

Test Alert Snooze: An Empirical Study of Consecutive Test Failures on CI

Ayane Shirakawa 

Nara Institute of Science and Technology, Ikoma, Japan

Tatsuya Shirai 

Nara Institute of Science and Technology, Ikoma, Japan

Yutaro Kashiwa  

Nara Institute of Science and Technology, Ikoma, Japan

Masanari Kondo 

Kyushu University, Fukuoka, Japan

Yasutaka Kamei 

Kyushu University, Fukuoka, Japan

Hajimu Iida 

Nara Institute of Science and Technology, Ikoma, Japan

Abstract

Background. Continuous Integration (CI) aims to shorten release cycles by automating tests on every change. When the same test method keeps failing across consecutive revisions, new defects get buried among existing failures, dulling developer vigilance and weakening CI's benefit of rapid bug localization.

Aims. We define **Test Alert Snooze** as the state in which the same test method fails across two or more consecutive revisions, and provide its first empirical characterization: its prevalence, persistence in revisions and elapsed time, the commits developers make during **Test Alert Snooze**, and what they change to resolve it.

Method. We analyzed 27 open-source Python and Java projects on GitHub Actions with high test execution frequencies. From build histories and logs over a 90-day window (December 21, 2025 to March 20, 2026), we identified test methods that failed across two or more consecutive revisions, and classified both the commits made during **Test Alert Snooze** and those that resolved it.

Results. **Test Alert Snooze** accounts for 42.9% of observed test failures, with a median persistence of 2 consecutive revisions and about one day; extreme cases reached 11 commits and 16 days. During **Test Alert Snooze**, **fix** commits made up only 8.2% of developer activity, while **docs**, **test**, **refactor**, and **feat** collectively dominated. Resolutions were mostly single-category modifications to *Test* or *Product* files, and among resolving commits **test** was the most frequent (32.4%), well above **fix** (11.8%).

Conclusions. Test-failure persistence is not a single phenomenon but differs across the build, job, and test-method levels. At the test-method level, resolving a failure often takes the form of test-code maintenance rather than bug fixing, so predicting, detecting, and repairing **Test Alert Snooze** calls for techniques targeting test code alongside product code. This first characterization lays the groundwork for such techniques and for CI features that visualize test-method-level failure persistence.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging; Software and its engineering → Software development process management; Software and its engineering → Empirical software validation

Keywords and phrases Continuous integration (CI), Test failures, Empirical analysis

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.24

Category Technical Track Paper



© Ayane Shirakawa, Tatsuya Shirai, Yutaro Kashiwa, Masanari Kondo, Yasutaka Kamei, and Hajimu Iida;

licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 24; pp. 24:1–24:20



Leibniz International Proceedings in Informatics

LIPIcs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Funding We gratefully acknowledge the financial support of JSPS KAKENHI grants (JP24K02921, JP25K03100, JP26K23818, JP26H02500), as well as ASPIRE grant (JPMJAP2415), JST CREST grant (JPMJCR23M1, JPMJCR26X7), JST FOREST (JPMJFR252W), and the Inamori Research Institute for Science for supporting Yasutaka Kamei via the InaRIS Fellowship.

1 Introduction

Continuous Integration (CI) has become a widely adopted practice in software projects that follow modern development methodologies such as Agile and DevOps. In a CI environment, an automated build and test pipeline is triggered on a dedicated server every time a developer commits a change to the shared repository. By keeping the codebase in a constantly release-ready state, CI is expected to shorten the release cycle, that is, the time required to deliver new features to end users, and to provide rapid feedback on integration defects [8].

For CI to deliver these benefits, developers are expected to follow a fundamental discipline [8]: before pushing a commit to the shared repository, they should run the relevant tests in their local environment and submit the change only if those tests pass. This discipline is critical because a newly introduced defect can easily be masked by pre-existing test failures. Once the CI build is already broken, any regression caused by a later commit produces the same failing status as the original problem, and developers cannot distinguish a new defect from the old one without additional investigation. Consequently, the bug-introducing commit may go undetected for an extended period of time, weakening one of the most important benefits of CI, namely the prompt identification of faulty changes.

Although this discipline is repeatedly highlighted in both practitioner guides and academic studies [12, 27], prior empirical work has reported that test failures are surprisingly frequent in real-world repositories [3]. Our preliminary observation is consistent with this finding. As illustrated in Figure 1, several open-source projects exhibit long streaks of consecutive failing builds, during which the CI system reports test failures across many commits and, in some cases, across several days or weeks.

When test failures become routine, developers tend to treat the failing CI status as background noise rather than as an actionable alert. We refer to this state, in which the same test method fails across two or more consecutive revisions without timely remediation, as **Test Alert Snooze**. Under this condition, the CI pipeline continues to execute, yet its diagnostic value is severely degraded: failing builds no longer signal fresh problems, the responsibility for resolving the failure is diffused across the team, and the effort required to restore a green build grows together with the number of accumulated defects. While the broader topic of software test failures has attracted substantial attention in recent software engineering research [3, 22, 18], very few studies have focused specifically on consecutive failures and the developer behaviors that surround them.

This paper characterizes the phenomenon of **Test Alert Snooze** through an empirical analysis of open-source repositories. To structure this investigation, we formulate five research questions that address (i) how prevalent **Test Alert Snooze** is among CI test failures, (ii) how many consecutive revisions **Test Alert Snooze** persists for, (iii) how long **Test Alert Snooze** persists in elapsed time, (iv) what kinds of commits developers make during **Test Alert Snooze**, and (v) what they change to resolve **Test Alert Snooze**.

This paper makes the following contributions:

- **Definition.** We define **Test Alert Snooze** as the state in which the same test method fails across two or more consecutive revisions on CI, formalizing it as a behavioral proxy for test-method-level failure persistence that is distinct from the build-level and job-level persistence examined in prior work.

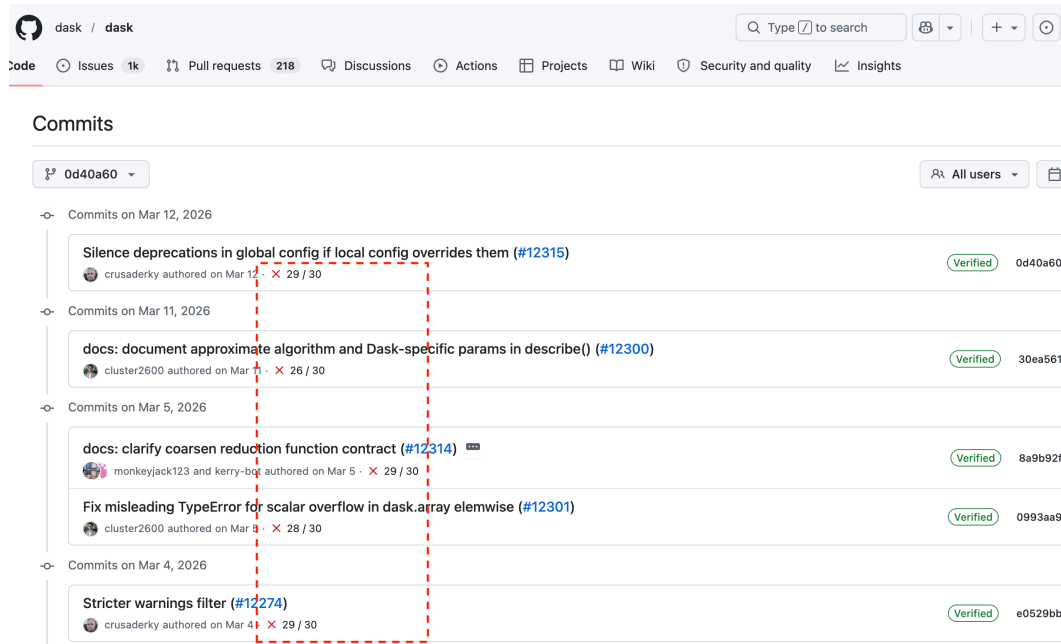


Figure 1 Examples of consecutive test failures in a project (URL: <https://github.com/dask/dask/commits/0d40a604284e021d098682b71c55d0387a414925>).

- **Identification methodology.** We propose a procedure for identifying **Test Alert Snooze** occurrences from CI logs that handles three sources of noise: inconclusive revisions such as skipped or cancelled tests, workflow and job configuration drift across commits, and renamed test methods detected via RefactoringMiner.
- **Empirical characterization.** Our analysis of 27 open-source Python and Java projects on GitHub Actions over a 90-day window reveals that **Test Alert Snooze** accounts for 42.9% of observed test failures, with a median persistence of 2 consecutive revisions and approximately one day, and that resolving commits are dominated by **test** (32.4%) rather than **fix** (11.8%), indicating that test-code maintenance plays a central role in resolution.
- **Replication package.** We publicly release our dataset, which includes test failure logs, RefactoringMiner detection results, per-revision test details, and the scripts used for each research question, to support replication and follow-up studies.

Paper Structure. The remainder of this paper is organized as follows. Section 2 reviews related work on CI practices and software test failures. Section 3 describes our dataset and analysis methodology. Section 4 presents the results for each research question. Section 5 discusses the implications of our findings. Section 6 discusses threats to validity. Section 7 concludes the paper and outlines directions for future work.

2 Background

This section provides a motivating example of consecutive test failures observed in an open-source project, and reviews prior empirical work on CI failures and test failures.

2.1 Motivating Example

Figure 1 shows a sequence of commits in the `dask` project, a widely used Python library for parallel computing. GitHub reports the outcome of CI jobs for each commit through a status indicator adjacent to it. When the count falls short of the total, at least one of the configured jobs failed for that revision. As Figure 1 illustrates, several consecutive commits in this project carry such failure marks, indicating that the CI signal stayed in a non-passing state across multiple revisions.

The persistence of failures at the build and job level has been documented in prior empirical work. Jin and Servant [13] reported that, of all failed builds, a median of 52% were preceded by another failed build, a state they term *Subsequent Build Failure*. Gallaba *et al.* [10] further showed that at least 44% of failing builds are cases in which only a subset of the jobs within the build actually failed while the others passed. At the granularity of individual tests, however, the persistence of failures across revisions has not been empirically investigated. We refer to such a state, in which the same test method continues to fail across consecutive revisions, as a “*Test Alert Snooze*”, which we formalize in Section 3.1.

A failure notification from automated tests on CI is one of the primary channels through which developers learn of a defect in their code changes. Upon receiving such a notification, developers first identify which tests have failed before opening an investigation. What happens, then, when a test that has already been investigated continues to fail across consecutive revisions? In broader cybersecurity and operational domains, this condition is known to induce *alert fatigue*: a phenomenon in which repeated exposure to high volumes of identical or false-positive alerts gradually desensitizes the recipient, to the point that even critical threats may be overlooked [21]. The same pattern has been documented in software engineering: developers exposed to repeated failures from flaky tests have been shown to give up on identifying the root cause and instead settle for re-running the tests [19, 9, 11].

The consequences extend beyond developer awareness. As a foundational discipline of CI, broken builds, including those caused by test failures, must be fixed immediately [8]. Yet when the same test continues to fail across consecutive revisions, a newly introduced defect surfacing in a similar setting can be dismissed as “the usual failure”, leaving the bug unfixed. Therefore, independent of general process factors such as time pressure and prioritization, this study analyzes the occurrence patterns, resolution times, and fix strategies of Test Alert Snooze. We believe that understanding this phenomenon will help developers identify candidate locations that require fixing, thereby contributing to preventing bugs from being overlooked and left unaddressed.

2.2 Related Work

CI failures are an everyday phenomenon in modern software development, and they have been studied extensively in prior empirical work. Kerzazi *et al.* [15] reported a build failure rate of 17.9% in an industrial CI environment, with each build failure requiring an average of 57 minutes to repair. In open-source contexts, Beller *et al.* [3] similarly showed that 10–20% of CI builds on Travis CI contain test failures, confirming that CI failures are routine rather than exceptional events.

Among the sources of these CI failures, automated test failures have consistently been identified as the dominant origin. Rausch *et al.* [20] analyzed Java open-source CI workflows and found that test failures account for over 80% of build failures in some projects. Vassallo *et al.* [24] contrasted open-source and industrial CI failure profiles and observed that unit test failures alone account for 28% of open-source build failures. More recently, Zheng *et*

al. [28] analyzed GitHub Actions workflow failures and reported that test failures constitute 32% of all workflow failures, the single largest category. These findings motivate a focus on test failures as the primary CI failure category.

The underlying causes of test failures have also been studied, with flaky tests in particular receiving the most concentrated research attention. Luo *et al.* [18] conducted a foundational analysis identifying asynchronous waits, concurrency, and test order dependency as the dominant root causes of flaky tests. Subsequent studies have documented how developers experience this phenomenon. Eck *et al.* [9] combined a classification analysis of 200 flaky test instances fixed by 21 professional developers with an online survey of 121 developers (median industry experience of 5 years). They found that 79% of respondents regard flaky tests as a moderate-to-severe problem, and that only 25% of time-dependency flaky tests were repaired rather than disabled.

3 Study Design

This section defines **Test Alert Snooze** as a behavioral proxy for consecutive test-method failures, details the procedure for identifying its occurrences from CI logs, and describes the dataset used in our analysis.

3.1 Definition of Test Alert Snooze

Figure 2 illustrates five test methods (Test A–E) observed across four consecutive revisions r_i , r_{i+1} , r_{i+2} , and r_{i+3} . Test A fails at both r_i and r_{i+1} ; this constitutes a **Test Alert Snooze** because the same test method fails across two consecutive revisions. Test B, which fails only at r_{i+1} and passes at r_{i+2} , does not qualify because the failure does not persist.

A revision is counted toward consecutiveness only when its test result is conclusively determined, *i.e.*, when the test either passes or fails. Revisions whose test results are not conclusively determined, which we refer to as *inconclusive revisions*, are excluded from the count. An inconclusive revision falls into one of three cases: *Test Skipped* (the CI system skips a test without executing it), *Test Cancelled* (execution is interrupted before completion), or *No Test* (no test result is available at all). Thus, if a test method fails at revision r_i and fails again at the next conclusive revision, the state qualifies as a **Test Alert Snooze** regardless of any inconclusive revisions between them.

When a failure persists across two or more consecutive conclusive revisions, we treat the entire span as a single **Test Alert Snooze** rather than as multiple occurrences. We define its resolution as the moment when the failing result transitions to a passing one.

3.2 Test Alert Snooze Identification

We identify **Test Alert Snooze** occurrences from CI logs using the following procedure.

1. Identify the workflow, job, and step that run the tests. For each project, we target the workflows defined as YAML files (`.yaml`) under the `.github/workflows` directory on the default branch, and consider only those that execute tests. We exclude workflows used for purposes other than testing, such as code review or dependency management [6], as well as workflows that do not explicitly list individual test methods upon failure.

Using the GitHub API, we retrieve the workflow execution history of the past 90 days ending at the initial latest commit, which corresponds to the maximum log retention period of GitHub Actions [7]. We then refer to the latest revision with a successful workflow execution and manually identify the workflow, job, and step names associated with testing.

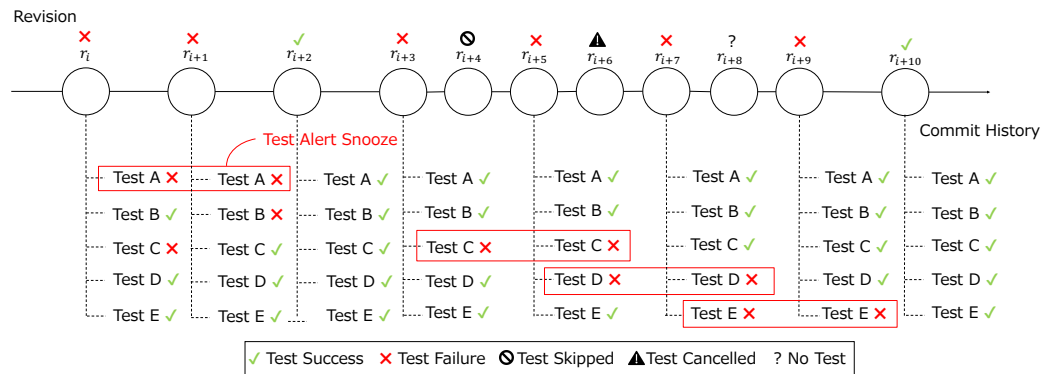


Figure 2 Definition of Test Alert Snooze.

When multiple jobs run tests within the workflow, we focus on the job with the highest test execution frequency in each project. When multiple jobs share the maximum count, we prioritize those meeting three criteria: a Linux OS, `allow_failure: false`, and the most recent execution environment. If the selection is still ambiguous, unit tests are prioritized over other types. For package tests, full-featured environments are selected, while the most fundamental modules are chosen for module-level tests. Any further ties are resolved by selecting the job with the higher number of test cases. If multiple test steps exist within the selected job, we apply the same criteria but prioritize steps other than doctests and smoke tests, including the latter only if no other test types are available.

If all executions within the 90-day window failed, we identify the most recent successful revision for each individual job to assess the testing process. If every job also failed, we evaluate the latest available revision to determine whether tests were being executed.

2. Retrieve and parse job logs. To preserve build logs beyond GitHub’s 90-day retention limit, we built a system that programmatically collects and stores logs in a local environment, and runs a monthly automated retrieval process.

Because the GitHub Actions API does not provide a way to retrieve logs at the step level, we target jobs in which the test step identified in Step 1 failed, and retrieve the entire job log. We then parse the portion of the retrieved log that corresponds to the test step.

Figure 3 shows an example of such a log. We extract lines that begin with the fixed marker “FAILED” followed by a fully qualified test method name (*e.g.*, the red lines in Figure 3) and treat the listed test methods as failing in that revision. All other lines are ignored.

3. Identify Test Alert Snooze occurrences. In this step, we use the information obtained above to scan the commit history and identify the spans of Test Alert Snooze occurrences. As a preliminary step, we assign one of three states to each (revision, test method) pair:

- **Pass:** none of the above applies.
- **Fail:** the corresponding test step failed and the test method was extracted under a “FAILED” marker during the parsing in Step 2.
- **Inconclusive:** the test result is not conclusively determined. This covers the three cases defined in Section 3.1: *Test Skipped*, where the GitHub API reports the step status as `skipped`; *Test Cancelled*, where the step status is `cancelled` (*i.e.*, execution was

```

37092 XPASS dask/dataframe/tests/test_groupby.py::test_dataframe_aggregations_multilevel[std-disk-1-<lambda>3] - numeric_only=False not implemented
37093 XPASS dask/dataframe/tests/test_groupby.py::test_dataframe_aggregations_multilevel[std-disk-1-<lambda>4] - numeric_only=False not implemented
37094 XPASS dask/dataframe/tests/test_groupby.py::test_dataframe_aggregations_multilevel[std-disk-2-<lambda>0] - numeric_only=False not implemented
37095 XPASS dask/dataframe/tests/test_groupby.py::test_groupby_reduction_split[mean-disk-split_every] - numeric_only=False not implemented
37096 XPASS dask/dataframe/tests/test_groupby.py::test_groupby_reduction_split[mean-disk-split_out] - numeric_only=False not implemented
37097 XPASS dask/dataframe/tests/test_groupby.py::test_groupby_reduction_split[mean-tasks-split_every] - numeric_only=False not implemented
37098 XPASS dask/tests/test_dot.py::test_dot_graph[jpeg-Image] - jpeg not always supported in dot
37099 XPASS dask/tests/test_dot.py::test_dot_graph_no_filename[jpeg-Image] - jpeg not always supported in dot
37100 XPASS dask/tests/test_order.py::test_string_ordering_dependents - see comment
37101 FAILED dask/array/tests/test_array_core.py::test_tiledb_roundtrip - DeprecationWarning: Setting the shape on a NumPy array has been deprecated in NumPy 2.5.
37102 As an alternative, you can create a new view using np.reshape (with copy=False if needed).
37103 FAILED dask/array/tests/test_array_core.py::test_tiledb_multiattr - DeprecationWarning: Setting the shape on a NumPy array has been deprecated in NumPy 2.5.
37104 As an alternative, you can create a new view using np.reshape (with copy=False if needed).
37105 = 2 failed, 16377 passed, 1037 skipped, 336 xfailed, 282 xpassed, 7 warnings in 1360.30s (0:22:40) =
37106 Error: Process completed with exit code 1.

```

■ **Figure 3** For example of build logs for failed tests (URL: <https://github.com/dask/dask/actions/runs/23369249959/job/67989498814>).

interrupted before completion); and *No Test*, where the target test step itself does not exist in this revision, which can occur when the workflow or job configuration changes across commits and the step identified in Step 1 is temporarily absent.

We then reconstruct the parent–child relationships of all commits within the period specified in Step 1 on the default branch. Starting from a parent revision and proceeding through its successive children, we examine the state sequence of each test method. When merge commits occur, the most recent child commit is prioritized.

During this scan, each run of consecutive **Fail** states is recorded as one **Test Alert Snooze** occurrence. **Inconclusive** states are treated as state-preserving: they do not affect the consecutiveness, and the next conclusive revision determines whether the run continues.

To prevent renamed test methods from being incorrectly merged into a single **Test Alert Snooze**, we use RefactoringMiner [2] to detect Rename refactorings (at the package, file, class, and method levels) between adjacent revisions.¹ If a Rename is detected within a candidate **Test Alert Snooze** span, the span is split at that point, because the renamed test method is no longer considered the same test. Rename refactorings detected within **Inconclusive** intervals are aggregated and reflected in the final determination. Test method identity is verified by the triple of file path, class name, and method name.

3.3 Dataset

In this study, we selected 27 projects by filtering the 20 Java and Python projects from Bouzenia and Pradel [4] and 14 projects from Cheng *et al.* [5] to include only those with revisions created on the default branch between December 21, 2025, and March 20, 2026, that underwent at least one test execution. The specific details of these 27 projects are presented in Table 1, and they serve as the basis for investigating our Research Questions.

Furthermore, due to the constraints of using RefactoringMiner [2], this study is limited to projects where the primary programming language is either Java or Python.

¹ <https://github.com/tsantalis/RefactoringMiner>

■ **Table 1** Studied projects (As of May 9, 2026), Build data (90-day period from December 21, 2025, to March 20, 2026).

Project	Main Language	Commits	Stars	Contributors	Total Build	Build Top Jobs
Activiti	Java	11,434	10,524	228	35	35
aeon	Python	5,170	1,384	323	1,868	86
ansible-lint	Python	3,226	3,863	334	830	83
apscheduler	Python	1,324	7,473	59	30	3
commons-csv	Java	3,110	408	53	286	26
cpython	Python	131,326	72,624	355	18,398	740
dask	Python	9,763	13,826	417	1,132	54
django	Python	34,570	87,451	393	1324	166
dubbo	Java	8,844	41,525	399	5,095	141
flink	Java	37,849	25,988	283	3,754	21
guava	Java	7,340	51,491	328	953	106
ipython	Python	28,397	16,700	413	432	34
keras	Python	12,371	64,057	408	2,371	287
langchain	Python	15,906	136,251	469	6,250	429
molecule	Python	3,157	4,095	307	225	25
networkx	Python	8,294	16,889	438	1,182	70
numpy	Python	41,183	31,996	437	398	73
pandas	Python	37,956	48,699	414	27,422	617
pytest	Python	17,327	13,836	379	1,977	75
pytest-django	Python	1,364	1,529	146	240	12
pytest-xdist	Python	1,378	1,841	101	240	12
pytorch-lightning	Python	11,075	31,128	436	4,785	42
rocketmq	Java	9,082	22,430	336	142	71
scikit-learn	Python	33,629	66,016	410	2,213	235
scipy	Python	37,483	14,678	412	1,201	25
trimesh	Python	5,859	3,565	212	110	5
ultralytics	Python	3,940	56,926	354	3,366	341

4 Research Questions

To characterize **Test Alert Snooze** empirically, we formulate five research questions that examine its prevalence, persistence, and the developer activity surrounding it. RQ1 quantifies how prevalent **Test Alert Snooze** is among CI test failures. RQ2 and RQ3 measure how long a **Test Alert Snooze** persists, in terms of consecutive failing revisions and elapsed time respectively. RQ4 examines what kinds of commits developers make while a **Test Alert Snooze** is ongoing, and RQ5 analyzes what they change to resolve it.

RQ1: How Prevalent Is Test Alert Snooze?

Motivation. Beller *et al.* [3] reported that approximately 10–20% of CI builds on Travis CI contain test failures, indicating that test failures are a routine phenomenon in CI practice. However, the proportion of these failures that persist across consecutive revisions, *i.e.*, those that qualify as **Test Alert Snooze**, has not been quantified. If **Test Alert Snooze** is rare, most failures are resolved promptly and CI functions as intended. If it is common, persistent failures represent a non-trivial challenge that warrants systematic investigation. Therefore, in RQ1, we quantify the prevalence of **Test Alert Snooze** across projects. Specifically, by quantifying the proportion of test failures that transition into **Test Alert Snooze**, we highlight the severity of these failures and the delays in addressing them.

Approach. For each project, we use the job selected in Section 3.1 and quantify the prevalence of **Test Alert Snooze** along two dimensions. First, we report the fraction of projects with at least one **Test Alert Snooze** during the observation window. Second,

■ **Table 2** Test Alert Snooze prevalence per project (only projects with ≥ 1 failure).

Project	# All Test Failures	# Isolated Test Failures	# Test Alert Snooze (Occurrences)	# Test Failures during Test Alert Snooze
aeon	17	1	5	16(94.1%)
dask	34	5	5	29(85.3%)
flink	143	83	24	60(42.0%)
rocketmq	11	7	2	4(36.4%)
ansible-lint	1	1	0	0(0.0%)
guava	1	1	0	0(0.0%)
molecule	5	5	0	0(0.0%)
pandas	35	35	0	0(0.0%)
scikit-learn	2	2	0	0(0.0%)
ultralytics	5	5	0	0(0.0%)
Total	254	145	36	109(42.9%)

*All Test Failures = Isolated Test Failures + Test Failures during Test Alert Snooze

treating each isolated failure and each **Test Alert Snooze** occurrence (regardless of length) as one failure event, we compute the fraction of failure events that were **Test Alert Snooze**, per project and overall.

Results. Table 2 summarizes the prevalence of **Test Alert Snooze** across the 27 projects analyzed. 10 projects (approximately 37.0%) had at least one test failure during the observation window, and **Test Alert Snooze** was observed in 4 of these 10 projects (40.0%). This indicates that **Test Alert Snooze** was observed in 4 out of all 27 projects (approximately 15.0%). Overall, **Test Alert Snooze** accounted for 42.9% of all test failures across the studied projects. Furthermore, every one of the 4 projects in which **Test Alert Snooze** was observed exhibited a per-project **Test Alert Snooze** share exceeding 36%.

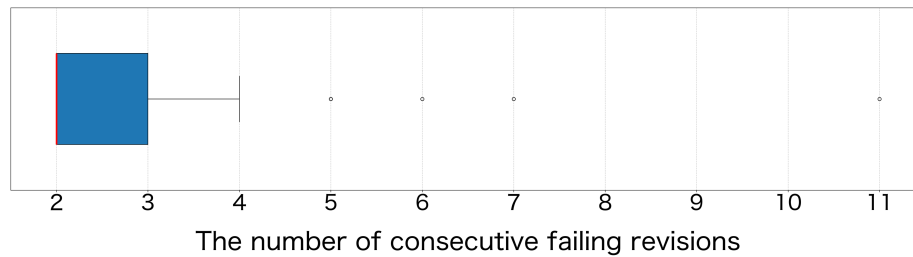
Answer to RQ1

Test Alert Snooze occurred in 4 out of all 27 projects (approximately 15.0%), and accounted for 42.9% of all observed failure events.

RQ2: How Many Revisions Does Test Alert Snooze Continue?

Motivation. Jin and Servant [13] reported that 52% of build failures occur in builds where the previous build also failed, and Gallaba *et al.* [10] showed that 83% of builds following a branch-point failure also fail immediately. These findings suggest that consecutive build failures are common and that immediate resolution is often difficult. However, build-level analysis cannot capture how many consecutive revisions the same test method continues to fail once a **Test Alert Snooze** begins. If **Test Alert Snooze** persists for many revisions, developers are repeatedly exposed to the same failing test method. Tariq *et al.* [21] showed that such repeated exposure causes operators to dismiss even genuine alerts, and Levén *et al.* [17] demonstrated that exposure to such degraded states lowers developers' coding standards, increasing the risk of introducing further defects. RQ2 measures the number of consecutive revisions for which each **Test Alert Snooze** persists, clarifying how many times developers end up observing the same failing test method.

Approach. For each **Test Alert Snooze** occurrence, we count the number of consecutive failing revisions of the same test method. As described in Section 3.1, inconclusive revisions (*e.g.*, **Test Skipped** or **Test Cancelled**) preserve consecutiveness but are not counted toward the run length. We report the distribution (median and maximum) over all occurrences. As a secondary observation, we also count, for each **Test Alert Snooze**-experienced method, how many **Test Alert Snooze** occurrences it had, and report whether any method experienced **Test Alert Snooze** more than once (recurrence) on a per-project basis.



■ **Figure 4** Distribution of consecutive failing revisions per **Test Alert Snooze** occurrence.

Results. Figure 4 shows the distribution of consecutive failing revisions per **Test Alert Snooze** occurrence. The number of consecutive failing revisions per **Test Alert Snooze** occurrence had a median of 2 and a maximum of 11. Notably, 61.1% of occurrences ended at 2 consecutive revisions, indicating that most cases are resolved quickly. However, the long upper tail (with outliers at 5, 6, 7, and 11 revisions) shows that some **Test Alert Snooze** occurrences persist for multiple revisions before the failing test is fixed, a behavior inconsistent with the principle of immediate fix in CI [8].

As a secondary observation, we examined whether the same test method experiences **Test Alert Snooze** more than once during the observation period. In 3 of the 4 projects with **Test Alert Snooze** (aeon, dask, and rocketmq), every **Test Alert Snooze**-experienced method hit a **Test Alert Snooze** only once. flink was the exception: all 3 of its **Test Alert Snooze**-experienced methods recurred, giving a 100% recurrence rate and showing substantial inter-project variability.

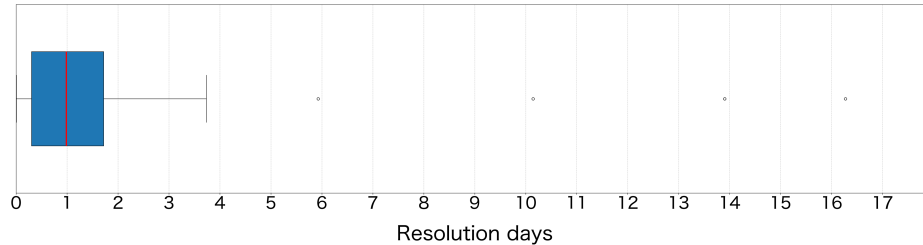
Answer to RQ2

Test Alert Snooze occurrences last for a median of 2 consecutive revisions, with 61.1% falling at this length, while the maximum reaches 11 consecutive revisions.

RQ3: How Long Does It Take to Resolve Test Alert Snooze?

Motivation. While RQ2 measures the persistence of **Test Alert Snooze** in terms of revision count, revision count alone does not capture how long a failure remains visible to developers in real time. Gallaba *et al.* [10] reported that build breakages persist for seven days on average, indicating that elapsed time is an independent dimension of failure persistence. A **Test Alert Snooze** spanning five revisions may take only a few hours in one case and several weeks in another, with substantially different implications for developers. The longer a **Test Alert Snooze** persists in elapsed time, the greater the risk of developer desensitization and missed defects. RQ3 measures the elapsed time of each **Test Alert Snooze** to clarify its temporal impact on developers.

Approach. For each **Test Alert Snooze** occurrence, we compute the elapsed time as the difference between the commit timestamp of the first failing revision and that of the revision at resolution (*i.e.*, when the test returned to passing), as defined in Section 3.1. We report the distribution (median and maximum) over the 34 occurrences resolved within the observation window (the remaining 2 are right-censored).



■ **Figure 5** Distribution of resolution days per **Test Alert Snooze** occurrence.

Results. Figure 5 shows the distribution of resolution days per **Test Alert Snooze** occurrence. The elapsed time per **Test Alert Snooze** occurrence had a median of 23h 38min and a maximum of 16 days. 52.9% of occurrences were resolved within one day, indicating that most cases are addressed quickly. On the other hand, cases persisting for up to 16 days were also observed. Such prolonged exposure increases the risk of developer desensitization and missed defects.

Placed alongside the consecutive failing revisions observed in RQ2 (median of 2, maximum of 11), the typical persistence of a **Test Alert Snooze** occurrence is on the scale of “two revisions and about one day”. Although this comparison is limited to medians, the fact that the median revision count is as low as 2 while the median elapsed time spans approximately one day suggests that revision count and elapsed time may not always align.

Answer to RQ3

Test Alert Snooze occurrences are resolved in a median of approximately 24 hours (23h 38min), with the maximum reaching 16 days. While the median revision count of 2 and the median elapsed time of about one day both indicate that typical persistence is short, the two dimensions do not always coincide at the level of individual occurrences.

RQ4: What Commits Do Developers Make During Test Alert Snooze?

Motivation. When a CI test failure first occurs, Parry *et al.* [19] and Vassallo *et al.* [23] characterized typical developer responses such as rerunning the build, inspecting logs, and attempting local reproduction. At industrial scale, Aïdasso *et al.* [1] further showed that silent failures, in which build jobs are marked as successful despite failing to complete their tasks, are routinely dismissed by reruns rather than by code fixes, producing an illusion of success. Although **Test Alert Snooze** differs from silent failures in that its failure is explicit, both share the property of repeated failures that developers may dismiss without addressing the root cause. However, how developers respond when a test failure persists across many revisions has not been studied. They may apply the same rerun-driven dismissal that Aïdasso *et al.* observed, ignore the failure under alert fatigue, or be unaware that the same test continues to fail. RQ4 investigates what kinds of commits developers make during a **Test Alert Snooze**.

■ **Table 3** CCS categories (%) during **Test Alert Snooze** ($n = 73$).

Category	Description	Ratio
feat	Addition of new features	19.2%
build	Changes related to the build system	4.1%
style	Code style changes	0.0%
chore	Miscellaneous tasks or other changes	6.8%
perf	Performance improvements	4.1%
ci	Changes related to CI	6.8%
test	Addition or improvement of tests	15.1%
refactor	Code refactoring	11.0%
docs	Documentation-only changes	24.7%
fix	Bug fixes	8.2%

Approach. For each **Test Alert Snooze** occurrence, we examine the commits made between the second failing revision and the last failing revision. We exclude the first failing revision because developers may not yet recognize the failure as a recurring issue at that point; the second failing revision onward marks when the **Test Alert Snooze** pattern becomes observable. We classify each commit using the fine-tuned large language model released by Zeng *et al.* [25], which is designed specifically for assigning CCS (Conventional Commits Specification) types to commit content. CCS is a widely adopted convention that categorizes commits by their intent into ten types such as **feat**, **fix**, **docs**, **refactor**, and **test**. We report the distribution of CCS categories across all classified commits.

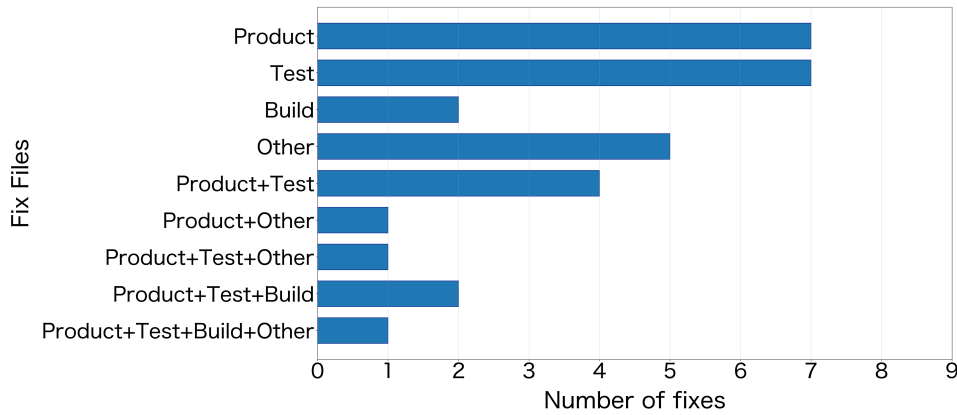
Results. We analyzed 73 commits drawn from snooze periods across all studied projects. Table 3 (column “RQ4: During”) summarizes the distribution of CCS categories during **Test Alert Snooze**. The most frequent category was **docs** (24.7%), followed by **test** (15.1%), **refactor** (11.0%), and **feat** (19.2%). Notably, commits explicitly aimed at fixing the failure (**fix**) accounted for only 8.2% of the total.

This pattern differs from the typical CCS distribution reported by Zeng *et al.* [25], in which **fix** is the most frequent category across 88,704 conventional commits from 116 open-source projects. The relatively low proportion of **fix** during **Test Alert Snooze** periods suggests that fix-driven engagement was not the dominant activity.

Furthermore, the distribution reveals two notable patterns. First, categories with no apparent connection to the failing test – **docs** (24.7%), **feat** (19.2%), and **chore** (6.8%) – together account for 50.7%, suggesting that developers continue routine development activities while the **Test Alert Snooze** persists. Second, categories that could potentially relate to the failing test – **test** (15.1%) and **refactor** (11.0%) – together account for 26.1%.

Answer to RQ4

During **Test Alert Snooze**, explicit **fix** commits accounted for only 8.2% of developer activity, while **docs** (24.7%), **test** (15.1%), **refactor** (11.0%), and **feat** (19.2%) were more frequent. Commits whose connection to the failing test is not immediately apparent collectively dominated, suggesting that fix-driven engagement was not the primary activity while the failure persisted.



■ **Figure 6** Ratio of Fix Locations for `Test Alert Snooze` by all projects.

RQ5: What Do Developers Change to Resolve `Test Alert Snooze`?

Motivation. Vassallo *et al.* [23] showed that resolving CI test failures is challenging, often requiring local reproduction and deep understanding of the changes involved. The structural scope of such fixes has been studied for general bugs by Zhong and Su [29], who found that real-world fixes frequently span multiple files, and for test failures by Vahabzadeh *et al.* [22], who showed that the buggy artifact is sometimes the test code itself rather than the code under test. However, whether fixes for `Test Alert Snooze` are confined to a single artifact or span multiple file categories has not been investigated. RQ5 examines what developers change when resolving `Test Alert Snooze`, by analyzing both the categories of files modified and the intent of the resolving commits.

Approach. RQ5 is computed over the 34 resolved occurrences. For each `Test Alert Snooze` occurrence, we analyze the resolution from two perspectives: the categories of modified files and the classification of resolving commits.

For the first perspective, we extract the files modified between the second failing revision and the final commit (*i.e.*, where the test returned to a passing state) using `git diff`, and classify each modified file into one of four categories: *Test* (`.py`, `.java`, `.scala`, or `.ts` files whose path or name contains “test”), *Product* (the remaining `.py`, `.java`, `.scala`, or `.ts` files), *Build* (build-related files such as `pyproject.toml`, `pom.xml`, `build.gradle`, `.yml`, and `.lock`), and *Other* (*e.g.*, `README`). We report the frequency of each combination of modified file categories across all occurrences. For the second perspective, we classify the resolving commit using the same LLM as in RQ4 [25], and report the distribution of CCS categories.

Results. Figure 6 shows the classification of modified files in the `Test Alert Snooze`-resolved revisions across the 4 projects in which `Test Alert Snooze` occurred. Overall, single-category modifications were predominant, accounting for 70.0% of `Test Alert Snooze` resolutions, with *Product* and *Test* being the most frequent and equally represented categories.

As for the resolving commit classification, Table 4 (column “RQ5: Resolving”) summarizes the distribution of CCS categories assigned to the 34 resolving commits. The most frequent category was `test` (32.4%), followed by `docs` (8.8%) and `refactor` (8.8%). Notably, `test` alone accounts for nearly one third of resolving commits, while explicit `fix` commits accounted

■ **Table 4** CCS categories (%) during **Test Alert Snooze** ($n = 73$) and resolving ($n = 34$).

Category	Description	RQ4: During	RQ5: Resolving	Diff
feat	Addition of new features	19.2%	8.8%	-10.4
build	Changes related to the build system	4.1%	5.9%	+1.8
style	Code style changes	0.0%	5.9%	+5.9
chore	Miscellaneous tasks or other changes	6.8%	2.9%	-3.9
perf	Performance improvements	4.1%	5.9%	+1.8
ci	Changes related to CI	6.8%	8.8%	+2.0
test	Addition or improvement of tests	15.1%	32.4%	+17.3
refactor	Code refactoring	11.0%	8.8%	-2.2
docs	Documentation-only changes	24.7%	8.8%	-15.9
fix	Bug fixes	8.2%	11.8%	+3.6

for only 11.8%. Compared to the activities during **Test Alert Snooze** (Table 4, column “RQ4: During”), **test** replaces **docs** as the most frequent category, rising from 15.1% to 32.4% (a difference of +17.3), while **docs** drops from 24.7% to 8.8%.

Answer to RQ5

Test Alert Snooze resolutions are dominated by single-category modifications, most frequently to *Product* or *Test* files. Among resolving commits, the most frequent CCS category was **test** (32.4%) rather than **fix** (11.8%), indicating that resolving **Test Alert Snooze** often involves modifications to the test code itself rather than to product code alone.

5 Implications

In this section, we discuss the implications of our findings for researchers, practitioners, and CI tool providers.

5.1 Implications for Researchers

The implications for researchers concern perspectives for positioning **Test Alert Snooze** as a new research target and directions for future research.

Researchers should measure the persistence of Test Alert Snooze at the test-method level rather than inferring it from build- or job-level results. Jin and Servant [13] reported a 52% consecutive failure rate at the build level, and Gallaba *et al.* [10] showed that 83% of builds following a branch-point failure also fail immediately. At the test-method level, we observed **Test Alert Snooze** at a rate of 42.9% (RQ1). These figures are not directly comparable: they are defined over different granularities and different events, and our measurement covers only a small number of tests across 4 projects. What they do indicate is that the relationship between persistence at the build or job level and persistence at the level of individual test methods has not yet been established. Researchers should therefore measure persistence at the build, job, and test-method levels separately to determine at which level alert fatigue actually arises, and validate our test-method-level results on a substantially larger set of projects and tests.

Researchers should develop new prediction and repair approaches that target test code.

The results of RQ5 showed that 32.4% of resolving commits fell into the `test` category, while explicit `fix` commits accounted for only 11.8%. This suggests that the resolution of `Test Alert Snooze` is not a simple bug-fixing task but rather involves modifications to the test code itself. Existing Just-in-time defect prediction techniques [14], which predict whether a commit introduces a defect at the time of the commit, predominantly target product code. Consequently, applying these techniques directly to `Test Alert Snooze` is insufficient. Researchers should develop new prediction models and automated repair techniques that jointly target both test code and product code.

Researchers should clarify the boundary between `Test Alert Snooze` and flaky tests. In this study, we defined `Test Alert Snooze` solely based on temporal persistence (Section 3.1); however, it remains unclear what fraction of the observed `Test Alert Snooze` occurrences stems from genuine defects versus flaky tests (tests whose outcomes are non-deterministic). Researchers should leverage rerun-based flaky test detection techniques (*e.g.*, [16]) together with our persistence-based definition to separate “`Test Alert Snooze` caused by genuine defects” from “`Test Alert Snooze` caused by flaky tests” and to compare how each affects practitioners’ cognitive load.

Researchers should establish techniques for distinguishing individual defects within a `Test Alert Snooze`. Even when the same test method fails consecutively during a `Test Alert Snooze`, there is no guarantee that the underlying defect remains the same. If a newly introduced defect produces the same failure status (*e.g.*, `AssertionError`) as the existing failing test, practitioners may have difficulty distinguishing between them, potentially delaying the detection of the new defect. The results of RQ5 in this study showed that a non-trivial fraction of `Test Alert Snooze` resolutions involved modifications spanning multiple file categories, which also suggests that multiple distinct defects may have coexisted during the `Snooze` period. Prior work such as BuildSheriff [26] has proposed techniques for clustering test failures within a single build according to their root causes; however, what is needed in the context of `Test Alert Snooze` is a technique that identifies individual defects along the temporal axis within a sequence of consecutive failures spanning multiple builds. Researchers should investigate techniques that use failure logs, stack traces, and error messages to distinguish individual defects within consecutive failures.

5.2 Implications for Practitioners

The implications for practitioners concern suggestions for how to respond when encountering a `Test Alert Snooze`.

Practitioners should recognize that resolving a `Test Alert Snooze` does not always conclude with modifications confined to a single artifact category. The results of RQ5 showed that while single-category modifications were the most common pattern, a non-trivial fraction of `Test Alert Snooze` resolutions involved modifications spanning multiple categories among *Product*, *Test*, *Build*, and *Other*. Practitioners should remain open to the possibility that resolving a failing test may require modifications outside the immediately suspected artifact (*e.g.*, outside the test code itself), particularly when an initial fix does not restore the build to a passing state.

Practitioners should treat the test itself as a primary candidate for modification, not only the product code under test. RQ5 showed that, among single-category resolutions, *Test* and *Product* files were modified with equal frequency, and that the most common CCS category among resolving commits was `test` (32.4%), well above `fix` (11.8%). This indicates that a persistently failing test is often resolved by updating the test itself (*e.g.*, to reflect intentional changes in product behavior, to relax brittle assertions, or to remove ordering dependencies) rather than by debugging the product code. Practitioners should therefore include the failing test in their early investigation rather than defaulting to product-code debugging, which may shorten the resolution time and reduce the team’s exposure to the failing alert.

5.3 Implications for CI Tool Providers

The implications for CI tool providers concern functionalities that CI platforms could provide to address `Test Alert Snooze`.

CI tool providers should offer visualization of consecutive failures at the test method level.

Most current CI platforms, including GitHub Actions, visualize pass/fail status at the build level but do not provide standard functionality for tracking how many times an individual test method has failed consecutively. As suggested by the results of RQ1, persistence at the test method level is a different indicator from persistence at the build level (42.9% vs. 52%, though the two metrics are defined at different granularities) and carries information that build-level visualization alone cannot capture. By presenting this information on the dashboard, CI tool providers can enable practitioners to intervene before alert fatigue sets in.

CI tool providers should offer mechanisms that present individual defects within consecutive failures as distinguishable signals to practitioners.

When the same test fails consecutively during a `Test Alert Snooze`, a newly introduced defect may become buried under the existing failing test as it carries the same status, creating the risk that practitioners will overlook its detection. This significantly undermines the *rapid bug localization* capability that CI is intended to provide. As discussed in the Implications for Researchers, techniques for distinguishing individual defects within consecutive failures remain an open research challenge; building on such future advances, CI tool providers should consider design choices that provide notifications and visualizations that distinguish newly introduced defects from existing failing tests. Such mechanisms would help practitioners maintain attention to new defects even during a `Test Alert Snooze`.

6 Threats to Validity

6.1 Internal Validity

We retrieved 90 days of build logs from GitHub Actions, which corresponds to the upper limit of its log retention period. `Test Alert Snooze` occurrences that are truncated at either end of our observation window, or that persist beyond 90 days, cannot be observed; therefore, the maximum values reported in this study (11 commits and 16 days) should be interpreted as conservative estimates. Nevertheless, 61.1% of the observed `Test Alert Snooze` occurrences were resolved within 2 commits, suggesting that cases persisting for as long as 90 days are likely to be exceptionally rare.

For each project, we analyzed only the job with the highest test execution frequency. Furthermore, the identification of the workflow, job, and step that run the tests was performed through manual inspection, which may introduce selection bias. Other jobs within the same project (running on different operating systems, language versions, or test layers) may also exhibit **Test Alert Snooze** occurrences that we do not detect. Consequently, the **Test Alert Snooze** rates reported in this study should be interpreted as lower bounds on the project-wide rates.

6.2 Construct Validity

We defined **Test Alert Snooze** based solely on the temporal persistence of test failures, regardless of their underlying causes. This persistence-only definition is a behavioral proxy rather than a direct measurement of developer attention or alert fatigue. Therefore, continuous test failures do not necessarily indicate that developers ignore **Test Alert Snooze**. Rather, the observed instances of **Test Alert Snooze** may arise from several distinct underlying mechanisms. Examples include flaky tests [18], whose failures are non-deterministic and unrelated to actual defects; active investigation of difficult bugs that legitimately requires multiple commits to resolve; and long-running build pipelines that produce repeated failures during normal execution. Moreover, the determination of resolution, defined as a transition from failing to passing, is similarly susceptible to flakiness-driven transient successes. In addition, because our analysis is grounded in commit history and CCS classification, developer activities that do not result in a commit (*e.g.*, local debugging) fall outside the scope of our observation.

Although these multiple factors may be intertwined – representing a limitation of our approach – the observable phenomenon that the same test fails repeatedly across consecutive revisions remains consistent with alert fatigue reported in other domains [21]. Therefore, treating these cases collectively as **Test Alert Snooze** provides a useful behavioral proxy for studying this class of failures, even when the underlying cognitive mechanism cannot be directly measured.

6.3 External Validity

We analyzed CI logs from 27 projects; however, test failures were observed in only 10 projects, and **Test Alert Snooze** occurrences in only 4 projects. As a result, the findings from RQ2 through RQ5 rest on a small project base. In particular, the 100% recurrence rate reported for one project is based on only 3 test methods and is therefore susceptible to outliers. Furthermore, the projects analyzed were drawn from prior studies [4, 5], and most of them are large and active open-source projects. Smaller projects or those with weaker testing discipline may exhibit different patterns. In this study, we focused on a selected set of projects with high test execution frequency and conducted a careful analysis; however, a more accurate assessment of **Test Alert Snooze** requires replication over a longer observation period and a larger set of projects.

This study is constrained to projects written in Java or Python that use GitHub Actions as their CI platform. Projects in other languages (*e.g.*, JavaScript, Go, C/C++) or those using other CI tools (*e.g.*, Jenkins, GitLab CI, CircleCI) may exhibit different **Test Alert Snooze** patterns due to differences in testing conventions, retry policies, and job structures. Investigations covering additional languages, CI platforms, and industrial settings are left as future work.

7 Conclusion

This paper introduced **Test Alert Snooze**, a state in which the same test method fails across two or more consecutive revisions on CI. While prior empirical work has examined the temporal persistence of CI failures at the build or job level [13, 10], persistence at the granularity of individual test methods has remained empirically uncharacterized. We formalized **Test Alert Snooze** as a behavioral proxy that captures this finer-grained persistence and provided its first empirical characterization.

Through an analysis of 27 GitHub Actions projects written in Python or Java over a 90-day period, we found that **Test Alert Snooze** accounts for 42.9% of observed test failures, with a median persistence of 2 consecutive revisions and approximately one day. While most occurrences are resolved promptly, extreme cases persisted for up to 11 commits and 16 days. Examination of developer activity during **Test Alert Snooze** revealed that explicit **fix** commits accounted for only 8.2%, with the majority being commits whose connection to the failing test is not immediately apparent. Furthermore, resolving commits were dominated by **test** (32.4%) rather than **fix** (11.8%), suggesting that the resolution of **Test Alert Snooze** frequently involves modifications to test code itself rather than to product code alone.

Data Availability Statement

To support reproducibility, we provide an anonymous replication package at <https://github.com/ayane-shi/TestAlertSnooze-Replication>. The repository contains data for the subject projects, covering revisions created during the 90-day period from Dec 21, 2025, to Mar 20, 2026. The dataset includes test failure logs, RefactoringMiner detection results, test details for each revision, and the scripts used for each research question.

References

- 1 Henri Aïdasso, Francis Bordeleau, and Ali Tizghadam. On the illusion of success: An empirical study of build reruns and silent failures in industrial CI. *CoRR*, abs/2509.14347, 2025. doi:10.48550/arXiv.2509.14347.
- 2 Pouria Alikhanifard and Nikolaos Tsantalis. A novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools. *ACM Transactions on Software Engineering and Methodology*, 34(2):40:1–40:63, 2025. doi:10.1145/3696002.
- 3 Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: an explorative analysis of travis CI with github. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017)*, pages 356–367. IEEE Computer Society, 2017. doi:10.1109/MSR.2017.62.
- 4 Islem Bouzenia and Michael Pradel. You name it, I run it: An LLM agent to execute tests of arbitrary projects. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1054–1076, 2025. doi:10.1145/3728922.
- 5 Runxiang Cheng, Kaiyao Ke, and Darko Marinov. pytest-ranking: A regression test prioritization tool for python. In *Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion 2025)*, pages 1089–1093, 2025. doi:10.1145/3696630.3728587.
- 6 Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. On the use of github actions in software development repositories. In *Proceedings of the 38th IEEE International Conference on Software Maintenance and Evolution (ICSME 2022)*, pages 235–245, 2022. doi:10.1109/ICSME55016.2022.00029.

- 7 GitHub Docs. Configuring the retention period for github actions artifacts and logs in your organization. Accessed: May 19th, 2026. URL: <https://docs.github.com/en/organizations/managing-organization-settings/configuring-the-retention-period-for-github-actions-artifacts-and-logs-in-your-organization>.
- 8 P.M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Signature Series. Pearson Education, 2007.
- 9 Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. Understanding flaky tests: the developer’s perspective. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*, pages 830–840, 2019. doi:10.1145/3338906.3338945.
- 10 Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. Noise and heterogeneity in historical build data: an empirical study of travis CI. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*, pages 87–97, 2018. doi:10.1145/3238147.3238171.
- 11 Sarra Habchi, Guillaume Haben, Mike Papadakis, Maxime Cordy, and Yves Le Traon. A qualitative study on the sources, impacts, and mitigation strategies of flaky tests. In *Proceedings of the 15th IEEE Conference on Software Testing, Verification and Validation (ICST 2022)*, pages 244–255, 2022. doi:10.1109/ICST53961.2022.00034.
- 12 Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, pages 426–437, 2016. doi:10.1145/2970276.2970358.
- 13 Xianhao Jin and Francisco Servant. A cost-efficient approach to building in continuous integration. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020)*, pages 13–25. ACM, 2020. doi:10.1145/3377811.3380437.
- 14 Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013. doi:10.1109/TSE.2012.70.
- 15 Nouredine Kerzazi, Foutse Khomh, and Bram Adams. Why do automated builds break? an empirical study. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME 2014)*, pages 41–50, 2014. doi:10.1109/ICSME.2014.26.
- 16 Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. idflakies: A framework for detecting and partially classifying flaky tests. In *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST 2019)*, pages 312–322, 2019. doi:10.1109/ICST.2019.00038.
- 17 William Levén, Hampus Broman, Terese Besker, and Richard Torkar. The broken windows theory applies to technical debt. *Empirical Software Engineering*, 29(4):73, 2024. doi:10.1007/S10664-024-10456-6.
- 18 Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, pages 643–653, 2014. doi:10.1145/2635868.2635920.
- 19 Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. Surveying the developer experience of flaky tests. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2022)*, pages 253–262, 2022. doi:10.1109/ICSE-SEIP55303.2022.9793965.
- 20 Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017)*, pages 345–355, 2017. doi:10.1109/MSR.2017.54.

- 21 Shahroz Tariq, Mohan Baruwal Chhetri, Surya Nepal, and Cécile Paris. Alert fatigue in security operations centres: Research challenges and opportunities. *ACM Computing Surveys*, 57(9):224:1–224:38, 2025. doi:10.1145/3723158.
- 22 Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. An empirical study of bugs in test code. In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME 2015)*, pages 101–110. IEEE Computer Society, 2015. doi:10.1109/ICSM.2015.7332456.
- 23 Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. Every build you break: developer-oriented assistance for build failure resolution. *Empirical Software Engineering*, 25(3):2218–2257, 2020. doi:10.1007/S10664-019-09765-Y.
- 24 Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. A tale of CI build failures: An open source and a financial organization perspective. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME 2017)*, pages 183–193, 2017. doi:10.1109/ICSME.2017.67.
- 25 Qunhong Zeng, Yuxia Zhang, Zhiqing Qiu, and Hui Liu. A first look at conventional commits classification. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE 2025)*, pages 2277–2289, 2025. doi:10.1109/ICSE55347.2025.00011.
- 26 Chen Zhang, Bihuan Chen, Xin Peng, and Wenyun Zhao. Buildsheriff: Change-aware test failure triage for continuous integration builds. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE 2022)*, pages 312–324, 2022. doi:10.1145/3510003.3510132.
- 27 Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: a large-scale empirical study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*, pages 60–71, 2017. doi:10.1109/ASE.2017.8115619.
- 28 Lianyu Zheng, Shuang Li, Xi Huang, Jiangnan Huang, Bin Lin, Jinfu Chen, and Jifeng Xuan. Why do github actions workflows fail? an empirical study. *ACM Transactions on Software Engineering and Methodology*, 35(5), 2026. doi:10.1145/3749371.
- 29 Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015)*, pages 913–923, 2015. doi:10.1109/ICSE.2015.101.