Smelly Bad Practices in Performance System Tests: Detection, Prevalence, and Lifetime

Sergio Di Meglio[§], Valeria Pontillo[¶], Luigi Libero Lucio Starace[§], Luana Martins[‖], Dario Di Nucci[‖], Fabio Palomba[‖]
[§]Software Quality and Intelligent Data-driven Systems (SQuIDS) Lab, University of Naples Federico II, Italy
Email: (sergio.dimeglio, luigiliberolucio.starace)@unina.it
[¶] Gran Sasso Science Institute (GSSI), L'Aquila, Italy
Email: valeria.pontillo@gssi.it
[‖] Software Engineering (SeSa) Lab, University of Salerno, Italy
Email: (lalmeidamartins, ddinucci, fpalomba)@unisa.it

***Abstract*—Performance testing is essential to ensure that software systems sustain realistic workloads and meet expected service levels under load. While prior research has extensively investigated performance testing methodologies, workload modeling, and tool support, there is limited consolidated knowledge regarding recurring bad practices that may silently compromise the reliability of performance evaluations. This paper provides the first catalog of bad practices in performance test code, derived through a gray literature review of practitioner sources. Building on this catalog, we introduce PT-LINTR, an automated tool that detects instances of these bad practices in Apache JMeter test artifacts. We apply PT-LINTR to the E2EGit dataset, analyzing 244 open-source test artifacts to investigate the prevalence, evolution, and lifespan of performance test bad practices, as well as the reasons behind their introduction, removal, and worsening over time. The study identified ten bad practices and found that they are widespread, often introduced near major releases during periods of high developer workload.**

***Index Terms*—Performance Testing; Bad Practices; Software Quality; Empirical Software Engineering.**

I. INTRODUCTION

Performance testing ensures that modern systems handle realistic workloads and meet service levels under load. By simulating concurrent users and monitoring metrics such as latency, throughput, and resource usage, it identifies scalability limits and bottlenecks before production [1], [2]. Inaccurate evaluations can lead to operational consequences [3], [4].

To support performance evaluation, practitioners rely on automated tools such as Apache JMeter and Locust, which enable the definition, execution, and monitoring of complex load scenarios [2], [5]. Additionally, over the past decades, research has extensively investigated performance testing methodologies, workload modeling techniques, and automated analysis approaches [6]–[9]. These contributions primarily provide prescriptive guidance on designing and executing performance tests. For instance, researchers have proposed systematic approaches for workload generation and characterization [6], [10], techniques for detecting performance regressions from execution data [8], and tools to automate the construction and execution of performance tests [6], [9]. In other words, these contributions primarily provide prescriptive guidance on designing and executing performance tests. While such guidance is essential, it largely reflects how performance testing should be conducted *under ideal conditions*. In practice, however, performance test artifacts are often developed incrementally and evolve alongside the system under test, being repeatedly adapted to changing requirements, time constraints, and release pressures [2]. As a result, recommended principles may be applied only partially, misunderstood, or gradually eroded over time, leading to deviations between intended testing practices and their concrete implementation. In this context, seemingly minor flaws in workload configuration, parameter handling, validation logic, or test structure may silently compromise the realism, validity, and interpretability of performance evaluation results. Despite their potential impact, much less attention has been devoted to identifying and systematically characterizing these recurring suboptimal practices as they emerge in real-world performance test artifacts. Indeed, within the scope of our research, we identify a notable gap in the empirical understanding of these practices and their diffusion in contemporary projects.

This gap is particularly striking when compared to other areas of software testing. In unit testing [11], [12] and GUI-level testing [13]–[15], researchers have systematically studied recurring suboptimal patterns, commonly referred to as test smells [16], that negatively affect maintainability, reliability, and test effectiveness. These studies have led to structured taxonomies [17], [18], automated detection techniques [19]–[22], and refactoring guidelines [14], [23]–[27].

In light of the observations above, this paper presents **the first empirical investigation into recurring bad practices in software performance testing**. We conducted a gray literature review to compile a catalog of performance bad practices, which informed the development of PT-LINTR, a Python-based static analysis tool that detects some of these bad practices in Apache JMeter test artifacts. We leveraged PT-LINTR on the E2EGit dataset [5], comprising 244 performance test artifacts. We analyzed both their current snapshots and full version histories to investigate (i) the prevalence of the identified bad practices, (ii) their evolution and lifespan across project history, and (iii) the contextual factors associated with their introduction, removal, and worsening.

The results show that the bad practices are widely distributed across performance test cases. Of the 244 analyzed test cases, only one exhibited no bad practices. When analyzing the lifetime of bad practices, we found that 90% of commits introduce the bad practice for the first time during test file creation or subsequent modifications. While commits rarely modify existing smelly code, new bad practices are frequently introduced near a release, whereas their removal tends to occur during off-cycle periods. Since introduction and removal occur during ongoing development activities, most of them involve experienced contributors and shared ownership.

II. Related Work

Despite the recognized importance of performance testing for ensuring system reliability and scalability, the existing literature lacks a systematic, empirically grounded investigation of bad practices in this domain. Prior work has predominantly concentrated on testing methodologies, workload modeling, tool support, and open research challenges, rather than on identifying recurring practitioner bad practices or anti-patterns that may compromise the validity of performance testing.

The most helpful contributions are those by Pargaonkar [28] and Hassan et al. [29]. Their work discusses metric selection, scenario design, workload realism, and large-scale result analysis, and provides valuable, experience-based recommendations for practitioners. However, the authors describe recommended processes and considerations, without systematically examining which incorrect or suboptimal practices repeatedly occur in real projects. Likewise, studies on workload characterization in cloud and web systems [30], [31] identify important modeling challenges, including insufficient workload attributes, limited granularity, unrealistic network assumptions, and difficulties in capturing workload evolution. Yet, they do not empirically investigate how these shortcomings manifest as concrete bad practices.

A parallel research stream has focused on evaluating and comparing performance testing tools, including Apache JMeter, LoadRunner, NeoLoad, and others [2], [32]–[37]. These studies analyze as dimensions the usability, scripting effort, reporting capabilities, regression support, and cost. While such analyses inform tool selection and adoption, they do not examine how these tools are misconfigured, misapplied, or gradually degraded within evolving codebases, nor do they assess the persistence of problematic testing patterns over time.

Our Contribution.

Our work adopts an empirical, practice-oriented approach: following the research methods of previous studies [38]–[40], we systematically identify and classify recurring bad practices in software performance testing. We then quantify the extent of these practices across projects to assess their practical significance. Finally, we investigate their temporal behavior to determine whether they are short-lived smells promptly corrected by developers or persistent anti-patterns that remain embedded in the testing infrastructure over extended periods. Through this longitudinal, large-scale analysis, we aim to complement existing guidance with evidence-based insights into how performance testing is conducted in practice.

III. Goal and Research Questions

The *goal* of this study is to overview the bad practices in performance testing activity with the *purpose* of empirically assessing (i) the practitioners' insights on the matter, (ii) the prevalence of these bad practices in real-world projects, and (iii) the lifetime of the bad practices previously identified. The *perspective* targets both researchers and practitioners: for researchers, it offers an evidence-based characterization of recurring anti-patterns and their evolution, enabling further empirical studies and tool support; for practitioners, it provides a data-driven view of common pitfalls, their frequency, and persistence, supporting more informed decisions when designing and maintaining performance test suites.

We structured the study around three research questions. First, we examined the most common bad practices in performance testing by combining practitioners' perspectives with evidence from real-world artifacts, clarifying which issues are considered problematic and how they manifest in test artifacts.

RQ$_1$. *What are the common bad practices in software performance testing?*

Building on the catalog, we quantified the presence of these practices in real-world projects. Through automated analysis of performance test artifacts, we measured their prevalence and diffusion in contemporary development ecosystems.

RQ$_2$. *How prevalent are software performance testing bad practices in real-world projects?*

Third, we adopted a longitudinal perspective to examine how bad practices evolve across project histories. We analyzed when they are introduced, whether they are improved or worsened, and how long they persist in performance test infrastructures, distinguishing short-lived issues from long-standing anti-patterns.

RQ$_3$. *What is the lifetime of a software performance testing bad practice?*

To address our research questions, we conducted a multi-phase empirical study combining qualitative and quantitative analyses. We first gathered practitioners' perspectives on bad practices in performance testing, identifying recurring issues and contextual factors to build a catalog grounded in experience. We then performed a large-scale analysis of performance testing artifacts from real-world repositories to (i) detect these practices, (ii) measure their prevalence, and (iii) examine their persistence over time. We adhered to established empirical software engineering guidelines [41] and aligned our

methodology with the *ACM/SIGSOFT Empirical Standards*.[1] We followed the *"General Standard"*, *"Repository Mining"*, and *"Mixed Methods"* guidelines.

IV. $\mathbf{RQ}_1$: What are the common bad practices in software performance testing?

A. Research Method

To identify an initial set of bad practices in software performance testing, we conducted a gray literature review to capture practitioners' perspectives and recurring issues discussed outside academic venues. The search string combined terms related to bad and best practices with terms related to performance testing and tools:

```
(``bad practices'' OR ``best practices'' OR
    ``recommendation'' OR ``tips'') AND
        ( ``performance test*'')
```

This structure reflects approaches adopted in prior studies [42], [43]. The first group of keywords refers to guidelines and practices discussed within practitioner communities, while the second group targets the concept of performance testing.

The search was performed using the GOOGLE search engine in incognito mode, which provides broad coverage of practitioner-oriented content such as technical blogs, documentation, and industry reports. The analysis was conducted between early March and mid-April 2025. For each query, the first 15 pages of results were examined, as recommended in previous work [42], since relevant documents rarely appeared beyond this threshold. This led to the inspection of approximately 1,800 documents. Given the broad scope of Google Search, we applied inclusion and exclusion criteria to filter the retrieved documents. The inclusion criterion (I1) requires that a document explicitly address bad practices in the development of performance testing. Regarding the exclusion criteria, we employed the same filtering rules adopted in prior work [42]. Specifically, a document was excluded if it met even one of the following criteria: (E1) it was not written in English; (E2) it provided only guidelines for manual testing or generic tool usage without addressing performance testing practices; (E3) it was a video or a book, which is challenging to analyze systematically; and (E4) its access was restricted by a paywall.

The first two authors, both with over three years of experience in performance testing, manually screened all retrieved documents and retained only those directly aligned with the study objectives, resulting in a final set of 139 primary sources. They then independently analyzed the selected documents and extracted the identified bad practices. The extracted practices were systematically recorded, traced back to their original sources, and organized into a catalog derived inductively from the data, without relying on a predefined classification scheme. The two independent catalogs produced by the authors were subsequently merged into a unified version. Disagreements over the identification or interpretation of specific practices were resolved through consultation with the third author, who is also experienced in performance testing.

[1]Available at: https://github.com/acmsigsoft/EmpiricalStandards.

B. Analysis of the Results

We identified ten bad practices in performance testing, summarized in Table I. These practices threaten the realism, validity, and interpretability of performance testing outcomes, often leading to misleading conclusions about system behavior.

Table I: Overview of the Performance Testing Bad Practices

ID	Bad Practice	# Documents
BP01	Unrealistic workload	43
BP02	Ignoring think times	31
BP03	Inadequate test duration	10
BP04	Improper ramp-up and cold-down configuration	35
BP05	Non-representative test environments	37
BP06	Ignoring network condition	16
BP07	Reliance on default request names	10
BP08	Overly rigid assertions	15
BP09	Lack of response validation	20
BP10	Excessive listeners and logging	14

A major source of distortion is the lack of realism in workload modeling. In particular, `Unrealistic workload (BP01)` emerges as the most frequently reported issue: test plans often simulate automated bots rather than real users, for instance, by repeatedly executing identical requests or ignoring realistic interaction sequences such as authentication flows or multi-step transactions. This problem is closely related to `Ignoring think times (BP02)`, whereby requests are issued without delays between user actions. Together, these practices generate artificial workloads that inflate system load and decouple test results from actual user experience. As a consequence, critical performance bottlenecks may remain hidden, while observed anomalies may reflect unrealistic test conditions rather than genuine system limitations.

Another group of bad practices concerns temporal and workload dynamics. `Inadequate test duration (BP03)`, characterized by tests that are either too short to reveal long-term issues (e.g., memory leaks) or unnecessarily long without yielding additional insights, reduces the effectiveness of testing activities. Similarly, `Improper ramp-up and cool-down configuration (BP04)` introduces abrupt load changes that lead to transient system states, such as cold caches or uninitialized resources, which rarely occur in production. These configurations may produce misleading performance spikes and result in tuning decisions based on unrealistic scenarios, thereby weakening the reliability of performance evaluations.

The representativeness of the testing context is further compromised by environmental factors. `Non-representative test environments (BP05)` arise when performance tests are executed in staging or simplified environments that differ from production in hardware capacity, service config-

uration, or network topology. This issue is compounded by `Ignoring network conditions (BP06)`, as many tests are conducted in fast local networks without simulating latency, bandwidth limitations, or unstable connections. Together, these practices yield overly optimistic performance assessments and conceal bottlenecks that may significantly affect user experience in real-world deployments, particularly in distributed, web-based, and mobile systems.

Beyond test design and execution, several bad practices affect the interpretability and validity of test artifacts and results. `Reliance on default request names (BP07)`, for instance, reduces the readability and maintainability of test scripts, increasing the risk of misinterpretation and hindering collaboration and debugging. At the level of validation, two opposite yet equally bad practices emerge: `Overly rigid assertions (BP08)`, which rely on fixed performance thresholds and may incorrectly classify acceptable systems as failing due to occasional outliers, and `Lack of response validation (BP09)`, which measures performance without verifying functional correctness and may produce deceptively favorable results even when the system returns erroneous outputs.

Finally, issues related to the testing infrastructure also matter. `Excessive listeners and logging (BP10)`, particularly in tools such as APACHE JMETER, introduce additional computational overhead during test execution. To minimize this effect, practitioners typically recommend using no more than three listeners during load tests. Exceeding this threshold increases memory and CPU consumption, distorts response times and throughput measurements, and makes it difficult to distinguish genuine system-under-test performance limitations from artifacts introduced by the testing setup.

V. $\mathbf{RQ}_2$: How prevalent are software performance testing bad practices in real-world projects?

A. Research Method

Once the preliminary catalog of performance testing bad practices had been identified, we analyzed which of them could be automatically detected. Several practices depend on external conditions or require contextual knowledge that cannot be reliably inferred from the test plan alone. In particular, violations of `BP01` (Unrealistic workload), `BP03` (Inadequate test duration), `BP05` (Non-representative test environments), and `BP06` (Ignoring network conditions) are often influenced by project-specific characteristics, execution environments, or external infrastructure, making them difficult to identify through static analysis of test artifacts.

We proposed PT-LINTR to support the automated detection of six bad practices that can be inferred directly from APACHE JMETER performance test specifications (Table II). PT-LINTR is a PYTHON-based tool designed to be extensible, enabling future support for multiple performance testing frameworks. In its default mode, the analysis targets the current version of the test plan; however, when linked to the corresponding GitHub repository, it can also produce historical reports that summarize the evolution of detected bad practices across commits. PT-LINTR produces two complementary output formats to support both interactive inspection and automated processing. First, it generates an HTML report that allows users to navigate the detected issues and explore their context within the test plan. Second, it provides the same structured information in JSON format, enabling downstream analysis, integration into CI pipelines, or further processing by external tools.

We manually analyzed 10% of the smelly test cases identified by the tool to evaluate its accuracy (Table II). Since a random sampling was performed independently for each bad practice at the test case level, test cases exhibiting different bad practices could be selected in more than one category, thereby increasing the effective proportion of manually analyzed test cases for some bad practices beyond the initial 10% target. The manual validation was performed by the first author, who has experience with Apache JMeter. For each sampled test case, the corresponding JMeter test plan was opened in the JMeter GUI, and the elements flagged by the tool were manually inspected. Since the considered bad practices were primarily structural (i.e., related to the presence, absence, or configuration of specific JMeter components), the first author verified whether each reported violation actually met the formal definition adopted in this study.

Finally, we empirically investigated the prevalence of bad practices in real-world software projects. Since PT-LINTR is designed for APACHE JMETER, we ran the it on 72 repositories of the E2EGIT dataset [5]. Out of 244 APACHE JMETER test cases in the dataset, 10 test cases were corrupted. Therefore, we ran our tool on the latest versions of the 234 analyzable tests (snapshot from August). After detecting all test smells, we apply descriptive statistics concerning the number of test cases with a given test smell and the number of occurrences of a given test smell per test case.

B. Analysis of the Results

The manual validation indicates a high level of precision across most of the analyzed bad practices (Table II). We identified a limited number of missed violations (5) for `BP02` and `BP10` due to the use of unsupported external plugins.

As shown Table III, bad practices in performance testing are widely distributed across projects with only one test case free from bad practices. Two bad practices are present in most test cases: `Ignoring think times` (`BP02`) occurs in 184 cases (78.6%), and `Improper ramp-up and cold-down configuration` (`BP04`) occurs in 173 cases (73.9%). Other bad practices occur in fewer than 6% of test cases, yet they result in a high frequency of violations. The `Reliance on default request names` (`BP07`) results in 275 violations across 11 cases, averaging 25 violations per test. The `Overly rigid assertion` (`BP08`) has 83 violations across 12 test cases, averaging 6.92 violations per test case. Differently, `Lack of response validation` (`BP09`) and `Excessive listeners and logging` (`BP10`) occur in 95 cases (40.59%) and 50 cases (21.36%), respectively.

Table II: PT-LINTR validation results and detection rule

Smell	Test Cases	Violations Analyzed	TP	Precision
BP02: verifies whether at least one timer element (e.g., Constant Timer, Gaussian Random Timer, or other JMETER timer components) is present between any two subsequent requests.	21	26	23	88.5%
BP04: analyzes whether the ramp-up or ramp-down time is set to 0 or 1 second, causing the workload to jump directly from 0 to the maximum number of users, or to drop abruptly from the maximum load to zero, without intermediate steps.	27	38	38	100%
BP07: checks whether sample requests retain the default JMETER names (e.g., "HTTP Request") instead of being renamed with meaningful identifiers.	28	33	33	100%
BP08: identifies overly rigid assertions, such as exact equality checks on response times or response sizes. Exceptions are made for standard HTTP status codes (e.g., "200" or "OK").	8	39	39	100%
BP09: verifies that each sampler in the APACHE JMETER test plan includes at least one response assertion.	9	27	26	96.3%
BP10: checks whether the test plan includes more than three listeners (e.g., View Results Tree, Summary Report, Aggregate Report, etc.), as suggested by practitioner guidelines identified in the grey literature review.	5	6	5	83.3%

Table III: Distribution of test smells and violation statistics

	Frequency		Violations per Test					
Smell	Tests	Violations	Avg/Test	Min	Q1	Median	Q3	Max
BP02	184	209	1.14	1	1	1	1	5
BP04	173	272	1.57	1	1	1	2	10
BP07	11	275	25.0	4	6	10	17	170
BP08	12	83	6.92	1	1	2.5	10.25	35
BP09	95	95	1.00	1	1	1	1	1
BP10	50	50	1.00	1	1	1	1	1

These findings confirm that, even considering only a subset of the practices we investigated, the identified bad practices are widespread in open-source performance test artifacts. This result indicates that practitioners either consciously accept them (e.g., valuing simplicity or speed over rigor), or unconsciously overlook them, suggesting a gap in awareness and tooling support. Interestingly, only one test did not present violations of any of the bad practices considered.

VI. $\mathbf{RQ}_3$: WHAT IS THE LIFETIME OF A SOFTWARE PERFORMANCE TESTING BAD PRACTICE?

A. Research Method

After characterizing their recurrence, we investigate the lifetime of software performance bad practices and analyze their evolution. Our goal is to characterize (i) when bad practices are introduced into performance tests, and (ii) how they evolve, i.e., whether they improve (decrease in number) or worsen (increase in number) as the test changes over time.

We reconstruct the history of each test artifact using PyDriller [44]. For every commit that modifies a test file, we execute PT-LINTR to detect the presence and number of violations for each bad practice. By comparing consecutive revisions, we identify three types of commits with respect to a given bad practice in a given test file: (i) 'INTRODUCTION COMMIT': the commit in which the bad practice first appears in the history of a test, which may correspond either to the creation of the test file or to a subsequent modification that introduced the first violation; (ii) 'IMPROVING COMMIT': a commit in which the number of violations of a given bad practice decreases compared to the previous revision (possibly reaching zero); and (iii) 'WORSENING COMMIT': a commit in which the number of violations of a given bad practice increases compared to the previous revision.

Our unit of analysis is a "*(commit, test file, bad practice)*" instance rather than a unique commit. In other words, we count a commit once for each bad practice in each test file whose number of violations changes in that revision. As a result, the same Git commit can contribute multiple observations (e.g., if it introduces two different bad practices or affects multiple test files). This is intentional and aligns with our goal of characterizing maintenance actions at the level of specific bad practices within individual test artifacts.

In addition, we also track NO-CHANGE instances, i.e., commits that modify a test file but leave the number of violations of a given bad practice unchanged. We use these instances as a baseline that captures the *exposure* of bad practices to test-modifying activity, and we leverage them to contextualize introduction, improving, and worsening commits by assessing whether change events are over- or under-represented relative to the overall population of edits to performance test artifacts.

To investigate contextual factors associated with introduction, improvement, and worsening, and to compare them against the no-change baseline, we annotate each commit with project- and developer-related tags (Table IV), following prior work on code smell analysis [12], [40], [45].

At the project level, we assign the *Project proximity* and *Project startup* tags. *Release proximity* measures the signed temporal distance in days between a commit and the closest major release, with negative values indicating commits performed before the release and positive values those performed after it. This metric allows us to assess whether introduction, improvement, or worsening events concentrate around release dates. We focus exclusively on major releases, identified by analyzing Git tags and commit messages, as

Table IV: Tags assigned to introduction, improving, and worsening commits

Type	Tag	Description	Values
Project	Release proximity	Signed number of days between the commit date and the closest major release date (negative = before, positive = after).	integer days
Project	Project startup	The commit was performed within [value] after the start of the project.	one week/one month/one year/more than one year
Dev	Activity level	Developer workload at the time of the commit, computed from commit activity in a fixed time window (and discretized into levels).	low/med/high
Dev	Ownership	Whether the developer is the main contributor (owner) of the test file up to that point in history.	true/false
Dev	Newcomer	Whether the developer is a newcomer at the time of the commit (e.g., within their first 3 commits in the project).	true/false

they represent substantial milestones, whereas minor releases typically address isolated bug fixes. *Project startup* captures the project's maturity at the time of the commit by discretizing the time elapsed since project inception into four intervals: within 1 week, within 1 month, within 1 year, and more than 1 year. The project start date is retrieved from the E2EGit dataset [5] and compared with the commit date, enabling us to examine whether and when bad practices are more frequently introduced, improved, or worsened.

In addition to project-level characteristics, we consider developer-related factors. The *Activity level* tag measures a developer's engagement within the target project at the time of a relevant commit. For each developer, we partition their contribution history into monthly windows starting from their first project commit and define activity in month m as the number of commits performed during that month. For each introduction, improvement, or worsening commit by developer d in month m, we compute the distribution of monthly commit counts of all developers active in the same project during that month. Let $Q1$ and $Q3$ denote the first and third quartiles of this distribution; we classify activity performed by d as low if it is below $Q1$, medium if it falls between $Q1$ (inclusive) and $Q3$ (exclusive), and high if it is at least $Q3$. This measure approximates in-project engagement rather than overall workload, as contributions outside the analyzed repository are not observable, and commit sizes may vary. Finally, we assign two additional developer-status tags. The *Ownership* tag is assigned when the developer who introduces, improves, or worsens the smell is responsible for more than 75% of the commits to the affected file, following Bird et al. [46]. The *Newcomer* tag is assigned when a smell-introducing commit belongs to the developer's first three commits in the project, allowing us to assess whether limited project experience correlates with a higher likelihood of introducing bad practices.

B. Analysis of the Results

We first summarize the total number of bad-practice instances and their distribution across types, then organize the results by tag and compare the distributions of introduction, improvement, and worsening commits.

Table V: Overall number of change instances by bad practice and variation type. Percentages (in parentheses) are computed within each commit-type group (column-wise) and rounded.

Bad practice	Introduction	Improving	Worsening
BP02	187 (34%)	3 (10%)	3 (10%)
BP04	177 (32%)	9 (30%)	11 (37%)
BP07	15 (3%)	8 (27%)	6 (20%)
BP08	16 (3%)	4 (13%)	7 (23%)
BP09	99 (18%)	4 (13%)	1 (3%)
BP10	52 (10%)	2 (7%)	2 (7%)
Total	546 (100%)	30 (100%)	30 (100%)

*1) **Overall distribution of bad-practice changes**:* Table V reports the overall number of observed bad-practice change instances, broken down by bad practice and by variation type (introduction, improving, and worsening). Overall, we identify 606 bad-practice change instances, with a strong skew toward introductions (546, 90.1% of bad-practice change instances), while improving and worsening instances are comparatively rare and perfectly balanced (30 each, ≈5% of the change instances). This global imbalance suggests that, among commits that do affect bad-practice counts, bad practices are far more frequently added than actively adjusted (either reduced or intensified) once they are present.

While these figures describe the composition of change instances, we also contextualize them against the no-change baseline. Across all test-modifying instances, we observe 5,776 no-change instances (i.e., bad-practice count remained unchanged for the considered practice), compared to 606 change instances. Thus, the majority of test-modifying activity does not alter the violation counts of the considered bad practices, and only 9.5% of instances result in a change in the bad-practice count (8.6% introductions, and 0.5% each improvement and worsening).

Table V also highlights a highly uneven distribution across bad practices. Introductions are dominated by `BP02` (*Ignoring think times*) and `BP04` (*Improper ramp-up/cool-down*), which together account for roughly two-thirds of all introduction instances. A second tier includes `BP09` (*Lack of*

Table VI: Release-cycle proximity summary by variation type. Percentages refer to variations occurring within $\pm k$ days from the closest major release. Median and IQR are computed on the absolute distance $|d|$ (days).

	Variation type			
Metric	**No-change**	**Introd.**	**Improv.**	**Worsen.**
Num. of commits	4246	501	30	28
Within ±7 days (%)	47.9	40.1	36.7	53.6
Within ±14 days (%)	54.9	48.3	36.7	75.0
Within ±30 days (%)	66.9	55.5	50.0	75.0
Median $\|d\|$	10	15	34	5
IQR $\|d\|$	47.0	62.0	48.8	19.0

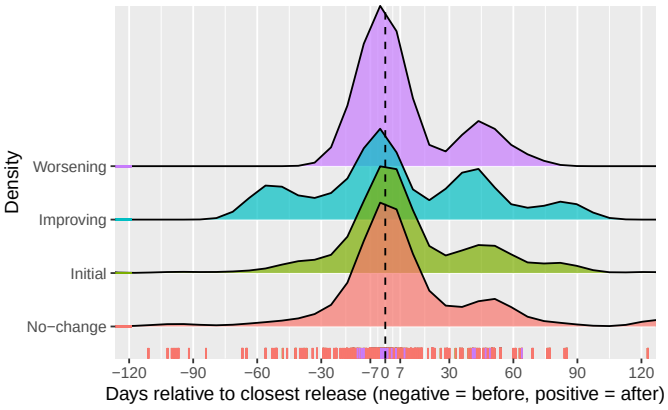

Figure 1: Ridge plot of the signed distance (in days) between each commit and its closest major release, stratified by variation type. The vertical line at day 0 marks the closest release date. Rug marks show individual observations.

response validation) and `BP10` (*Excessive listeners*), while `BP07` (*Reliance on default request names*) and `BP08` (*Overly rigid assertions*) are introduced less often. Despite their low introduction counts, `BP07` and `BP08` exhibit non-negligible post-introduction activity with 8 improving/6 worsening and 4 improving/7 worsening, respectively. This result suggests that these issues are more likely to be revisited as tests evolve: developers may improve them when refactoring for readability, maintainability, or robustness (e.g., renaming requests or relaxing brittle checks), but may also worsen them when making expedient changes to get a test working (e.g., adding quick, overly strict validations or leaving default names in place) during iterative debugging and adaptation of the workload.

Finally, the No-change instances are themselves broadly distributed across bad practices (each accounting for roughly 13–19% of the no-change population), indicating that the predominance of unchanged violation counts is not driven by a single bad practice, but is a general characteristic of performance-test modifications in our dataset.

*2) **Release-cycle proximity**:* We computed release-cycle proximity only for repositories with at least one identifiable major release. Consequently, we excluded 47 change instances from this analysis (45 introductions and 2 worsening) and 1,530 instances from no-change variations, as they originated from projects for which no releases could be retrieved. The release-cycle analysis points to a clear asymmetry between worsening and improving changes. As shown in the ridge plot in Figure 1 and the summary in Table VI, worsening commits are strongly concentrated around major releases: more than half occur within ±7 days of the closest release (53.6%), and three quarters fall within ±14 days (75%). This concentration is also reflected in the central tendency: the median absolute distance to the closest release is only 5 days for worsening commits, with a relatively tight spread (IQR 19). In the ridge plot, this appears as a sharp peak around day 0, indicating that deterioration in test practices tends to happen during the most release-adjacent period.

In contrast, the distribution of improving commits in Figure 1 is more dispersed and less centered on the release date, which aligns with the statistics in Table VI. Only 36.7% of improving commits fall within ±7 days (and the same proportion within ±14), and their median absolute distance is much larger and amounts to 34 days (IQR 48.8). Together, the figure and table suggest that improvements are less closely coupled with deadline-adjacent activity and instead occur more "*off-cycle*", likely reflecting work scheduled during less time-constrained phases (e.g., routine maintenance, test refactoring, or follow-up hardening after observing performance behavior).

Introduction commits show a broader distribution of introductions than worsening, as Table VI reports a median distance of 15 days and a wide IQR (62). However, when contrasted with the no-change baseline, introductions are not particularly release-centered: only 40.1% fall within ±7 days, compared to 47.9% for no-change. This result indicates that introductions occur throughout development and are not disproportionately concentrated around releases. Conversely, worsening appears genuinely release-adjacent relative to baseline activity, especially within ±14 days (75.0% vs. 54.9% for No-change), suggesting that the risk of exacerbating existing poor practices increases during release-intensive periods.

*3) **Project startup**:* To understand whether bad practices are mainly a "startup artifact" or rather emerge as the test suite matures, we analyzed the *Project startup* tag across introduction, improvement, and worsening commits. The results are reported in Table VII, and indicate that the project startup phase accounts for only a small fraction of observed bad-practice changes. Across all three categories, instances occur predominantly well after the initial ramp-up of the project: more than three-quarters of introductions, improvements, and worsening occur beyond the first year of project life, and roughly one-fifth occur between one month and one year.

This skew is stronger than that observed for the NO-CHANGE baseline, where instances are mostly concentrated beyond the first year (62%). Still, they exhibit a comparatively larger share in the first year (30% in the "one year" bin and 8% within the week/month bin). This contrast suggests that while performance test files are modified throughout

Table VII: Project-startup tags by bad practice and variation type. Each cell reports the percentage of instances in a startup time bin, computed *row-wise within each outcome group*. Percentages are rounded.

Bad practice	**No-change** (n=5776)				**Introduction** (n=546)				**Improving** (n=30)				**Worsening** (n=30)			
	1w	**1m**	**1y**	**>1y**	**1w**	**1m**	**1y**	**>1y**	**1w**	**1m**	**1y**	**>1y**	**1w**	**1m**	**1y**	**>1y**
BP02	1%	6%	27%	65%	5%	5%	21%	68%	0%	0%	0%	100%	0%	0%	0%	100%
BP04	1%	7%	28%	63%	4%	1%	20%	75%	0%	0%	33%	67%	0%	9%	9%	82%
BP07	2%	6%	31%	62%	13%	0%	13%	73%	0%	0%	50%	50%	0%	0%	83%	17%
BP08	2%	6%	32%	60%	0%	0%	0%	100%	0%	0%	0%	100%	0%	0%	0%	100%
BP09	2%	6%	31%	61%	0%	1%	23%	76%	0%	0%	0%	100%	0%	0%	0%	100%
BP10	2%	6%	31%	62%	6%	0%	10%	85%	0%	0%	0%	100%	0%	0%	0%	100%
Total	2%	6%	30%	62%	4%	2%	18%	77%	0%	0%	23%	77%	0%	3%	20%	77%

the project lifecycle, changes that actually alter bad-practice counts tend to occur later, during more mature stages of test-suite evolution. Conversely, the "early" phase (within one week/month) is scarcely represented for bad-practice count changes. Only 6% of introduction commits occur in these earliest bins; no improving commits fall within them, and worsening activity is nearly absent, with only a single commit in the one-month bin and none in the first week of project life.

Taken together, these results suggest that bad practices are not primarily a transient effect of initial project bootstrapping, but rather a phenomenon that emerges and is later maintained or adjusted during the longer-term adoption, evolution, and maintenance of performance tests. This result aligns with prior work showing that performance testing is often introduced relatively late in a project's lifecycle [2]. Accordingly, the concentration of bad-practice-related changes beyond the first year does not necessarily signal "late problem emergence" in a mature testing setup. Instead, it likely reflects the delayed introduction of performance tests themselves, which shifts both the emergence and evolution of related bad practices away from the project's initial phase.

At the level of individual bad practices, Table VII shows a broadly consistent pattern, with most activity occurring after the first year. Some differences emerge in how the remaining activity is distributed, but several bad-practice/commit-type combinations rely on very few observations (sometimes fewer than 10), so percentages should be interpreted descriptively rather than inferentially.

For introduction commits, most bad practices emerge after the first year, with limited early cases. `BP02` appears 10% within one month (5% within one week), and `BP07` 13% within one week. About one-fifth of `BP02`, `BP04`, and `BP09` introductions fall in the "1y" bin, whereas `BP10` is largely introduced after one year. `BP08` appears only beyond the first year. Relative to the no-change baseline, where roughly 60–65% of instances also occur after one year, the late concentration is not unique to change events. However, introductions are even more skewed toward later years for some practices, especially `BP10` and `BP08`, suggesting that these tend to arise in more mature phases of test-suite evolution.

A possible explanation is that `BP02` and `BP04` are often introduced during initial test scaffolding, whereas `BP10` emerges later as tests become embedded in routine execution and troubleshooting. The early presence of `BP07` may similarly reflect quick initial setups or imported scripts where structure and readability are deferred.

Improving and worsening commits are even more concentrated after the first year. Only `BP04` and `BP07` show any improvement and worsening activity before one year (`BP04`: 33% of improvements, 18% of worsening; `BP07`: 50% of improvements, 83% of worsening, in the "1y" bin). This pattern suggests that load-profile tuning (`BP04`) and test-plan organization (`BP07`) are revisited relatively early during stabilization. In contrast, other practices show almost no early corrective activity, implying they are typically addressed only after the test suite is more established, though small counts warrant cautious interpretation.

*4) **Activity level**:* The results of our activity level analysis are reported in Table VIII. In no-change instances, for which we use an exposure baseline, only 4% of instances occur under low activity, while medium and high activity account for the remainder in equal shares (48% and 48%). This indicates that, when developers edit performance test files, the vast majority of bad-practice variation instances they "touch" occur during medium-to-high activity periods.

Against this baseline, bad-practice changes show a clearer tilt toward high activity. Introductions are more concentrated under high activity than under no change (57% vs. 48%), and worsening is even more skewed (63% vs. 48%). This suggests that during intense development periods, edits are not only more frequent but also more likely to alter bad-practice counts, particularly by exacerbating existing violations. Improving instances exhibit a slightly different profile: they are more frequent under medium activity (53%) than under high (47%), with no cases under low activity. While the lack of low-activity improvements prevents a direct low-vs-high comparison, the shift from high to medium is consistent with improvements being more feasible during periods of reduced intensity than during introductions and worsening.

Looking across bad practices, introductions consistently skew toward high workload, particularly for `BP08` (75%), `BP09` (65%), and `BP10` (58%), and remain high for the two most frequent practices `BP02` and `BP04` (52% and 56%). For improvements and worsening, the same tendency persists

Table VIII: Developer activity level at the time of performance-test modifications, by bad practice and variation category. Cells report percentages computed *row-wise within each variation type* (Low/Medium/High sum to 100%).

Bad practice	**No-change** (n=5776)			**Introduction** (n=546)			**Improving** (n=30)			**Worsening** (n=30)		
	Low (%)	**Med (%)**	**High (%)**	**Low (%)**	**Med (%)**	**High (%)**	**Low (%)**	**Med (%)**	**High (%)**	**Low (%)**	**Med (%)**	**High (%)**
BP02	4	48	47	6	42	52	0	67	33	0	67	33
BP04	4	49	47	6	38	56	0	78	22	0	45	55
BP07	4	47	49	7	40	53	0	25	75	0	0	100
BP08	4	47	49	12	12	75	0	100	0	29	29	43
BP09	4	49	47	7	28	65	0	0	100	0	0	100
BP10	4	45	51	8	35	58	0	50	50	0	0	100
Overall	4	48	48	7	37	57	0	53	47	7	30	63

across most practices, with several categories observed almost exclusively at medium/high workload levels (e.g., `BP09`, `BP10` for worsening; `BP08`, `BP09` for improving).

*5) **Ownership and Newcomer experience**:* We further analyze the developer-related dimension of bad-practice changes by considering two complementary signals: whether the author is the *owner* of the modified test artifact and whether the author is a *newcomer* to the project at the time of the change. Table IX reports the distribution of bad-practice change instances across these two dimensions, split by variation type and bad practice, using no-change as a baseline.

Overall, ownership shares are remarkably stable across variation types. As shown in Table IX, owner involvement is close to half of the instances for both no-change (43%) and change variations (45% for introductions, 30% for improvements, and 43% for worsening). This suggests that bad-practice changes are not primarily driven by a single subgroup, such as non-owners "breaking" someone else's tests, but rather occur in collaborative maintenance, where both owners and non-owners contribute to performance test evolution. The only clearer deviation is in improving instances, where owner shares are lower (30%), suggesting that improvements are more often performed by active contributors beyond the primary file owners. One plausible explanation could be that non-owners provide a "fresh eyes" effect: developers who did not originally create a test may be more likely to notice suboptimal patterns and refactor them, whereas owners may be more cautious about altering tests they rely on.

Newcomer involvement varies more markedly across outcomes than in the baseline. While newcomers account for 18% of no-change instances, they represent 24% of introductions, indicating a modest over-representation relative to their overall participation in performance-test edits. In contrast, their share declines sharply for improvements (10%) and is absent for worsening instances. This pattern suggests that newcomers primarily contribute edits that either preserve the status quo or introduce new bad practices, whereas subsequent refinements are undertaken almost exclusively by established contributors. A possible explanation is that modifying and tuning performance tests requires project-specific expertise and confidence, which short-tenure contributors may lack.

At the level of individual bad practices, Table IX shows additional nuances. In the no-change baseline, newcomer participation is higher for `BP07`/`BP08` (21%) than for `BP02`/`BP04` (14%), suggesting that newcomers may more often touch "surface" aspects of test scripts (e.g., naming or assertions) than configuration-level practices. For introduction instances, `BP02` shows the highest newcomer share (28%), while `BP08` introductions involve no newcomers, hinting that some bad practices are introduced mainly by established contributors.

These findings complement the workload activity level analysis by indicating that both improvements and degradations are mostly associated with experienced contributors and ongoing development activity, rather than being driven primarily by newcomers or isolated ownership dynamics.

VII. Threats to Validity

We discuss the possible limitations of our study below, along with the main types of validity [41].

1) Construct Validity: To detect performance bad practices, we have defined a set of rules and implemented them in an automated tool, called PT-Lintr. We are aware that our results may be affected by false positives and false negatives. To overcome this limitation, we manually validated a subset corresponding to 10% of detected smell violations, in line with previous work [47]. This manual validation was performed by the first author, and the results were discussed with the second author. As reported in Table II, 169 smells were validated in total. The manual validation results indicated a mean precision of 97%. Only five cases were not correctly identified by the tool due to the use of external plugins that were not supported in the tool's current version at the time of the analysis.

We rely on Git author metadata rather than committer information because not all authors have commit privileges in open-source projects; using committers would therefore yield an incomplete view of actual contributions. Nonetheless, authorship metadata may be inaccurate or incomplete.

The *Workload* tag captures activity within a single project, whereas developers may simultaneously contribute to other projects. Measuring workload across the broader ecosystem could mitigate this limitation, but would introduce new biases, e.g., inflating workload for developers active in multiple projects within the same ecosystem while underestimating those engaged elsewhere. Finally, approximating workload by

Table IX: Ownership and newcomer experience by bad practice and variation category. We report only the percentage of instances where the tag is **True** (i.e., authored by an *owner* or a *newcomer*); **False** is implied. Percentages are computed within each variation type and bad practice and rounded.

Bad practice	**No-change** (n=5776)		**Introduction** (n=546)		**Improving** (n=30)		**Worsening** (n=30)	
	Owner True (%)	**Newcomer True (%)**	**Owner True (%)**	**Newcomer True (%)**	**Owner True (%)**	**Newcomer True (%)**	**Owner True (%)**	**Newcomer True (%)**
BP02	37	14	46	28	67	0	33	0
BP04	38	14	44	25	11	33	55	0
BP07	49	21	33	13	50	0	33	0
BP08	47	21	31	0	0	0	29	0
BP09	40	20	53	24	50	0	100	0
BP10	46	18	40	15	0	0	50	0
Total	43	18	45	24	30	10	43	0

commit counts is inherently rough. We deliberately avoid using commit size as a proxy for effort, since small commits can still entail substantial development work.

2) Internal Validity: For the literature review, the primary concern is the potential bias during the selection and classification of documents. To mitigate this, we repeated searches with varied queries and carefully validated the relevance of selected documents. Still, we do not claim that the corpus is complete. Another source of bias is manual classification, as there is no ground truth for categorizing performance testing guidelines; therefore, our catalog is based on interpretation. To reduce subjectivity, the classification was iteratively discussed among authors and validated in multiple rounds.

In RQ_3, we examined tags capturing selected aspects of the project lifecycle, including commit characteristics, developer roles, and project status. Other factors may also influence the introduction of smell. However, establishing causal relationships between the introduction of smell and product or process factors is beyond the scope of this study.

3) External Validity: The GLR is limited to publicly accessible practitioner platforms and may not reflect private discussions or industry-internal practices. We attempted to mitigate these threats by ensuring broad coverage across domains and contexts; however, we recognize that further work, particularly through industrial case studies, would be valuable to deepen our perspective and validate our findings in other settings.

4) Conclusion Validity: Part of our interpretation is based on manual classification and subjective judgment, which may introduce bias. This threat was mitigated through multiple rounds of cross-checking and discussion among authors to reach consensus on coding and interpretation. Despite these precautions, we acknowledge that further replication or complementary investigations would help validate and extend our findings. In this sense, the material released may serve as a basis for further replications.

VIII. Discussion and Conclusion

We presented an investigation of recurring bad practices in software performance testing. Starting from a GLR, we derived a catalog of ten bad practices and operationalized six of them in PT-LINTR, an automated detector for Apache JMeter artifacts. Its application to 234 analyzable performance tests shows that bad practices are pervasive: almost all tests contain at least one violation, with `Ignoring think times` and `Improper ramp-up/cool-down configuration` dominating both in frequency and distribution across projects. Beyond prevalence, our longitudinal analysis highlights a structural asymmetry in the evolution of these practices. Introductions vastly outnumber improvements, and once embedded, bad practices tend to persist rather than being systematically addressed. Worsening commits cluster around major releases, suggesting that release pressure coincides with degradations in performance test quality. Moreover, most changes occur well beyond the first year of project life, indicating that these issues are not merely artifacts of early project bootstrapping but emerge and evolve over the long-term maintenance phase. Finally, both improvements and degradations are primarily associated with experienced and active contributors rather than newcomers, reinforcing the view that performance tests are maintained by a relatively small and established subset of developers.

Our findings reveal a gap between performance testing guidelines and practice. For *researchers*, this study positions performance test bad practices as a distinct research stream, analogous to test smells in other testing domains, and provides both a catalog and a detection infrastructure to support refinement and extension. The concentration of worsening events near releases and under high development activity suggests opportunities for predictive models and CI-integrated linter that proactively flag high-risk modifications.

For *practitioners*, the results indicate that performance test quality does not improve organically over time and therefore requires explicit governance. Preventive controls, such as automated linting in continuous integration, targeted review guidelines for performance tests, and stabilization phases before major releases, are likely more effective than post hoc cleanup. More broadly, performance test artifacts should be treated as first-class, evolving assets whose design and maintenance demand the same rigor applied to production code, rather than as disposable scripts executed only at release time.

Data Availability

We provide a replication package [48] that includes the data and scripts to rerun the experiments.

Acknowledgment

The work has been partially supported by the European *HORIZON-KDT-JU-2023-2-RIA research project MATISSE* (grant 101140216-2, KDT232RIA_00017) and by the Italian PNRR MUR project PE0000013-FAIR.

References

[1] E. Battista, S. Di Martino, S. Di Meglio, F. Scippacercola, and L. L. L. Starace, "E2e-loader: A framework to support performance testing of web applications," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2023, pp. 351–361.

[2] S. Di Meglio, L. L. L. Starace, V. Pontillo, R. Opdebeeck, C. De Roover, and S. Di Martino, "Performance testing in open-source web projects: Adoption, maintenance, and a change taxonomy," in *41st IEEE International Conference on Software Maintenance and Evolution (ICSME 2025)*. IEEE, 2025.

[3] Z. M. Jiang and A. E. Hassan, "A survey on load testing of large-scale software systems," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1091–1118, 2015.

[4] K. Eaton, "How one second could cost amazon $1.6 billion in sales," 2012, last accessed: Oct. 24, 2022. [Online]. Available: https://web.archive.org/web/20221006004855/https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales

[5] S. Di Meglio, L. L. L. Starace, V. Pontillo, R. Opdebeeck, C. De Roover, and S. Di Martino, "E2egit: A dataset of end-to-end web tests in open source projects," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE/ACM, 2025, pp. 10–15.

[6] S. Di Meglio, L. L. L. Starace, and S. Di Martino, "Web app performance testing in industrial contexts: Supporting workload generation with e2e-loader++," *Journal of Systems and Software*, vol. 232, p. 112684, 2026. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412122500353X

[7] S. Di Meglio, L. L. L. Starace, and S. Di Martino, "E2E-Loader: A tool to generate performance tests from end-to-end gui-level tests," in *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2025.

[8] W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using regression models on clustered performance counters," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 15–26. [Online]. Available: https://doi.org/10.1145/2668930.2688052

[9] J. Chen, W. Shang, A. E. Hassan, Y. Wang, and J. Lin, "An experience report of generating load tests using log-recovered workloads at varying granularities of user behaviour," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 669–681.

[10] C. Vögele, A. van Hoorn, E. Schulz, W. Hasselbring, and H. Krcmar, "Wessbas: extraction of probabilistic workload specifications for load testing and performance prediction—a model-driven approach for session-based application systems," *Software & Systems Modeling*, vol. 17, no. 2, pp. 443–477, 2018.

[11] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *2012 28th IEEE international conference on software maintenance (ICSM)*. IEEE, 2012, pp. 56–65.

[12] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 4–15.

[13] R. Rwemalika, S. Habchi, M. Papadakis, Y. Le Traon, and M.-C. Brasseur, "Smells in system user interactive tests," *Empirical Software Engineering*, vol. 28, no. 1, p. 20, 2023.

[14] T. Fulcini, G. Garaccione, R. Coppola, L. Ardito, and M. Torchiano, "Guidelines for gui testing maintenance: a linter for test smell detection," in *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*, 2022, pp. 17–24.

[15] B. Yang, Z. Xing, X. Xia, C. Chen, D. Ye, and S. Li, "Uis-hunter: Detecting ui design smells in android apps," in *2021 IEEE/ACM 43rd international conference on software engineering: companion proceedings (ICSE-companion)*. IEEE, 2021, pp. 89–92.

[16] A. Deursen, L. M. Moonen, A. Bergh, and G. Kok, "Refactoring test code," CWI (Centre for Mathematics and Computer Science), NLD, Tech. Rep., 2001.

[17] W. Aljedaani, A. Peruma, A. Aljohani, M. Alotaibi, M. W. Mkaouer, A. Ouni, C. D. Newman, A. Ghallab, and S. Ludi, "Test smell detection tools: A systematic mapping study," in *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, 2021, pp. 170–180.

[18] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of systems and software*, vol. 138, pp. 52–81, 2018.

[19] F. Palomba, A. Zaidman, and A. De Lucia, "Automatic test smell detection using information retrieval techniques," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 311–322.

[20] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "Tsdetect: An open source test smells detection tool," in *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1650–1654.

[21] V. Pontillo, D. Amoroso d'Aragona, F. Pecorelli, D. Di Nucci, F. Ferrucci, and F. Palomba, "Machine learning-based test smell detection," *Empirical Software Engineering*, vol. 29, no. 2, p. 55, 2024.

[22] V. Pontillo, L. Martins, I. Machado, F. Palomba, and F. Ferrucci, "An empirical investigation into the capabilities of anomaly detection approaches for test smell detection," *Journal of Systems and Software*, vol. 222, p. 112320, 2025.

[23] F. Pecorelli, G. Grano, F. Palomba, H. C. Gall, and A. De Lucia, "Toward granular search-based automatic unit test case generation," *Empirical Software Engineering*, vol. 29, no. 4, p. 71, 2024.

[24] S. Di Meglio, L. L. L. Starace, V. Pontillo, R. Opdebeeck, C. De Roover, and S. Di Martino, "Investigating the adoption and maintenance of web gui testing: Insights from github repositories," *Information and Software Technology*, vol. 189, p. 107928, 2026. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584925002678

[25] L. Martins, H. Costa, M. Ribeiro, F. Palomba, and I. Machado, "Automating test-specific refactoring mining: A mixed-method investigation," in *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2023, pp. 13–24.

[26] E. Soares, M. Ribeiro, R. Gheyi, G. Amaral, and A. Santos, "Refactoring test smells with junit 5: Why should developers keep up-to-date?" *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1152–1170, 2022.

[27] L. Martins, V. Pontillo, H. Costa, F. Ferrucci, F. Palomba, and I. Machado, "Test code refactoring unveiled: where and how does it affect test code quality and effectiveness?" *Empirical Software Engineering*, vol. 30, no. 1, p. 27, 2025.

[28] S. Pargaonkar, "A comprehensive review of performance testing methodologies and best practices: software quality engineering," *International Journal of Science and Research (IJSR)*, vol. 12, no. 8, pp. 2008–2014, 2023.

[29] T.-H. Chen, M. D. Syer, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Analytics-driven load testing: An industrial experience report on load testing of large-scale systems," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp. 243–252.

[30] S. Shishira, A. Kandasamy, and K. Chandrasekaran, "Workload characterization: Survey of current approaches and research challenges," in *Proceedings of the 7th international conference on computer and communication technology*, 2017, pp. 151–156.

[31] M. Curiel and A. Pont, "Workload generators for web-based systems: Characteristics, current status, and challenges," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1526–1546, 2018.

[32] R. Abbas, Z. Sultan, and S. N. Bhatti, "Comparative analysis of automated load testing tools: Apache jmeter, microsoft visual studio (tfs), loadrunner, siege," in *2017 international conference on communication technologies (comtech)*. IEEE, 2017, pp. 39–44.

[33] V. Chandel, S. Patial, and S. Guleria, "Comparative study of testing tools: Apache jmeter and load runner," *International Journal of Computing and Corporate Research*, vol. 3, no. 3, pp. 1–7, 2013.

[34] A. Arora and M. Sinha, "Web application testing: A review on techniques, tools and state of art," *International Journal of Scientific & Engineering Research*, vol. 3, no. 2, p. 1, 2012.

[35] S. T. Rina, "A comparative study of performance testing tools," *International Journal of Advanced Research in Computer Science and Software Engineering (IJARCSSE)*, vol. 3, no. 5, pp. 1300–1307, 2013.

[36] D. S. Afroz, N. E. Rani, and N. I. Priyadarshini, "Web application-a study on comparing software testing tools," *International Journal of Computer Science and Telecommunications*, vol. 2, no. 3, pp. 1–6, 2011.

[37] M. D. M. Suffian and F. R. Fahrurazi, "Performance testing: Analyzing differences of response time between performance testing tools," in *2012 International Conference on Computer & Information Science (ICCIS)*, vol. 2. IEEE, 2012, pp. 919–923.

[38] G. Recupito, G. Giordano, F. Ferrucci, D. Di Nucci, and F. Palomba, "When code smells meet ml: on the lifecycle of ml-specific code smells in ml-enabled systems," *Empirical Software Engineering*, vol. 30, no. 5, p. 139, 2025.

[39] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2014.

[40] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 403–414.

[41] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén *et al.*, *Experimentation in software engineering*. Springer, 2012, vol. 236.

[42] F. Ricca and A. Stocco, "Web test automation: Insights from the grey literature," in *International Conference on Current Trends in Theory and Practice of Informatics*. Springer, 2021, pp. 472–485.

[43] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and software technology*, vol. 106, pp. 101–121, 2019.

[44] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, 2018, p. 908–911.

[45] C. Vassallo, G. Grano, F. Palomba, H. C. Gall, and A. Bacchelli, "A large-scale empirical exploration on refactoring activities in open source software projects," *Science of Computer Programming*, vol. 180, pp. 1–15, 2019.

[46] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code! examining the effects of ownership on software quality," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 4–14.

[47] R. Opdebeeck, A. Zerouali, and C. De Roover, "Smelly variables in ansible infrastructure code: Detection, prevalence, and lifetime," in *Proceedings of the 19th international conference on mining software repositories*, 2022, pp. 61–72.

[48] Anonymous, "Online appenxdix." [Online]. Available: https://doi.org/10.5281/zenodo.18872673