order. https://pypi.org/project/pytest-random-order.

[37] PyPI 2025. pytest-randomly. https://pypi.org/project/pytest-randomly.

[38] PyPI 2025. pytest-ranking. https://pypi.org/project/pytest-ranking.

[39] PyPI 2025. pytest-xdist. https://pypi.org/project/pytest-xdist/.

[40] Pytest 2025. Cache dir. https://docs.pytest.org/en/7.1.x/reference/reference.html#confval-cache_dir.

[41] Pytest 2025. Config cache. https://docs.pytest.org/en/7.1.x/reference/reference.html#config-cache.

[42] Pytest 2025. Configuration. https://docs.pytest.org/en/stable/reference/customize.html#command-line-options-and-configuration-file-settings.

[43] Pytest 2025. Item. https://docs.pytest.org/en/7.1.x/reference/reference.html#item.

[44] Pytest 2025. Item node ID. https://docs.pytest.org/en/7.1.x/reference/reference.html#pytest.nodes.Node.nodeid.

[45] Pytest 2025. Making your plugin installable by others. https://docs.pytest.org/en/stable/how-to/writing_plugins.html#making-your-plugin-installable-by-others.

[46] Pytest 2025. Parametrize. https://docs.pytest.org/en/stable/how-to/parametrize.html.

[47] Pytest 2025. Pytest. https://docs.pytest.org/en/stable/.

[48] Pytest 2025. pytest-dev. https://github.com/pytest-dev.

[49] Pytest 2025. Pytest plugin manager. https://docs.pytest.org/en/7.1.x/reference/reference.html#pytestpluginmanager.

[50] Pytest 2025. pytest_collection_modifyitems. https://docs.pytest.org/en/7.1.x/reference/reference.html#pytest.hookspec.pytest_collection_modifyitems.

[51] Pytest 2025. pytest_runtest_logreport. https://docs.pytest.org/en/7.1.x/reference/reference.html#pytest.hookspec.pytest_runtest_logreport.

[52] Pytest 2025. pytest_terminal_summary. https://docs.pytest.org/en/7.1.x/reference/reference.html#pytest.hookspec.pytest_terminal_summary.

[53] Pytest 2025. Rerun failed tests. https://docs.pytest.org/en/stable/how-to/cache.html#how-to-re-run-failed-tests-and-maintain-state-between-test-runs.

[54] Pytest 2025. TestReport. https://docs.pytest.org/en/7.1.x/reference/reference.html#pytest.TestReport.

[55] Pytest 2025. Writing hooks. https://docs.pytest.org/en/stable/how-to/writing_hook_functions.html#writinghooks.

[56] Python 2025. Entry points. https://packaging.python.org/en/latest/specifications/entry-points/.

[57] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test Case Prioritization: An Empirical Study. In *ICSM*.

[58] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing Test Cases for Regression Testing. *TSE* 27 (2001).

[59] David Saff and Michael D Ernst. 2003. Reducing Wasted Development Time via Continuous Testing. In *ISSRE*.

[60] 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 *ICSE*.

[61] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating Test-Suite Reduction in Real Software Evolution. In *ISSTA*.

[62] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A Framework for Automatically Fixing Order-Dependent Flaky Tests. In *FSE*.

[63] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In *ISSTA*.

[64] Per Erik Strandberg, Wasif Afzal, Thomas J Ostrand, Elaine J Weyuker, and Daniel Sundmark. 2017. Automated System Level Regression Test Prioritization in a Nutshell. *IEEE Software* 34 (2017).

[65] Surefire 2025. [SUREFIRE-2041] Ordering test classes and methods according to -Dtest property. https://github.com/apache/maven-surefire/pull/560.

[66] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online Defect Prediction for Imbalanced Data. In *ICSE*.

[67] UV 2025. UV: An extremely fast Python package and project manager. https://docs.astral.sh/uv/.

[68] Hao Wang, Pu Yi, Jeremias Parladorio, Wing Lam, Darko Marinov, and Tao Xie. 2024. Hierarchy-Aware Regression Test Prioritization. In *ISSRE*.

[69] Ruixin Wang, Yang Chen, and Wing Lam. 2022. iPFlakies: a framework for detecting and fixing python order-dependent flaky tests. In *ICSE Companion*.

[70] Ahmadreza Saboor Yaraghi, Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel C Briand. 2022. Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts. *TSE* 49 (2022).

[71] Pu Yi, Hao Wang, Tao Xie, Darko Marinov, and Wing Lam. 2022. A Theoretical Analysis of Random Regression Test Prioritization. In *TACAS*.

[72] Shin Yoo and Mark Harman. 2012. Regression Testing Minimisation, Selection and Prioritisation: A Survey. *STVR* 22 (2012).

[73] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *ISSTA*.

[74] Jianyi Zhou, Junjie Chen, and Dan Hao. 2021. Parallel Test Prioritization. *TOSEM* 31 (2021).