A Taxonomy of Bad Practices in Software Performance Testing: Insights from Gray Literature and Practitioner Validation

SERGIO DI MEGLIO and LUIGI LIBERO LUCIO STARACE, SQuIDS Lab, Department of Electrical Engineering and Information Technology, University of Naples Federico II, Italy
VALERIA PONTILLO, Gran Sasso Science Institute (GSSI), Italy
LUANA MARTINS, DARIO DI NUCCI, and FABIO PALOMBA, Software Engineering (SeSa) Lab, University of Salerno, Italy

Performance testing is essential to ensure that software systems can sustain realistic workloads and meet expected service levels under load. While the research community has extensively studied performance testing methodologies, workload modeling, and tool support, little is known, and no consolidated knowledge exists, about how performance tests are actually implemented in practice and about the recurring mistakes that may silently compromise the reliability of performance evaluations.

This paper presents the first systematic investigation of bad practices in software performance testing. We focus on the recurring mistakes, misconceptions, and test smells that arise during the design, implementation, and execution of performance tests. To this end, we design a two-phase mixed-methods study that includes a gray literature review of 415 practitioner-oriented sources and a practitioner study comprising a survey and follow-up interviews with experienced performance engineers. The study yields an empirically grounded taxonomy of 29 bad practices and reveals how practitioners perceive their severity, prevalence, and impact on real-world performance evaluations. These findings offer an empirically grounded foundation for improving workload modeling and strengthening the overall quality and trustworthiness of performance testing processes.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Software performance**; **Empirical software validation**.

Additional Key Words and Phrases: Performance Testing, Bad Practices, Software Quality Assurance, Empirical Software Engineering.

ACM Reference Format:
Sergio Di Meglio, Luigi Libero Lucio Starace, Valeria Pontillo, Luana Martins, Dario Di Nucci, and Fabio Palomba. 2026. A Taxonomy of Bad Practices in Software Performance Testing: Insights from Gray Literature and Practitioner Validation. *ACM Trans. Softw. Eng. Methodol.* 37, 4, Article 111 (June 2026), 50 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

In the software domain, where systems often serve as business-critical platforms and must support large numbers of concurrent users, performance testing plays a central role in identifying

Authors' Contact Information: Sergio Di Meglio, sergio.dimeglio@unina.it; Luigi Libero Lucio Starace, luigiliberolucio.starace@unina.it, SQuIDS Lab, Department of Electrical Engineering and Information Technology, University of Naples Federico II, Napoli, Italy; Valeria Pontillo, valeria.pontillo@gssi.it, Gran Sasso Science Institute (GSSI), L'Aquila, Italy; Luana Martins, lalmeidamartins@unisa.it; Dario Di Nucci, ddinucci@unisa.it; Fabio Palomba, fpalomba@unisa.it, Software Engineering (SeSa) Lab, University of Salerno, Salerno, Italy.

ACM 1557-7392/2026/6-ART111
https://doi.org/XXXXXXX.XXXXXXX

bottlenecks, evaluating scalability, and assessing compliance with service-level agreements [26]. This is true across a wide spectrum of software systems, including web-based, service-oriented, cloud-native, and enterprise applications.

Software performance testing typically involves generating synthetic workloads (i.e., controlled sequences of operations, requests, or interactions) that emulate the behavior of concurrent users and assessing system behavior by monitoring metrics such as latency, throughput, and resource utilization, under varying load conditions [8]. For instance, a typical synthetic workload may consist of multiple user groups, i.e., independent cohorts of virtual users that all follow the same predefined sequence of actions (e.g., login → browse catalog → add item to cart → checkout). Their executions are staggered over time according to the group's configuration, e.g., by activating one new user every second until the target number of users is reached, so that the load grows gradually rather than all at once. Different user groups can be assigned different scripts, allowing testers to represent distinct categories of real users and to run them concurrently within the same performance test. Under these conditions, performance testing may reveal a range of potential issues, such as sudden increases in response times, throughput degradation, or the emergence of request errors when the system is no longer able to cope with the growing workload.

These activities are typically supported by automated testing solutions, such as Apache JMeter[1], Locust[2], or k6[3], which allow practitioners to define synthetic workloads, execute them, and monitor performance indicators collected from the application under test [24]. Performance testing automation enables scalability and repeatability, but also introduces complexity and challenges in workload modeling, data correlation, and result interpretation [21].

Over the last decades, the research community has extensively explored performance testing methodologies, workload modeling strategies, and tool capabilities [13, 21, 26, 67]. These efforts have allowed practitioners to design more realistic workloads, automate complex testing scenarios, and analyze performance data with increasing sophistication.

Research Gap. Although performance testing research provides extensive guidance on **best design practices**, it offers little consolidated knowledge about the **bad practices** that emerge when these principles are not followed or when they are applied inconsistently.

Understanding the recurring suboptimal practices in performance testing is crucial, as even minor design errors can distort the system's observed behavior and lead to incorrect conclusions. For instance, in the multi–user-group scenario described above, the use of hard-coded or static input data can significantly impact the validity of the results. If all virtual users repeatedly send the same product identifier, the system may respond from cache rather than executing the complete computation or database interaction that real users would trigger. As a result, the test may misleadingly report low response times and stable throughput, thereby masking genuine performance bottlenecks that would otherwise emerge under realistic and varied traffic conditions.

The absence of a systematic understanding of suboptimal performance-testing practices is particularly striking when compared to other areas of software testing. In domains such as unit testing [9, 72] and GUI-level testing [29, 65, 74], researchers have extensively investigated recurring suboptimal patterns, commonly referred to as *"test smells"*, that harm the maintainability, reliability, or realism of tests [2, 35, 48, 61, 70]. These studies have led to well-defined taxonomies of smells [4, 32], dedicated detection techniques [55, 58–60], and actionable guidelines [27, 29, 57] that help practitioners avoid or refactor problematic test designs [47, 69]. In contrast to such well-established knowledge in other testing or production domains [34, 62], where engineering practices have been

[1] https://jmeter.apache.org/
[2] https://locust.io/
[3] https://k6.io/

shown to substantially affect software quality and maintenance outcomes [33], performance testing, despite its inherent complexity and its critical role in assessing system behavior under load, *still lacks a consolidated understanding of the common pitfalls that practitioners encounter in real-world scenarios.*

Contribution.

This paper aims to fill this gap by conducting **the first systematic investigation of bad practices in software performance testing**. We specifically target recurring mistakes, misconceptions, and test smells that arise during the design, implementation, and execution of performance tests and may silently compromise the reliability and interpretability of performance evaluations. Our goal is first to contribute a *taxonomy* of such bad practices, distilled from practitioners' experience and organized to reflect the main phases of performance testing. Secondly, we aim to complement this taxonomy with *insights from practitioners on the severity, prevalence, and practical consequences of these practices*, thus providing a more complete picture of the challenges that affect performance testing in real-world scenarios.

Accordingly, we adopt a two-phase mixed-methods research design [17], beginning with an investigation of developer-oriented sources. In the first phase, we conduct a gray literature review [30] covering 415 practitioner-oriented sources, to extract and organize bad practices into a preliminary taxonomy. In the second phase, we refine this taxonomy through a survey [51] of 22 practitioners and follow-up semi-structured interviews with experienced performance engineers.

This research design deliberately targets software performance testing as a whole, rather than a specific testing type, because many of the practices that undermine the reliability of performance evaluations originate from shared activities, common across different performance testing approaches. By adopting this broader perspective, we aim to capture cross-cutting bad practices that affect performance testing outcomes independently of the particular testing intent.

The main results of the study present an empirically grounded taxonomy of 29 bad practices in software performance testing, spanning the design, implementation, and execution phases, as well as an in-depth characterization of their perceived severity and prevalence in industrial contexts. In addition, the practitioners' insights collected through the survey and interviews shed light on why these issues arise, how they affect the reliability of performance evaluations, and which are considered most harmful or most frequently encountered in real-world projects.

Structure of the Paper. Section 2 introduces the background on performance testing, while Section 3 reviews the relevant literature. Section 4 presents our two-phase research method, and Section 5 reports the resulting taxonomy of bad practices and practitioners' perceptions. Section 6 discusses the implications of our findings. In Section 7, we outline the main threats to validity affecting the study. Finally, Section 8 draws final remarks and outlines future research directions, concluding the paper.

2 Performance Testing

Performance testing represents a fundamental activity for verifying whether a system under test (SUT) behaves as expected under varying workload conditions. At its core, this practice involves generating synthetic workloads and observing the system's responses to detect bottlenecks or load-related failures. Such evaluations are essential to guarantee that end users consistently experience acceptable levels of service quality [22, 75].

The relevance of performance testing is particularly evident in many classes of software systems that operate under concurrent or high-demand conditions, including web-based, service-oriented,

and cloud-based applications [13, 38]. These systems often serve as business-critical platforms and must sustain large numbers of simultaneous users or requests [16]. They are frequently governed by Service Level Agreements (SLAs) that define performance-related constraints, such as maximum response times, throughput targets, or availability requirements. Violations of such constraints may result in contractual penalties, reputational damage, or substantial financial losses. For instance, it has been estimated that a one-second increase in page load time for a large-scale e-commerce platform such as Amazon could translate into losses of up to $1.6 billion per year [14, 28].

Given these stakes, performance testing is not only a technical necessity but also a strategic business concern. In the following sections, we introduce the fundamental concepts and methodologies required to understand the role of performance testing in modern software systems and to contextualize the contributions of this paper.

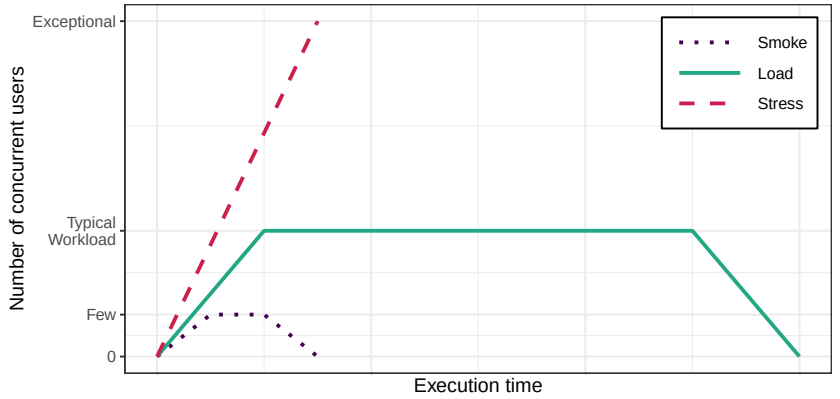

Fig. 1. Exemplified view of performance testing types in terms of user concurrency over execution time

2.1 Performance Testing Types

Performance testing generally encompasses load, stress, and smoke testing [20]. Each of these techniques evaluates system behavior under distinct workload conditions, as illustrated in Figure 1:

Smoke testing applies a lightweight workload to confirm that the system is functioning and to establish baseline performance indicators. Typically, this entails simulating a small number of concurrent users for a brief duration, often less than 30 seconds [10, 43].

Load testing assesses how the system under test behaves under typical, real-world usage scenarios. It reproduces sustained interactions from a defined number of concurrent users, following usage patterns that mirror everyday operational conditions [39, 54].

Stress testing investigates the resilience of a system by driving it beyond its normal operating thresholds [5]. It may involve applying exceptionally high load levels [40] or constraining essential resources such as CPU, memory, or network bandwidth to emulate infrastructure failures [53]. Stress testing is also used to verify whether the software architecture remains stable and responsive under such adverse conditions [19, 26].

While these testing types differ in their objectives and workload intensity, they share common foundations in workload modeling, test design, execution infrastructure, and result analysis. Consequently, several pitfalls may arise independently of the specific testing intent and can affect the trustworthiness of performance evaluations across multiple testing types. For this reason, in this

work, we adopt a holistic view of software performance testing, focusing on cross-cutting bad practices that may compromise performance testing results along the overall testing process.

2.2 Performance Testing Phases

The performance testing process is generally organized into three fundamental phases: *Test Design*, *Test Execution*, and *Test Analysis*. Each phase plays a distinct role in ensuring that the evaluation produces reliable and actionable insights.

2.2.1 Performance Test Design. The *Test Design* phase is widely regarded as the most critical step, since the realism and accuracy of the workload directly determine the validity of the resulting measurements [14]. In this stage, the workload is carefully defined to mirror either expected operational conditions or stress-inducing scenarios. To achieve this, workloads are typically structured into separate *user groups*, each representing a population of concurrent users performing the same session behavior. Several attributes characterize the configuration of a user group:

- **User session**: the ordered sequence of operations, requests, or interactions executed by each user in the group.
- **Number of users**: the total number of concurrent users participating in the group.
- **Initial delay**: the waiting time before the group starts generating load after the test begins.
- **Startup time**: the ramp-up interval during which users are progressively activated until full concurrency is reached. It is typically distributed evenly across all users (e.g., total users divided by ramp-up duration).
- **Hold load time**: the duration in which all concurrent users sustain continuous activity.
- **Shutdown time**: the period for gradually terminating user sessions and ceasing the requests.

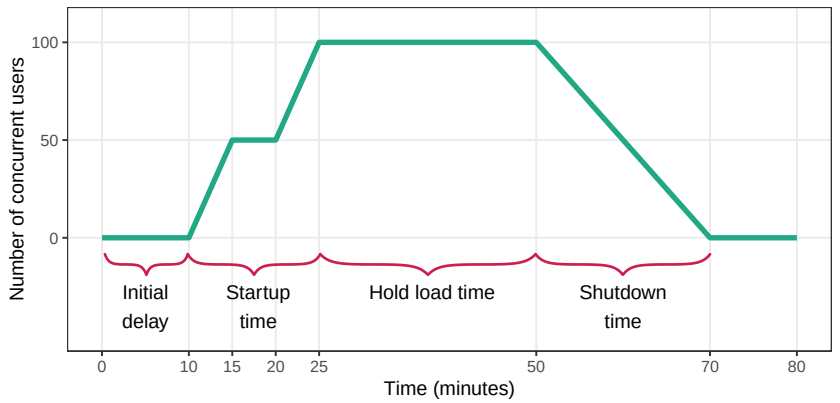

Fig. 2. Example of user group behavior [8]

Figure 2 illustrates an example of user-group behavior. The group begins with an initial 10-minute delay, during which no users are active. A startup phase follows, lasting 15 minutes, during which users ramp up in two steps: first reaching 50 concurrent users (at minute 15), then increasing to 100 at minute 25. The system then sustains this peak load for 25 minutes (hold load time). Finally, a 20-minute shutdown phase gradually reduces the number of active users to zero.

In practice, realistic workloads rarely consist of a single group. Instead, they comprise multiple overlapping user groups, each with distinct behaviors and timing patterns. This layered configuration allows the test to capture complex usage scenarios, such as peak traffic coinciding with

background tasks or resource-intensive operations. By modeling such heterogeneity, performance testing can more accurately reflect the dynamics of real-world environments and expose potential bottlenecks that would remain hidden under simpler workloads [24, 26].

2.2.2 Performance Test Execution. *Test Execution* happens once the workload has been defined. This stage involves converting the workload specification into an executable format, configuring the testing environment, simulating modeled interactions, and collecting key performance indicators such as response times, CPU and memory consumption, and throughput [39].

Although real users could generate workloads, this approach is rarely practical due to its limited scalability and lack of reproducibility. Therefore, performance engineers typically rely on tools such as Apache JMeter and Locust, which automate workload generation by simulating virtual users according to the defined behavioral models. Nevertheless, these tools introduce a two-fold challenge. First, designing workloads that are both realistic and effective often demands advanced programming expertise. Then, they require manual management of data dependencies across operations, which can be error-prone and time-consuming [68].

This challenge is particularly pronounced in modern software systems, where operations frequently depend on dynamic values produced by earlier interactions within the same execution context or session [45, 66]. Since such values vary across executions and concurrent users, it is essential to construct parametric workloads that automatically detect, extract, and propagate such dependencies. Without this capability, simulated interactions risk diverging from real-world behavior, thereby undermining the validity of the performance evaluation.

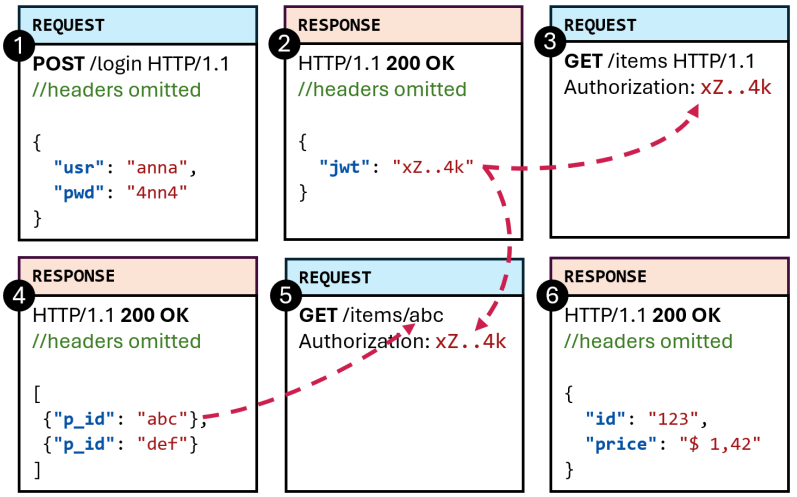

Fig. 3. Example of HTTP response and subsequent requests, with data dependencies highlighted with arrows

Figure 3 illustrates a typical pattern of request-response dependencies in API-based workloads, which commonly arise in web-oriented systems and serve as a representative case of the broader class of dependency patterns encountered in software performance testing. The sequence begins with a login request (Request ❶), in which the user credentials are submitted to the remote backend via a `POST` HTTP request. The corresponding Response ❷ returns an issued JSON Web Token (JWT), a compact bearer token commonly used for securely sharing claims between parties, which must be extracted and reused in subsequent requests, for example, as an `Authorization` header, as done in Requests ❸ and ❺.

Since each authentication attempt results in the issuing of a unique token, these values cannot be hardcoded in automated performance tests. Even repeated logins by the same user typically produce different tokens because JWTs include dynamic claims such as `iat` (issued-at timestamp) and `exp` (expiration time), and may also incorporate nonces or cryptographic signatures to prevent replay attacks. Instead, the test must dynamically capture the token from the login response and propagate it across all dependent requests within the same session. Similar dependencies also arise for other dynamic parameters, such as resource identifiers or session-specific values such as CSRF tokens, which may appear in response bodies and need to be injected into request paths, query strings, or headers. For example, in Request ❺, the request URL includes an item identifier (`abc`) as a path parameter, obtained from the body of the previous Response ❹.

Managing these relationships typically requires manual effort: testers must identify dependencies, implement extraction logic, and configure parameter injection rules [22]. Most performance testing tools lack automated correlation mechanisms, making this process time-consuming and error-prone, possibly yielding unrealistic workloads that fail to reflect actual user behavior [8, 18].

2.2.3 Performance Test Analysis. After test execution is complete, *Test Analysis* focuses on interpreting the collected data to determine whether the system meets the expected performance criteria and to identify potential issues related to load handling. Because performance testing typically produces large volumes of measurement data (e.g., time-series observations of response time, throughput, and resource utilization), manual inspection alone is rarely sufficient. Instead, the analysis process often leverages automated heuristics that can highlight anomalies, such as threshold violations or recurring patterns indicative of performance degradation [39, 44].

3 Related Work

Despite the crucial role of performance testing, to the best of our knowledge, no prior work has systematically investigated guidelines, pitfalls, or bad practices to avoid in this domain. The literature instead mainly focuses on performance testing methodologies, workload characterization, and open challenges, without explicitly addressing common mistakes or practitioner mistakes.

Among the most representative and practically helpful contributions are the works of Pargaonkar [56] and Hassan et al. [14]. Pargaonkar provides a comprehensive overview of performance testing strategies, discussing the choice of performance metrics, the design of realistic usage scenarios, and the emulation of diverse execution environments. Although the guidelines are not explicitly grounded in empirical evidence, they appear to be based primarily on the authors' experience and remain useful for practitioners seeking methodological advice.

Hassan et al. similarly offer an experience-driven perspective, aimed at both practitioners and researchers interested in testing and analyzing large-scale software systems. Their work highlights key considerations across the different phases of performance testing, from workload design to results analysis. For example, they note that while designing realistic loads is often presented as straightforward, in practice it remains a significant challenge [42]. They also emphasize that load testing generates large volumes of data, requiring scalable, efficient analysis to derive actionable insights. Like Pargaonkar, this work provides valuable guidance but does not attempt a systematic identification of common pitfalls or bad practices.

Beyond these experience-based contributions, most performance-testing research focuses on specific aspects, such as workload characterization, often through systematic reviews or surveys that identify open research challenges rather than documenting bad practices. Another recurrent theme is the comparison of workload generation and load testing tools, in which studies evaluate platforms such as Apache JMeter, LoadRunner, NeoLoad, and others based on criteria including usability, script generation, reporting capabilities, and cost. While these works provide valuable

insights into tool selection and highlight the technical trade-offs of workload modeling, they do not explicitly investigate the prevalence and consequences of poor performance-testing practices.

In the context of cloud systems, Shishira et al. [68] identify gaps in the attributes used to characterize workloads and in modeling granularity, noting that richer, application-specific attributes and combined levels of abstraction could yield more realistic results. Curiel et al. [18], focusing on web applications, highlight a broader set of challenges, including capturing workload evolution, simulating realistic network conditions, ensuring rigorous validation, modeling dynamic user interactions, handling non-stationary usage patterns, and reducing the manual effort required for workload scripting. Together, these works point to the complexity of designing realistic workloads and the lack of standardized practices in this area.

Several studies have examined tools such as Apache JMeter, HP LoadRunner, Microsoft Visual Studio (TFS), NeoLoad, WAPT, Soasta CloudTest, LoadStorm, Loadster, HTTPERF, LoadUI, and LoadImpact. For example, Abbas et al. [1] compare four automated load testing tools across dimensions like script generation, plug-in support, reporting, application support, and cost. Similarly, Vandana et al. [12] present a comparative study of Apache JMeter and LoadRunner, concluding that Apache JMeter is more usable due to its simpler UI. Other works take a broader view: Bhatti et al. [7] analyze a wide range of tools and recommend NeoLoad as the most effective for web applications, owing to its visual programming and scriptless design. Rina [64] evaluates NeoLoad, WAPT, and Loadster across different browsers, stressing that comparisons are difficult because tools differ in architecture, supported parameters, and simulation mechanisms. Likewise, Upadhyay [3] and Sufiani [71] focus on attributes such as regression testing capabilities, test documentation, usability, and response times.

To sum up, existing research on performance testing primarily provides methodological guidance, experience-driven recommendations, or tool-centered evaluations, without offering a systematic and empirically grounded characterization of recurring bad practices that may compromise the reliability of performance evaluations. In contrast, the *scientific novelty* of our work lies in explicitly identifying, organizing, and corroborating such bad practices through a mixed-methods study that combines a large-scale gray literature review with practitioner-based empirical insights. In doing so, we deliberately adopt a general perspective on software performance testing, rather than focusing on a specific testing type or application domain, with the aim of capturing cross-cutting pitfalls that arise across shared phases of test design, execution, and analysis. This positioning allows us to complement prior work by shifting the focus from how performance testing should be conducted to how it may fail in practice, thereby providing a novel, practitioner-informed foundation for improving the quality and trustworthiness of performance testing processes.

4 Research Objectives and Research Method

The *goal* of this study is to develop a comprehensive understanding of bad practices in software performance testing with the *purpose* of empirically assessing the practitioners' insights on the matter. The *perspective* is for both researchers and practitioners: the former are interested in empirical foundations for advancing research on test quality, while the latter are interested in actionable insights to recognize, assess, and mitigate common pitfalls that undermine the reliability and efficiency of performance testing. We structured our study around two research questions:

> **RQ1:** *What are the common bad practices in software performance testing?*

> **RQ2:** *How do practitioners perceive these bad practices?*

These research questions guide the construction and validation of a taxonomy of bad practices in software performance testing, which underpins our study. With $\mathbf{RQ}_1$, we aim to provide a mapping of the state of performance testing, offering practitioners clear guidance on what to avoid and establishing a structured reference for subsequent empirical analyses. $\mathbf{RQ}_2$ complements this goal by investigating practitioners' perceptions of the severity and prevalence of each bad practice in real-world performance testing contexts.

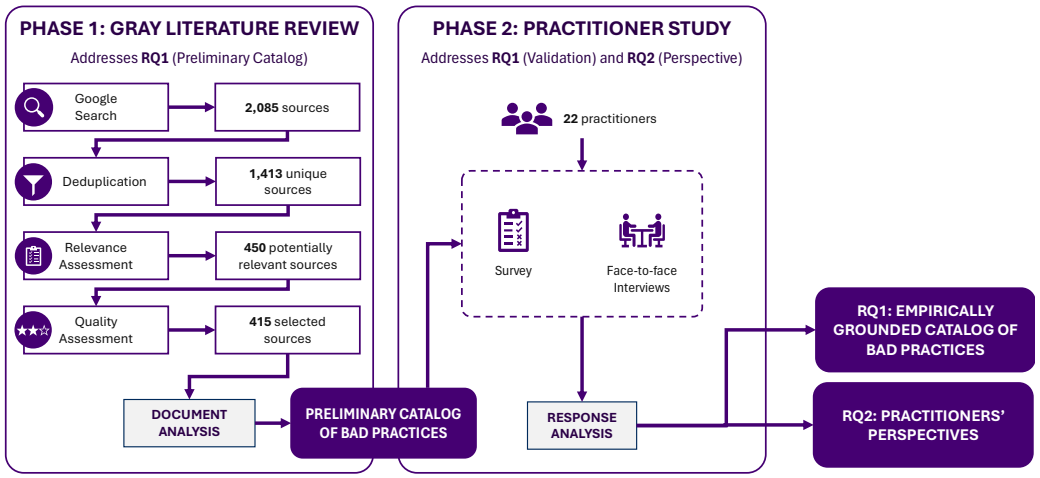

Fig. 4. Overview of the experimental design

From a methodological standpoint, we adopted a *two-phase mixed-method design* [50] that combines qualitative and quantitative analyses to triangulate insights from diverse data sources. As illustrated in Figure 4, the study begins with a *gray literature review* [31], through which we analyze non-academic yet influential sources, e.g., blog posts, industry reports, and tool documentation, to understand what practitioners discuss regarding bad practices in performance testing. We opted for a *gray literature review* because the academic literature has primarily focused on *how* performance tests should be designed and executed, offering limited insight into the challenges, misconceptions, and pitfalls that may arise during performance testing. Therefore, a *gray literature review* enables direct access to this knowledge, making it the most suitable approach for identifying real-world bad practices. The outcome of this phase is a preliminary catalog of bad practices that serves as the foundation for the subsequent phase, rather than final answers. We then conducted a combined study of surveys and face-to-face interviews involving 22 practitioners. This phase validates the preliminary findings, formally answers $\mathbf{RQ}_1$, produces an empirically grounded catalog of bad practices in performance testing, and captures practitioners' perspectives to address $\mathbf{RQ}_2$. In the following, we detail the research method and the analysis performed in each phase.

4.1 Data Collection

This section describes the data collection process from the two phases of our study.

4.1.1 Phase 1: Gray Literature Review. The goal of our gray literature review was to identify practitioners' discussions on common bad practices in software performance testing and to consolidate them into an initial taxonomy. This taxonomy serves as the foundation of our study, providing both researchers and practitioners with a structured overview of recurring pitfalls to avoid.

Inspired by the research methods used in prior empirical studies on software testing practices [31, 63], we adopted a systematic procedure for selecting and analyzing sources on the topic. Phase 1, reported in Figure 4, illustrates the overall process of the gray literature review, which was organized into three main steps: (i) Google search, (ii) document selection, and (iii) document analysis. Each step is described in detail below.

Search Strategy. We retrieved sources for our gray literature review by querying Google Search programmatically through a *Search Engine Results Page* (SERP) service. This allowed us to execute a predefined set of queries in a consistent way and to automatically record the returned metadata for screening and analysis. Since search requests were issued through the API service rather than via an interactive browser, no anonymous or incognito browsing mode was required. The search was configured to retrieve English-language results (`hl=en`) from the U.S. Google index (`gl=us`). We selected the U.S. index to obtain a broad and consistent English-language retrieval space, as much practitioner-oriented material on performance testing is published for a global audience through U.S.-oriented or internationally indexed web sources.

The search strategy combined two query families: **concept-led** and **tool-led** queries. The concept-led queries targeted practitioner discussions framed around testing approaches. We used the following concept anchors: *performance testing*, *load testing*, *stress testing*, *spike testing*, *soak testing*, and *smoke testing*. For each concept anchor, we issued a query based on the template reported in Listing 1.

Listing 1. Concept-led query template

```
"<concept>" AND ("best practices" OR "bad practices" OR "tips" OR "recommendations"
              OR "guidelines" OR "pitfalls" OR "anti-patterns" OR "lessons learned")
```

The tool-led queries targeted practitioner discussions framed around commonly used performance-testing tools. We used the following tool anchors: *JMeter*, *LoadUI*, *WAPT*, *BlazeMeter*, *Gatling*, *Locust*, *k6*, and *ApacheBench*[4]. For each anchor term, we combined it with a bundle of practice-oriented expressions commonly used in gray literature, namely *best practices*, *bad practices*, *tips*, *recommendations*, *guidelines*, *pitfalls*, *anti-patterns*, and *lessons learned*. For tool-led queries, we additionally constrained the search with the context terms *performance*, *load*, *stress*, *spike*, and *soak* to reduce off-topic results. At the end, we issued a query based on the template reported in Listings 2.

Listing 2. Tool-led query template

```
"<tool>" AND ("best practices" OR "bad practices" OR "tips" OR "recommendations"
          OR "guidelines" OR "pitfalls" OR "anti-patterns" OR "lessons learned")
        AND ("performance" OR "load" OR "stress" OR "spike" OR "soak").
```

This yielded 14 queries overall. The search was executed automatically via a script, ensuring a consistent, reproducible process for issuing queries, collecting results, and recording metadata. The script is available in our online appendix [23]. For each query, we retrieved up to 15 pages of results with 10 results per page, as recommended in prior work [63], since no relevant documents were typically found beyond that point, for a maximum of 2,100 raw results before deduplication.

[4]The set of tool anchors was informed by the authors' domain knowledge of performance testing, by preliminary exploratory searches aimed at identifying widely used tools in practitioner discourse, and by popular community-curated resources. In particular, we consulted publicly available GitHub resources with substantial community uptake (e.g., repositories with more than 300 stars), such as https://github.com/ZoranPandovski/awesome-testing-tools.

Table 1. Number of sources retrieved per search query

		Gray Literature Review Sources		
Query type	**Query anchor**	**Retrieved**	**After deduplication**	**Selected**
Concept-led	performance testing	147	118	102
Concept-led	load testing	150	109	90
Concept-led	smoke testing	147	123	24
Concept-led	stress testing	149	116	23
Concept-led	spike testing	150	81	15
Concept-led	soak testing	147	94	13
Tool-led	JMeter	148	134	69
Tool-led	BlazeMeter	150	100	21
Tool-led	k6	148	99	17
Tool-led	Locust	149	96	13
Tool-led	Gatling	150	92	13
Tool-led	LoadUI	150	94	11
Tool-led	WAPT	150	82	4
Tool-led	ApacheBench	150	75	0
Total		**2,085**	**1,413**	**415**

The process resulted in **2,085** sources. The detail on the number of retrieved sources for each query is reported in Table 1.

Source Deduplication. Since the same resource may be returned by multiple queries, we performed a deduplication step before applying any substantive screening criterion. Deduplication was based on URL canonicalization, so that URLs referring to the same underlying resource were normalized and counted only once. In cases where the same canonicalized URL appeared in the results of multiple queries, the source was assigned to the query for which it appeared earliest in the ranked results, operationalized as the smallest result page number. This ensured that each source was counted only once and linked to a single query for reporting purposes. Overall, this step reduced the risk of over-representing sources that were highly ranked across several queries and ensured that the subsequent stages operated on a set of unique candidate resources. After deduplication, **1,413** unique sources were retained. The number of retained unique sources for each query is reported in Table 1.

Screening. Because Google Search is intentionally inclusive, the retrieved corpus inevitably contained a substantial amount of noise and artifacts outside the scope of this study. We therefore applied a screening step to improve the reliability and reproducibility of the selection process before the subsequent quality assessment.

As an initial filtering action, we removed sources that were structurally unsuitable for this review, even if they had been retrieved by the search engine. In particular, we excluded Wikipedia pages, pages from general news outlets, short-form video platforms (e.g., TikTok, Instagram), and similar artifacts that are unlikely to provide stable, firsthand practitioner discussions of software performance testing bad practices. This initial filtering step was performed automatically, by matching the retrieved URLs against a blacklist of domains associated with structurally unsuitable source types.

The remaining sources were then assessed for topical relevance. This step was intentionally conservative: our goal was not to discard every borderline case, but rather to exclude only sources that were clearly outside the scope of the study. To operationalize this decision, we applied one inclusion criterion and a set of exclusion criteria. The inclusion criterion required that a source explicitly addressed bad practices in software performance testing. Therefore, a source was considered potentially relevant if it satisfied the inclusion criterion and did not meet any of the exclusion

criteria. As for the exclusion criteria, we employed the same rules adopted in prior work [63] to ensure methodological consistency with established gray literature review practices. Specifically, a document was excluded if it met even one of the following criteria: (E1) it provided only guidelines for manual testing or generic tool usage without addressing performance testing practices, to maintain focus on the target domain; (E2) it was a video or a book, which are challenging to analyze systematically and may hinder reproducibility; and (E3) its access was restricted by a paywall, preventing full inspection and traceability of the extracted information. The topical relevance assessment was conducted by the first two authors. Details on the annotation procedure and inter-rater agreement are in Section 4.2.1. After the screening step, only **450** potentially relevant sources were retained.

Quality Assessment. Relevance alone does not guarantee trustworthiness. Indeed, given the heterogeneous nature of gray literature, a key concern is the potential inclusion of low-quality sources that may affect the reliability of the results. To mitigate this risk, we introduced a quality assessment step prior to data extraction, following the guidelines proposed by Garousi et al. [31]. The goal of this step was to evaluate the credibility, quality, and thoroughness of each retrieved resource, and to exclude those that did not provide sufficient reliability for inclusion in the study. As such, starting from the quality assessment checklist proposed by Garousi et al. [31], we defined a checklist composed of the following questions:

- **Q1.** *Is the publication organization reputable?*
- **Q2.** *Is the author known?*
- **Q3.** *Does the author have expertise in the area?*

Each question was evaluated using a three-level scale: *"Yes"*, *"Partially"*, and *"No"*. To support a systematic assessment, we associated a numeric value to each label: *"Yes"* was assigned a value of 1, *"Partially"* a value of 0.5, and *"No"* a value of 0. The overall quality score for each source was computed by summing the scores of the three questions. Sources with a total score lower than 1.5 were excluded from the analysis. The checklist and the associated threshold were intentionally defined as a minimum credibility safeguard, with the purpose of excluding only sources for which insufficient evidence of outlet or author trustworthiness could be established, while avoiding an unnecessarily restrictive filter that would risk discarding relevant practitioner-oriented evidence.

The quality assessment was conducted by the first two authors (hereafter referred to as the inspectors), who manually reviewed each retrieved resource to answer the checklist questions. To assess the reputation of the publication organization (**Q1**), the inspectors first considered whether the organization was publicly known and recognized in the context of software development and testing. In cases where the organization was not immediately recognizable, the inspectors performed an online search to collect additional contextual information, including the organization's mission, its connection to software engineering activities, its online and social media presence (e.g., existence of a LinkedIn page), the number of followers, and other relevant indicators such as company size, geographical presence, and overall visibility.

To assess the author-related criteria (**Q2** and **Q3**), the inspectors followed a similar process. First, they verified whether the resource explicitly reported an author. Then, they collected information about the author by searching online sources, such as personal websites, professional profiles, and company pages. The evaluation considered aspects such as the author's current or previous roles, their involvement in software testing or performance engineering activities, and any evidence of relevant expertise. If sufficient evidence was found linking the author to expertise in the domain, the source was considered reliable for inclusion.

Also in this case, the manual quality assessment was preceded by a calibration phase independently conducted by both inspectors on a subset of 50 candidate sources. The goal of this phase was to align the interpretation of the checklist items and reduce possible ambiguities in their application.

Subsequently, the set of remaining candidate resources was equally divided between the two inspectors. In cases where the assessment was not straightforward, the inspectors engaged in discussion to reach a shared decision. If agreement could not be reached, a third author was available to support the decision-making process; however, in practice, all disagreements were resolved through discussion between the first two inspectors. As a result of this quality assessment process, a subset of **35** initially retrieved resources was excluded due to insufficient credibility or lack of supporting evidence. The distribution of selected sources across queries is reported in Table 1. The relatively limited number of exclusions at the quality-assessment stage indicates that most sources judged potentially relevant also provided sufficient authorship or outlet credibility to support inclusion. For transparency and replicability, the replication package [23] includes the detailed quality assessment scores assigned to each source.

4.1.2 Phase 2: Survey Study and Semi-structured Interviews. This phase aims to empirically ground and refine the initial taxonomy of bad practices in performance testing by incorporating practitioner feedback. In this context, grounding refers to assessing practitioners' level of agreement with the taxonomy and collecting their perceptions of the severity and prevalence of the identified bad practices, rather than establishing a definitive or community-wide validation.

We followed the guidelines by Kitchenham and Pleeger [41] to design a survey that strikes a balance between conciseness and effectiveness in addressing our research questions. The various answer options are provided in the online appendix [23]. We provided participants with an informed consent document and deliberately avoided collecting personal information, including gender, age, and email addresses. To ensure data quality, we incorporated attention checks to identify and subsequently discard responses from participants who provided careless answers.

Survey Design: Introductory Section. We started including a brief introductory text where we presented ourselves and the overall goals of the empirical study. In doing so, we reported that the goal of our investigation is to better understand common pitfalls and harmful practices in software performance testing. We also added that we were interested in practices considered ineffective or counterproductive during the design, implementation, or execution of performance tests to inform participants of our expectations, the expertise required to participate, and the research objectives.

Furthermore, we provided information on the survey length, which was estimated at 10 minutes and subsequently empirically assessed in a pilot study (details below). We also explained that participation was voluntary, that participants could withdraw from the survey at any time, and that all responses would be anonymous to preserve participants' privacy. In the introductory part of the survey, we also requested explicit consent from participants to use the collected data for research purposes.

Participant's Background. The first section of the survey was related to the participants' background, which was required to (i) characterize the sample of developers answering the survey and (ii) understand whether and which answers should have been removed because of the poor experience/expertise of some participants. Table 2 summarizes the background questions asked in our study. As shown, we requested information on years of experience in software and performance testing; their current role (e.g., Software Tester, QA Engineer, Researcher); whether they primarily developed in open-source or other contexts (e.g., industry); and their familiarity with performance testing concepts. Optionally, for participants from an industrial environment, we also asked whether they routinely developed performance tests in their work context. Finally, we asked

Table 2. List of questions for the background section in the survey with the type of response provided. The asterisk next to some questions indicates that an answer is required.

Section 1: Participant's Background		Type
#1	What is the highest level of education you have completed?*	Multiple choice (High School or equivalent, Bachelor's Degree, Master's degree, Ph.D., Other to specify)
#2	What is your current role?*	Checkbox (Software Developer/Engineer, Software Tester/QA Engineer, Researcher/Professor, Other to specify)
#3	Are you currently working in:*	Multiple choice (Industry, Academia, Both, Other to specify)
#4	How many years of experience do you have in the following areas?*	Multiple choice grid where each row represents a dimension (e.g., Software Engineering, Software Testing, Performance Testing) and each column represents an ordinal category (years of experience), with respondents allowed to select exactly one option per row.
#5	How would you rate your familiarity with performance testing concepts?*	Multiple choice (No experience, Beginner, Intermediate, Advanced, Expert)
#6	If you work in the industry, how often is system performance testing carried out for software projects developed within your company or team?	5-point Likert scale from Never to Very Often
#7	Which of the following system performance testing tools have you used? Select all that apply.*	Checkbox (Apache JMeter, Locust, Gatling, k6, Loadrunner, None, Other to specify)

for their familiarity with performance testing tools (e.g., Apache JMeter, Locust, k6). Participants with less than one year of experience and no expertise with any performance testing tool were excluded from completing the rest of the survey. This process ensured that the collected responses came from practitioners with sufficient domain knowledge.

Catalog Empirical Grounding. Participants then moved to the core of the survey, in which we asked about the bad practices in performance testing reported in the taxonomy. More particularly, we structured the survey around three development phases:

Test Design: that explores design flaws and modeling pitfalls in software performance testing;
Test Execution: that focuses on technical and structural flaws found during the implementation and execution of software performance tests;
Test Analysis: that addresses flaws related to the collection and analysis of their metrics.

The structure of the questionnaire is reported in Tables 3, 4, and 5. The questionnaire items refer to the set of bad practices identified in Phase I. At this stage, they are presented as survey items to be evaluated by participants; the full taxonomy and detailed description of each bad practice are introduced in Section 5. Each section started with a short introduction outlining the overall goal of that set of questions. We then asked to evaluate each bad practice along three dimensions:

Agreement: the extent to which the participant agrees that this is a bad practice;
Severity: the extent to which the consequences of this practice are considered severe if it occurs;
Prevalence: the extent to which the practice was observed in real-world performance testing.

Each dimension was assessed using a 5-point Likert scale ranging from *"Strongly Disagree / Not Severe at All / Never"* to *"Strongly Agree / Extremely Severe / Very Frequently"*.

A potential concern we explicitly accounted for relates to the possible misinterpretation of the bad practices presented to participants. In this respect, it is worth noting that most of the identified bad practices concern general performance testing design, implementation, or execution behaviors (e.g., unrealistic user behavior, insufficient test planning, ignoring think times), rather than highly specialized technical mechanisms. This reduces the likelihood of misunderstanding, as participants were asked to evaluate practices in realistic performance testing contexts rather than in narrow or tool-specific scenarios. To explicitly mitigate this risk, each bad practice was presented not only by its label but also through a short explanatory description aimed at clarifying

Table 3. List of questions for the test design issues section in the survey with the type of response provided. The asterisk next to some questions indicates that an answer is required.

Section 2: Test Design Issues		Type
#8	Unrealistic user behavior: Simulating users with unrealistic behavior leads to misleading results, such as unnatural or one-request sequences.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#9	Single scenario focus: on the one hand, testing only the happy path without stress, soak, or failure scenarios limits the test's usefulness; on the other hand, test files with no modularization or reuse, for example, the same single request repeated everywhere, do not reflect a real-world scenario and increase maintenance costs.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#10	Hardcoded or static inputs: Not varying input data causes caching, making the system appear faster than it is under realistic conditions; for example, CSV datasets with a single value, not using a Random Variable, or using the same username/password across all simulated users.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#11	Ignoring think times: Sending requests as fast as possible without delays between actions. This overloads the system unrealistically, as real users do not hammer endpoints nonstop.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#12	Too short or too long test duration: Tests that run too short may not trigger memory leaks or long-term performance problems; too long, and you risk resource overuse without added value.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#13	Improper Ramp-Up and Cool-Down: Abrupt load increases or drops without warm-up/cool-down phases distort performance metrics; for example, in JMeter, 0 ramp-up seconds for 1000 users; in Gatling, no `rampUsers` or `nothingfor()` to simulate natural traffic growth.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#14	Poor test planning, such as lacking clear performance objectives or failing to define key metrics, such as response time and throughput, reduces the effectiveness of performance testing.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#15	Confusing load testing with other types of testing: Mixing up load testing with stress, soak, or spike testing is a common mistake. Each type serves a distinct purpose—load testing assesses performance under expected traffic, while stress, soak, and spike tests are designed to evaluate behavior under extreme or prolonged conditions.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#16	Outdated or unmaintained tests: Are tests kept up-to-date with application changes? Are they regularly reviewed and refactored?*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).

its meaning in performance testing contexts. These descriptions were intentionally written in clear and accessible language, accompanied by brief explanations or examples to reduce ambiguity and ensure a consistent interpretation across participants. For instance, the practice *"Confusing load testing with other types of testing"*, which refers to the misinterpretation of different performance testing types, as detailed in Section 5, was presented together with the following clarification: *"Mixing up load testing with stress, soak, or spike testing is a common mistake. Each type serves a distinct purpose—load testing assesses performance under expected traffic, while stress, soak, and spike tests are designed to evaluate behavior under extreme or prolonged conditions"*. Similar explanatory descriptions were provided for all practices that could potentially be ambiguous.

Survey Closure. In the final section of the survey (see Table 6), participants were first given the opportunity to provide additional feedback through an optional open-ended question. Specifically, they were explicitly invited to suggest potentially missing bad practices, express disagreement with the proposed taxonomy, or report relevant experiences not captured by the predefined items.

Table 4. List of questions for the test implementation and execution issues section in the survey with the type of response provided. The asterisk next to some questions indicates that an answer is required.

	Section 3: Test Implementation and Execution Flaws	Type
#17	Testing in non-production-like environments: Running tests on underpowered machines or without realistic network conditions can lead to misleading results. Example: localhost, mock services.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#18	No proper cleanup between test runs; for example, leftover test data affects subsequent runs.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#19	Lack of modularity/reusability: (i) Duplication of logic: Same logic repeated across test cases instead of using functions/classes. (ii) No comments or documentation: Hard to understand what the test is doing or why certain values are chosen.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#20	Poor test data management: (i) No variation in test data: All the users hit the same URLs or send the same payload—this is not realistic and can result in caching effects. (ii) Poor parameterization: Using static endpoints, ports, and credentials instead of configurable variables. Example: JMeter: using ${__P(..)}.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#21	Using recorded scripts without refinement: Performance test scripts generated through recording (e.g., with JMeter or LoadRunner) are often used as-is, without cleaning up unnecessary steps, handling dynamic values (e.g., session tokens), or adding realistic think times. This can result in unrealistic and noisy load simulations.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#22	Neglecting realistic network conditions: Performance tests are often executed in ideal lab environments (e.g., over LAN or high-speed connections), which do not reflect real user scenarios like 3G or high-latency networks. Failing to simulate network constraints can lead to missing critical performance bottlenecks.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#23	Poor Naming Conventions: Naming samplers as "HTTP Request 1", "HTTP Request 2" makes analysis difficult in reports.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#24	Over-Mocking: Mocking dependencies too much leads to unrealistic tests.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#25	Reusing Connections or Sessions Incorrectly: Sharing sessions/cookies across virtual users may not represent real-world load and cause resource contention issues. Example: Opening new HTTP clients, sessions, or DB connections in test loops.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#26	Not Integrating with CI/CD: Performance testing done ad hoc, rather than as part of automated pipelines, makes regression detection reactive rather than proactive.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).

This open-ended component was designed to partially address the risk of incompleteness in the taxonomy by allowing respondents to contribute insights beyond the predefined categories.

After this section, participants were optionally asked to provide their email addresses to participate in a future follow-up semi-structured interview.

Survey Validation. Before disseminating the survey, we evaluated both its quality and the time required to complete it. While longer surveys may provide richer insights, excessive length can discourage participation [6]. To address this concern, after defining the first version of the survey, we conducted a pilot study [52].

The pilot was conducted by the fourth and fifth authors of this paper, both of whom have more than 2 years of experience in performance testing. Significantly, they had not been involved in

Table 5. List of questions for the test analysis issues section in the survey with the type of response provided. The asterisk next to some questions indicates that an answer is required.

	Section 4: Test Analysis	Type
#27	Ignoring Warm-up Phases: Measuring performance from the first request without giving the system time to warm up. Example: JIT compilation and caching result in artificially bad metrics.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#28	Lack of Test Repeatability: Not isolating variables like caching, background tasks, or external dependencies makes tests non-deterministic and hard to reproduce*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#29	Use Rigid Assertions: Hardcoded assertions (e.g., expecting a fixed response time of 200ms). It's better to use relative assertions (e.g., Gatling: `responseTime.percentile(90).lt(500)`).*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#30	No Assertions or Response Validation: Measuring performance without checking if the application responded correctly. Example: in Gatling: Not using .check(status.is(200)) or other content checks; in JMeter: Omitting Response Assertion elements.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#31	Overusing Listeners/Logging: Enabling multiple listeners (e.g., View Results Tree) during load tests. Consumes excessive memory and CPU, distorting performance metrics.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#32	Lack of Baseline Comparisons: Running performance tests without historical comparisons or benchmarks makes it hard to detect regressions.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#33	Skipping Test Validation: Running a 1-hour test without verifying if the first 5 min failed.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#34	High thread count on underpowered machines: Load is generated from the same machine JMeter runs on, which might bottleneck the load generator rather than the system under test.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#35	Ignoring Test Result Variability: Taking one test run as the final truth.*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).
#36	Inadequate Resource Monitoring: (i) Not monitoring system resources (CPU, memory, disk, network) (do not ignore system metrics). (ii) Not tracking performance metrics (response times, throughput, errors etc.).*	A multiple-choice grid was employed to evaluate three dimensions: Agreement (the extent to which the practice is perceived as a bad practice), Severity (the magnitude of its consequences), and Prevalence (the frequency with which the practice is encountered).

Table 6. List of questions for the survey closure section in the survey, with the type of response provided. The asterisk next to some questions indicates that an answer is required.

	Section 5: Survey Closure	Type
#37	We would love to hear from you if you would like to share additional insights, discuss any of the bad practices mentioned, or even suggest others that were not included in the survey.	Paragraph
#38	If you're open to being contacted for follow-up, please leave your email address.	Paragraph

the gray literature review or the construction of the taxonomy, ensuring that their feedback was unbiased with respect to the survey content. The two experts were provided with the survey link, clear instructions, and a text document in which to record their feedback on the clarity, structure,

and overall flow of the survey. They were also asked to track the time required to complete the survey, to obtain a realistic estimate of its length.

They completed the survey in approximately 10 minutes and returned their detailed feedback to the first author within one week. Their observations highlighted the need for minor improvements in the clarity of certain survey items and in the wording of the introductory texts. They also suggested minor adjustments to the order of certain questions to improve flow.

The first two authors jointly analyzed this feedback and revised the survey accordingly. A final confirmation with the two pilots verified that all their concerns had been addressed before launching the survey to the target population.

Ethical Considerations. In our universities, it is not yet mandatory to seek approval from an Ethical Review Board when releasing surveys. Nevertheless, when designing the survey, we mitigated many potential ethical and privacy concerns [36]. We ensured participants' privacy by collecting anonymous responses. Participants voluntarily provided their email addresses to receive a summary of the results and to be contacted for future follow-up semi-structured interviews. When recruiting practitioners, we stated the goal of the survey study and explicitly reported that the responses would be used solely for research purposes and would not involve the publication of sensitive data. Finally, we clarified that completed surveys would eventually be made public, thereby preserving participants' privacy.

Survey Recruitment and Dissemination. The survey was launched at the end of April 2025 and specifically targeted practitioners with experience in software performance testing. To reach this audience, we prepared a poster and shared it on LinkedIn, distributed it at three international software engineering conferences,[5] and, with the support of industrial partners in our contact network, disseminated it among affiliated companies operating in the IT sector. While we acknowledge that social networks are not typically the primary source of academic data, this broad dissemination strategy was necessary to reach our specific target audience. Indeed, recruiting specialists in performance testing is inherently challenging, as this practice is still not widely adopted in industrial settings [26].

Table 7. Example of questions about face-to-face interview participants' background on performance testing

Interview Questions
Can you tell me about your role and how it relates to system performance testing?
How long have you been involved in performance testing?
In which types of systems or domains have you applied performance testing (e.g., web apps, cloud systems, embedded systems)?
How often is performance testing applied in your projects?
At what stage of development do you typically run performance tests?

Semi-structured Interview Design. As the final question of the survey, we asked practitioners whether they would be willing to participate in a follow-up semi-structured interview. Three respondents expressed interest; therefore, we designed an interview guide to explore further the themes that emerged from the survey. All participants (i) have more than three years of experience in performance testing, (ii) work in teams dedicated to software quality assurance, and (iii) are employed in companies with a strong reputation and visibility. The semi-structured interviews

[5]Specifically, the 47th IEEE/ACM International Conference on Software Engineering (ICSE 2025), the 22nd International Conference on Mining Software Repositories (MSR 2025), and the 51st Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA 2025).

were designed in accordance with established guidelines [37, 46]. Their structure mirrored that of the previously described survey. Specifically, we began by focusing on background information, participants' roles in performance testing, their industrial experience, and the contexts in which they had applied these practices. The list of questions that have guided the conversation is reported in Table 7. The second part of the interview followed the three performance-testing phases (design, execution, and analysis) previously described and was structured around the taxonomy identified in the gray literature review. In particular, we shared the same list of bad practices identified in the survey, presented sequentially. For each bad practice, we asked participants to reflect on whether it could indeed be considered a bad practice, to share any relevant experiences, and to evaluate its criticality in real-world contexts. In addition, participants were explicitly encouraged to reflect on the completeness of the taxonomy and to suggest any additional bad practices they considered missing. This was facilitated by the semi-structured nature of the interviews, which allowed participants to elaborate beyond the predefined questions. No interviewee proposed additional distinct bad practices beyond those already captured in the taxonomy.

The interviews were conducted remotely via Microsoft Teams in July 2025 and March 2026, with each participant interviewed individually and moderated by the first author. With the interviewees' consent, all sessions were audio-recorded and subsequently transcribed using the transcription functionality provided by Microsoft Clipchamp, with manual corrections applied by the first author to ensure accuracy. The resulting transcripts were then jointly analyzed by the first and second authors to extract insights from the participants' responses.

It is worth noting that, before conducting the interviews, we conducted a small pilot study to validate the interview guide with respect to clarity, structure, and duration. The pilot involved two simulated interviews conducted by the second and third authors with the first author, using the same format as the actual study. Based on the pilot, we confirmed that the interview duration (approximately one hour) was appropriate and made minor refinements to the wording and ordering of questions, which were incorporated into the final guide.

4.2 Data Analysis

This section first provides an overview of how the data have been analyzed. Then, it provides an overview of the two distinct data sources of our analysis: gray literature and a survey employing semi-structured interviews. Both sources help address the two research questions (**RQs**). Specifically, the gray literature enables us to construct a preliminary catalog of bad practices, while the combined survey and interview phase serves a dual purpose: it empirically grounds the catalog and provides a definitive answer to $\mathbf{RQ}_1$; it also gathers the empirical evidence required to answer $\mathbf{RQ}_2$.

4.2.1 Overview of the Gray Literature Review.

Screening Reliability. The screening of the retrieved documents was conducted through a multi-stage process involving multiple authors, with the goal of reducing subjective bias and improving the reliability and reproducibility of the selection procedure. After collecting the 1,413 unique entries of the initial corpus, the first two authors jointly screened a pilot sample of 50 items to calibrate their understanding and application of the inclusion and exclusion criteria. This calibration phase allowed the two involved inspectors to identify ambiguities in the criteria and establish a shared interpretation before proceeding with independent screening.

Subsequently, both authors independently screened a subset corresponding to 10% of the corpus. Disagreements were analyzed and resolved with the support of the third author, ensuring an external adjudication mechanism. On this subset, we computed Cohen's κ [15], obtaining a value of 0.76, which indicates a *substantial* level of agreement [49] and suggests that the criteria were consistently applied across reviewers. After discussing disagreements and refining the screening guidelines,

the remaining items were divided between the first two authors and screened independently. As a further validation step, a cross-check of the screening decisions was performed on a subset of the screened items, resolving conflicts through discussion. This process yielded a Cohen's κ of 0.84, indicating almost *perfect* agreement and providing additional evidence of the stability and robustness of the screening process. The filtering process and the subsequent quality assessment ultimately resulted in a final set of 415 primary documents.

Document Analysis and Taxonomy Construction. From the set of 415 sources identified in the gray literature review, the first and second authors, both with over three years of experience in performance testing, independently reviewed and analyzed the candidate documents in detail to conduct the data extraction. For each document, the reviewers systematically extracted and recorded all identified bad practices, which were then organized into a structured taxonomy. Each bad practice was explicitly linked to the corresponding source document(s) to ensure full traceability. Notably, no predefined taxonomy was imposed; instead, the classification was derived inductively from the analyzed sources. To assess the consistency of the extraction process, we computed inter-rater agreement on a shared subset of the data of 10% of the data. We measured Cohen's κ [15] at the document level, i.e., by evaluating whether a given bad practice was identified in a given source. The resulting κ value was 0.78, indicating a *substantial* level of agreement [49] and suggesting a consistent application of the extraction procedure across reviewers.

At the end of the independent analysis, the reviewers produced two separate taxonomies of bad practices. These taxonomies were then merged into a unified version. All disagreements were systematically reviewed and discussed; when consensus could not be reached, the third author, also with more than three years of experience in performance testing, acted as an adjudicator to make the final decision. This process resulted in a consolidated and agreed-upon set of bad practices, which formed the basis of the final taxonomy.

Corpus Characterization. To provide a clearer descriptive overview of the final gray literature corpus, we further characterized the selected sources in terms of source type and document size. Specifically, each source was classified according to its origin and content into one of nine categories: technical blog, general web content, tutorial or guide, community Q&A or forum, project documentation, vendor or company content, professional network, industry report, and code repository. Table 8 reports the distribution of the 415 selected sources across these categories together with their size statistics, measured in number of words. Overall, the corpus is dominated by technical blogs (194 sources, 46.7%) and general web content (86, 20.7%), followed by tutorials or guides (38, 9.2%) and community Q&A or forum discussions (33, 8.0%). This distribution indicates that the preliminary taxonomy is grounded primarily in practitioner-oriented web material, while still incorporating more structured sources such as project documentation, vendor content, and industry reports. In terms of size, the corpus includes both relatively concise discussion-oriented sources, such as community Q&A entries (median 1,066 words), and longer, more structured documents, such as tutorials or guides (median 2,497 words) and industry reports (median 3,195 words), thus providing a heterogeneous evidence base for the construction of the preliminary taxonomy. We note that document size was measured automatically as the number of words in the textual content extracted from the selected source.

To improve transparency, we make this information available in the replication package [23], allowing readers to assess the distribution and characteristics of the sources. At the same time, we remain cautious in interpreting these aspects, as differences in source type and size do not directly translate into differences in the importance or validity of the reported practices. Overall, these additions provide a more comprehensive characterization of the gray literature corpus, while maintaining a balanced interpretation of the results.

Table 8. Selected sources by source type

Source type	# Docs	% Corpus	Median Words	IQR words
Technical blog	194	46.7%	2,150	1,838
General web content	86	20.7%	1,646	1,620
Tutorial or guide	38	9.2%	2,497	1,497
Community qna or forum	33	8.0%	1,066	972
Project documentation	30	7.2%	1,237	1,084
Vendor or company content	18	4.3%	2,132	1,272
Professional network	10	2.4%	1,888	734
Industry report	4	1.0%	3,195	1,550
Code repository	2	0.5%	2,100	1,196

4.2.2 Background of Survey and Interview Participants. As for the survey study, we collected 22 responses in total. Following the pre-screening criteria described earlier, no response was excluded for insufficient expertise, keeping all 22 valid participants for our study.

While we acknowledge that the sample size may pose a potential threat, the validity of smaller, focused samples in specialized populations is well recognized in Software Engineering research. For instance, recent high-impact work [11] reported significant findings from a sample of seven participants. Perhaps more importantly, identifying and recruiting software performance testers is inherently challenging, as they constitute a relatively small and specialized subset of practitioners. Performance testing roles often require a combination of advanced technical skills, domain knowledge, and sustained industrial experience. They are frequently embedded within broader quality-assurance teams rather than held as standalone positions. As a result, practitioners with substantial and dedicated expertise in performance testing are less prevalent and harder to reach than those in more general software engineering roles. According to these observations, our engagement with 22 practitioners may still be considered a sufficiently reliable and meaningful empirical basis for drawing insights into current performance testing practices.

From the 22 responses, we analyzed the closed-ended answers. Most questions were formulated to allow participants to express their opinions using a Likert scale with varying ranges. These answers were analyzed using statistical methods, including the construction of bar charts to show the distribution of the data.

Table 9 shows the background of the 22 participants considered for the survey. The majority of these practitioners work in industry, with most holding a Bachelor's or Master's degree, and a smaller portion holding a Ph.D. They primarily serve as software developers or engineers, alongside testers and researchers. Experience levels vary across software engineering, software testing, and performance testing. While participants report heterogeneous levels of expertise and some indicate familiarity with performance testing concepts, the overall years-of-experience distribution suggests that the sample is skewed toward early-career practitioners. Nevertheless, their perspectives are informed by practical engagement with performance evaluation activities. Despite this experience, the survey revealed that performance testing is often skipped or performed infrequently in their industrial projects, with only a minority of participants reporting widespread testing practices. This distribution reflects the way performance testing activities are often carried out in practice, where such tasks might be performed by software developers or QA engineers, rather than exclusively by dedicated performance testing specialists. As such, the survey captures perspectives from practitioners with hands-on experience in performance testing activities, although not necessarily characterized by long-term or specialized roles in this domain. Apache JMeter is the most commonly used tool, reflecting its prevalence in industry practice, followed by Locust and other tools such as LoadRunner, K6, and Gatling.

Table 9. Survey responses about education, role, experience, and performance testing

Question	Response Options	Count (%)	Impact (visual)
What is the highest level of education you have completed?	High School or equivalent	0 (0%)	
	Bachelor's degree	8 (36%)	
	Master's degree	9 (41%)	
	Ph.D.	5 (23%)	
	Other	0 (0%)	
What is your current role?	Software Developer / Engineer	13 (59%)	
	Software Tester / QA Engineer	4 (18%)	
	Researcher / Professor	3 (14%)	
	Student	4 (18%)	
	Other	1 (5%)	
Are you currently working in:	Industry	18 (82%)	
	Academia	3 (14%)	
	Both	1 (5%)	
Years of experience in Software Engineering	0–1	1 (5%)	
	2–5	14 (64%)	
	6–10	7 (32%)	
	10+	0 (0%)	
Years of experience in Software Testing	0–1	8 (36%)	
	2–5	10 (45%)	
	6–10	4 (18%)	
	10+	0 (0%)	
Years of experience in Performance Testing	0–1	13 (59%)	
	2–5	8 (36%)	
	6–10	1 (5%)	
	10+	0 (0%)	
Familiarity with performance testing concepts	No experience	0 (0%)	
	Beginner	9 (41%)	
	Intermediate	8 (36%)	
	Advanced	4 (18%)	
	Expert	1 (5%)	
How often is system performance testing carried out in industry?	1 (Never)	3 (14%)	
	2	6 (27%)	
	3	6 (27%)	
	4	3 (14%)	
	5 (Very often)	3 (14%)	
Which performance testing tools have you used? (Select all that apply)	Apache JMeter	13 (59%)	
	Locust	8 (36%)	
	Gatling	1 (5%)	
	LoadRunner	2 (9%)	
	K6	2 (9%)	
	None	3 (14%)	

Concerning the three practitioners with whom the interviews were conducted, the first has ten years of professional experience in software testing. As highlighted in the follow-up interview, their career progressed from Software Quality Assurance Specialist to Senior Quality Assurance Automation Engineer, in which they primarily test APIs and back-end services at scale, with performance directly affecting business-critical operations. Performance testing is mainly conducted using Locust due to its Python integration and distributed execution capabilities. The performance goals vary by context, from handling spike loads to ensuring long-term system stability.

The second practitioner has four years of professional experience and reported experience as a Test Automation Engineer, currently working as a Software Quality Engineer. Their role includes both back-end and end-to-end testing, with performance testing applied particularly to REST APIs

and asynchronous workflows. In this case as well, tools such as Locust and Playwright are used, highlighting the relevance of performance validation in event-driven and distributed systems.

The third practitioner holds a Ph.D. in Critical Systems Engineering and brings a combination of academic and industrial experience. He previously co-founded a startup, where he was primarily responsible for software quality assurance, and later assumed a Technical Manager role in another company. His experience includes performance testing across multiple industrial contexts, particularly for web-based applications with cloud-based back-end services. In these settings, performance testing activities were conducted using tools such as JMeter and Locust, supporting both traditional load testing and more flexible, script-based approaches.

Overall, the composition of the participant sample reflects two complementary perspectives. On the one hand, the survey captures a broader population of practitioners, including early-career participants who are directly involved in performance testing activities in everyday development contexts. On the other hand, the semi-structured interviews involve practitioners with more extensive experience, providing deeper and more contextualized insights into performance testing practices. This combination allows us to balance breadth and depth in the analysis: the survey offers a wider view of how bad practices are perceived across practitioners, while the interviews enable a more nuanced interpretation of these perceptions by incorporating the viewpoints of more experienced professionals. As such, the two data sources play a complementary role, supporting a more comprehensive understanding of performance testing bad practices without claiming full representativeness across all levels of expertise.

5 Analysis of the Results

Our study, which analyzed 415 selected documents, identified 33 bad practices, grouped into three main categories corresponding to the phases of performance testing. In *Phase II*, participants evaluated the practices using Likert-scale questions and were also given the opportunity to provide open-ended feedback; however, these responses did not reveal additional bad practices that would require extending the taxonomy. Additionally, to enrich the practitioners' perspective, we reported insights from semi-structured interviews with three expert practitioners for each identified bad practice. Based on the practitioners' perspectives, we refined the taxonomy by removing four practices ($\mathbf{RQ}_1$). In addition, we assessed the severity and prevalence of each of the remaining 29 practices from the taxonomy ($\mathbf{RQ}_2$).

Before presenting the results, it is important to contextualize the interpretation of the survey findings. The survey captures perceptions from a heterogeneous population of practitioners with varying levels of experience in performance testing. As such, the reported assessments of agreement, severity, and prevalence should be interpreted as reflecting a broad practitioner perspective, rather than exclusively the views of highly specialized performance testing experts. This perspective is particularly relevant given how performance testing activities are carried out in practice, where such tasks might be performed by software developers or QA engineers, rather than exclusively by dedicated specialists, as also reflected in the background of our survey participants (Section 4.2.2). To complement this broad perspective, we integrate the survey findings with insights from semi-structured interviews, providing additional depth in the interpretation of the results.

The following sections present the taxonomy of performance testing bad practices, grouped into three main categories corresponding to the phases of performance testing: test design (Section 5.1), test execution (Section 5.2), and test analysis (Section 5.3).

5.1 Test Design

As shown in Table 10, our analysis of the gray literature identified nine recurring bad practices in the test design phase. While they differ in scope, all of these factors compromise the realism and

Table 10. Performance testing bad practices identified in the Test Design phase. The last column refers to the number of sources associated with the specific bad practice, and percentages are computed based on the entire corpus of selected sources.

ID	Bad Practice	Description	#Documents
TD-BP01	Unrealistic user behavior	Simulating users that do not reflect actual usage patterns	60 (14%)
TD-BP02	Single scenario focus	Test scripts focus solely on the "happy path", neglecting stress, soak, or failure scenarios.	43 (10%)
TD-BP03	Hardcoded or static input	Using fixed or static input data prevents realistic variation, can trigger caching effects, and may make the system appear faster than under real-world conditions.	36 (9%)
TD-BP04	Ignoring think times	Sending operations, requests, or interactions as fast as possible without incorporating delays between user actions results in unrealistic system overloads since real users naturally interact at a slower pace.	34 (8%)
TD-BP05	Too short or too long duration	Tests that are too short may fail to reveal issues such as memory leaks or long-term performance degradation, whereas overly long tests can waste resources without providing additional insights.	16 (4%)
TD-BP06	Insufficient test planning	Tests lack clearly defined performance goals and objectives, and fail to identify key performance indicators, limiting the usefulness and interpretability of the results.	50 (12%)
TD-BP07	Improper ramp up and cool down	Abruptly starting or stopping the load without gradual warm-up/cool-down phases results in excessive stress on the system.	34 (8%)
TD-BP08	Confusing performance activities	Not distinguishing between load, stress, soak, or spike tests, leading to incorrect test design and goals.	11 (3%)
TD-BP09	Outdated or unmaintained tests	Tests are not updated with application changes, leading to obsolete scenarios and unreliable results.	16 (4%)

reliability of performance testing, thereby reducing its ability to produce actionable results. Below, we discuss the most prominent insights emerging from our taxonomy, enriched with examples extracted from the documents. In addition, Figure 5 reports the distribution of responses concerning the nine bad practices in the test design phase.

5.1.1 🏷 *Unrealistic User Behavior (TD-BP01).*

Description. The simulation of unrealistic user behavior is the most common, reported in 60 documents. In practice, this often means building test plans that resemble automated bots rather than real users. For instance, testers may design scripts that repeatedly submit the same requests or interactions without variation or ignore realistic sequences of actions, such as login flows or multi-step transactions. Several documents stress the importance of *"putting yourself in the mind of the user"* by reproducing real navigational paths, mixing browsers or devices, and ensuring that test flows align with production flows. Otherwise, the outcomes are decoupled from the actual user experience and may obscure performance bottlenecks.

Practitioners' Perceptions. The survey indicates strong agreement on its relevance, with 68% of participants recognizing it as a bad practice. Many also rated its severity as moderate (36%) and high (36%), although only a minority reported frequent occurrence. It suggests that, although its risks are well understood, it may not be prevalent in everyday projects. Expert interviews reinforce this interpretation. All experts noted that tests are sometimes designed around technically demanding scenarios rather than representative user flows, which can skew performance insights and resource priorities. At the same time, they stressed that "unrealistic" patterns are not inherently wrong: while realistic sessions are essential for traffic simulation, extreme behaviors are justified in stress

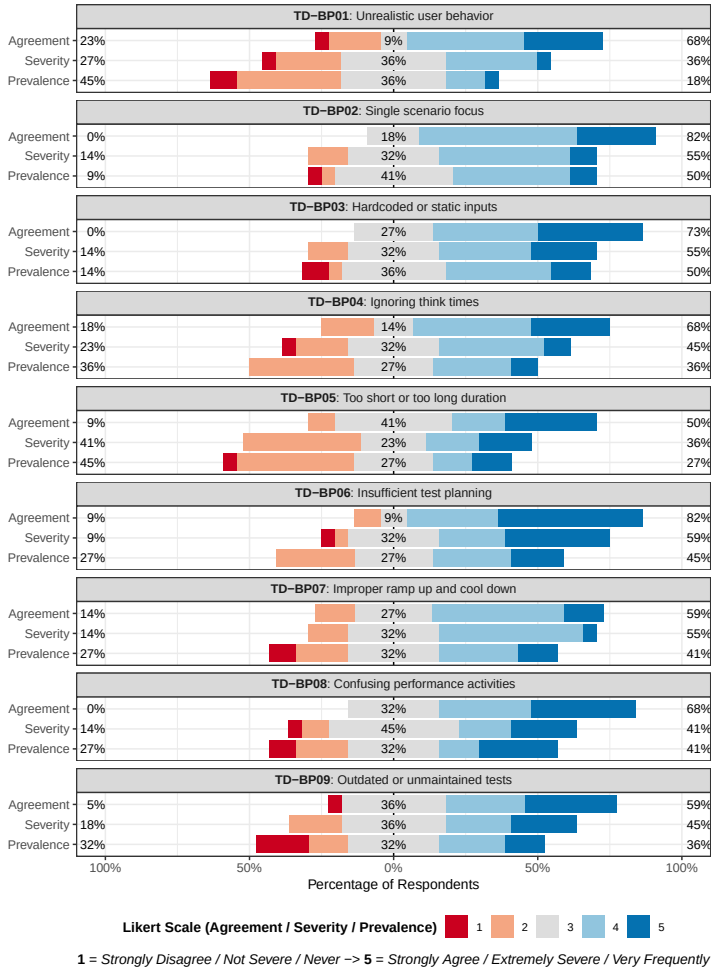

Fig. 5. Survey response distribution highlighting perceived agreement, severity, and prevalence of Test Design (**TD-BP**) bad practices.

testing. The key, therefore, is to ensure that test design matches the intended goal, thereby avoiding a mismatch between technical exploration and user-centric evaluation.

5.1.2 🏷 *Single Scenario Focus (TD-BP02).*

Description. Another recurring practice is the focus on a single scenario, highlighted in 43 documents. Tests are often designed to cover only the "happy path", i.e., the most common successful case, while neglecting alternative paths such as error handling, recovery situations, or long-running sessions. This narrow scope reduces the likelihood of detecting issues that arise under stress or in infrequently used patterns. Literature emphasizes the need for realistic variability and broader coverage, since only then can performance testing expose system fragility under diverse conditions.

Practitioners' Perceptions. Survey results show strong consensus on its relevance, with 82% of participants agreeing or strongly agreeing that this is a bad practice. Over half rated its severity as

very or extremely high (55%), and its prevalence was notable, with most respondents reporting that they frequently or occasionally experienced it in practice. The interviews reinforced these findings, highlighting the tendency to overemphasize a single "happy path." Experts noted this often occurs when developers test their own systems, consciously or unconsciously prioritizing stable and familiar functionalities while neglecting less predictable or failure-prone areas. This bias undermines the representativeness of performance testing and risks overlooking regressions in evolving features. They further stressed that scenarios must adapt as systems evolve. For instance, an e-commerce application may shift from write-heavy to read-heavy usage over time. Mitigating this practice requires diversifying test scenarios to cover multiple user behaviors, including stress and failure conditions, while keeping them modular and reusable to facilitate ongoing maintenance.

5.1.3 🏷 *Hardcoded or Static Input (TD-BP03).*

Description. The issue of hardcoded or static input data is mentioned in 36 documents, and has also emerged as particularly relevant. When fixed datasets or repetitive values are used, the system may appear to perform better than it actually does, because caching mechanisms can be triggered. Practitioners recommend instead generating diverse, dynamic data to avoid hiding real bottlenecks.

Practitioners' Perceptions. Survey results highlight strong practitioner consensus. Nearly three-quarters of participants (73%) agreed or strongly agreed that this is a problematic practice, with more than half rating its severity as very or extremely high (55%). Prevalence was also notable: half of the respondents reported encountering it frequently in practice, suggesting that static or reused inputs are both harmful and widespread in performance testing. The expert discussions reinforced this view, emphasizing that static inputs can yield misleading results due to caching effects: repeated identifiers may cause the system to serve responses from the cache, artificially inflating performance measurement. Static data can also create conflicts in concurrent executions, such as HTTP 409 errors when multiple tests operate on the same resource. To avoid these issues, experts emphasized the importance of generating unique, variable data by default, reserving static inputs only for targeted scenarios, such as explicitly testing caching mechanisms. They also noted that the severity of this practice depends on context: while controlled environments with known caching behavior may tolerate static data, uncontrolled or API-based systems require variability to capture realistic behavior. Finally, oversimplified inputs risk masking bottlenecks and preventing the discovery of system limits.

5.1.4 🏷 *Ignoring Think Times (TD-BP04).*

Description. We found 34 documents reporting ignoring thinking times as problematic. Many test scripts send operations, requests, or interactions as quickly as possible, without introducing delays between actions. While this creates heavy loads, it does not reflect real-world interaction, in which users naturally pause between clicks or form submissions. Overlooking think times, therefore, leads to inflated loads and misleading conclusions about scalability.

Practitioners' Perceptions. Survey results indicate that 68% of participants agree that omitting think times constitutes a bad practice. Approximately 45% rated it as very or extremely severe, while 36% reported encountering it frequently in their work. These responses suggest that while the issue is recognized, its prevalence is perceived as moderate. Insights from expert interviews further contextualize these findings. Ignoring think times often reflects an engineering mindset that assumes the system should handle a constant stream of requests, ignoring natural user pauses. For instance, actions such as generating a voucher and presenting a QR code involve inherent delays that should be accounted for in tests. Omitting think times can create unrealistic load patterns and may cause flakiness due to misaligned asynchronous operations. Experts also noted that the use of

think times depends on the testing goal: user-emulation tests should include realistic pauses, while stress or capacity tests may omit them to probe system limits. Ignoring think times indiscriminately in user-focused scenarios is therefore considered a bad practice.

5.1.5 🏷 *Too Short or Too Long Duration (TD-BP05).*

Description. We found 16 documents reporting issues related to inappropriate test duration. Some tests are too short to capture critical long-term issues such as memory leaks or resource exhaustion, while others are excessively long, wasting time and infrastructure without yielding additional insights. Issues with duration (TD-BP05) often interact with improper ramp-up or cool-down handling (TD-BP07), since insufficient stabilization time can make a test appear shorter than needed or prolong it unnecessarily.

Practitioners' Perceptions. Survey results indicate moderate agreement on its relevance, with 50% of participants agreeing or strongly agreeing that this constitutes a bad practice, while 41% were neutral. Regarding severity, 36% considered it very or extremely severe, and 23% moderately severe. Prevalence was judged lower, with only 27% reporting frequent occurrence and 45% rarely encountering it. Insights from expert interviews reinforce the importance of recognizing test duration as a critical bad practice. Experts emphasized that test duration is critical for capturing realistic system behavior: short tests may miss subtle issues such as memory leaks or resource exhaustion, whereas overly long tests can consume resources without providing additional insights. However, the need for long tests is often mitigated by modern frameworks and libraries that manage resources efficiently. Experts recommended evaluating test duration based on system complexity and testing goals, balancing realistic performance evaluation with practical constraints. They highlighted that collaborative monitoring strategies can reveal long-term performance issues even with moderate-duration tests, while excessively long tests may still be justified in security or resilience scenarios.

5.1.6 🏷 *Insufficient Test Planning (TD-BP06).*

Description. We found that 50 documents report missing or poorly defined goals, KPIs, or thresholds, which makes results difficult to interpret and reduces the usefulness of the collected data. These problems are frequently linked to a broader lack of knowledge about performance testing, where it is not consistently integrated into the software development lifecycle. In such cases, involving specialized teams or experts can help establish clear strategies and apply appropriate performance evaluations.

Practitioners' Perceptions. Insufficient test planning was strongly recognized as a problematic practice. Survey responses show that 82% of participants agreed or strongly agreed that insufficient planning constitutes a bad practice. Severity ratings were moderate to high, with 41% of respondents considering it very or extremely severe. Prevalence was more evenly distributed, with 41% reporting frequent occurrence, suggesting that lack of formal planning is common but not universal. Experts emphasized that poor planning—especially the lack of clear performance goals or data-driven targets—is a common and impactful issue. They highlighted that performance objectives must align with the nature of the service and its users; for example, a 5-second delay may be acceptable for casual users but frustrating for frequent professional users. Experts also warned against allowing stakeholder unverified assumptions to dictate targets, which can lead to overengineering or misaligned tests. Overall, data-informed, context-specific planning is essential for effective performance testing.

5.1.7 🏷 *Improper Ramp Up and Cool Down (TD-BP07).*

Description. We found 34 documents describing problems regarding the ramp-up and cool-down phases of performance tests. Starting or stopping the load abruptly imposes unrealistic stress on the system, often resulting in transient states such as cold caches or uninitialized resources. This can cause misleading spikes in resource consumption and lead to tuning decisions based on scenarios unlikely to occur in production. To ensure realistic results, performance tests must allow the system to reach a steady state. They should introduce the load gradually, mirroring real-world conditions where infrastructure is "warmed up" before peak demand is reached.

Practitioners' Perceptions. Survey results show that 59% of participants agreed or strongly agreed that this is a bad practice, while 36% were neutral and only 5% disagreed. Severity was rated high by the majority: 55% considered it very or extremely severe, and 32% moderately severe. Prevalence was more varied: 41% reported encountering it frequently, 32% occasionally, and 27% rarely or never. This suggests that while practitioners recognize the importance of proper ramp-up and cool-down phases, their occurrence in real projects is uneven. The expert interviews provide more in-depth insights. Experts emphasized that abrupt load changes without proper ramp-up or cool-down phases can distort performance measurement, particularly during initial system "cold starts" in environments such as cloud containers or serverless architectures. While stress or high-load tests may still reveal system capacity, gradual load changes better reflect realistic user behavior and uncover steady-state performance issues. Experts highlighted that the severity of this bad practice depends on the system and test objectives, but involving performance-focused expertise is crucial for complex user journeys, as neglecting ramp-up and cool-down can lead to misleading results or overlooked bottlenecks.

5.1.8 🏷 *Confusing Performance Activities (TD-BP08).*

Description. We found 11 documents showing that practitioners often conflate load, stress, spike, or soak testing, leading to test designs that do not align with their intended objectives. In several cases, this issue is intertwined with TD-BP06, as unclear goals can contribute to the misuse of performance test types.

Practitioners' Perceptions. Survey responses show that 68% of participants agreed or strongly agreed that this practice is a problem. Severity was rated moderate to high for both, and prevalence was generally lower, with most respondents reporting occasional occurrences. Experts highlighted that TD-BP08 often arises from equating all performance testing with stress testing, thereby neglecting distinctions among load, stress, soak, and spike tests. They stressed that clarity comes from communicating the test's objective rather than relying on labels alone. Using correct terminology and aligning tests with expected outcomes is key to avoiding misunderstandings.

5.1.9 🏷 *Outdated or Unmaintained Tests (TD-BP09).*

Description. We found 16 documents highlighting the risks associated with outdated or unmaintained tests. When scripts are not updated to reflect changes in the application, scenarios quickly become obsolete, and the corresponding results are unreliable. This highlights the need for continuous test review and refactoring.

Practitioners' Perceptions. 59% of the respondents indicated that this is a bad practice. Severity was rated moderate to high for both, and prevalence was generally lower, with most respondents reporting occasional occurrences. Experts emphasized that neglecting test maintenance entails tangible risks and is considered a bad practice across all testing activities.

Table 11. Performance testing bad practices identified in the Test Execution phase. The last column refers to the number of sources associated with the specific bad practice, and percentages are computed based on the entire corpus of selected sources.

ID	Bad Practice	Description	#Documents
TE-BP01	Testing in non-production-like environments	Running performance tests in environments that differ significantly from production (e.g., reduced hardware, missing services, simplified configurations), leading to misleading results that do not reflect real-world performance.	52 (13%)
TE-BP02	No proper cleanup	Leftover test data or state persists across runs, skewing results and reducing test reliability.	12 (3%)
TE-BP03	Lack of modularity and reusability	Duplication of logic, absence of shared functions or libraries, missing documentation, and poorly managed or unversioned test data.	13 (3%)
TE-BP04	Poor test data management	Reliance on static endpoints, ports, or credentials instead of configurable variables, with missing dynamic data handling or correlation between operations, requests, etc.	36 (9%)
TE-BP05	Don't use recorded scripts	Recorded scripts (e.g., from JMeter or LoadRunner) often contain noise and lack proper handling of dynamic values, correlations, or think times. Using them without cleanup leads to unrealistic workloads and requires additional manual effort to fix.	7 (2%)
TE-BP06	Don't forget to simulate the network condition	Running tests only on fast LAN connections without simulating latency or limited bandwidth can miss real-world bottlenecks.	15 (4%)
TE-BP07	Poor naming conventions	Using generic names (e.g., "HTTP Request 1") makes test scripts and reports hard to interpret.	9 (2%)
TE-BP08	Over-Mocking	Excessive mocking of dependencies leads to unrealistic workloads and obscures underlying system bottlenecks.	6 (1%)
TE-BP09	Incorrectly reusing connections or sessions	Sharing sessions/cookies across virtual users may not represent real-world load and causes inconsistent test behavior.	5 (1%)
TE-BP10	Not integrating performance testing with CI/CD	Performance testing conducted ad hoc, rather than as part of automated pipelines, yields late and unreliable regression detection.	32 (8%)

5.2 Test Execution

As shown in Table 11, we identified ten bad practices in the execution of performance tests. Figure 6 reports the distribution of the practitioners' responses concerning these bad practices.

5.2.1 🏷 *Testing in Non-production-like Environments (TE-BP01).*

Description. A recurring concern is the lack of representativeness of the execution environment, reported in 52 documents. Tests are often run in staging or simplified environments that differ from production in hardware capacity, service configuration, or network setup. This makes results unreliable, as they fail to capture real-world performance conditions.

Practitioners' Perceptions. 55% of the practitioners recognize this as an important concern. Practitioners agreed or strongly agreed that it is a bad practice, with half rating it very or extremely severe, while prevalence was more evenly distributed. Insights from expert interviews provide a nuanced perspective. Experts agree that testing in non-production-like environments can produce misleading results, either pessimistic if the system is underpowered or overly optimistic if critical components are absent. However, they note that such setups are often unavoidable during development and can be informative if differences from production are understood.

5.2.2 🏷 *No Proper Cleanup (TE-BP02).*

Description. We found 12 documents reporting insufficient cleanup between test executions. Residual state or leftover data can skew subsequent executions, producing wrong results and

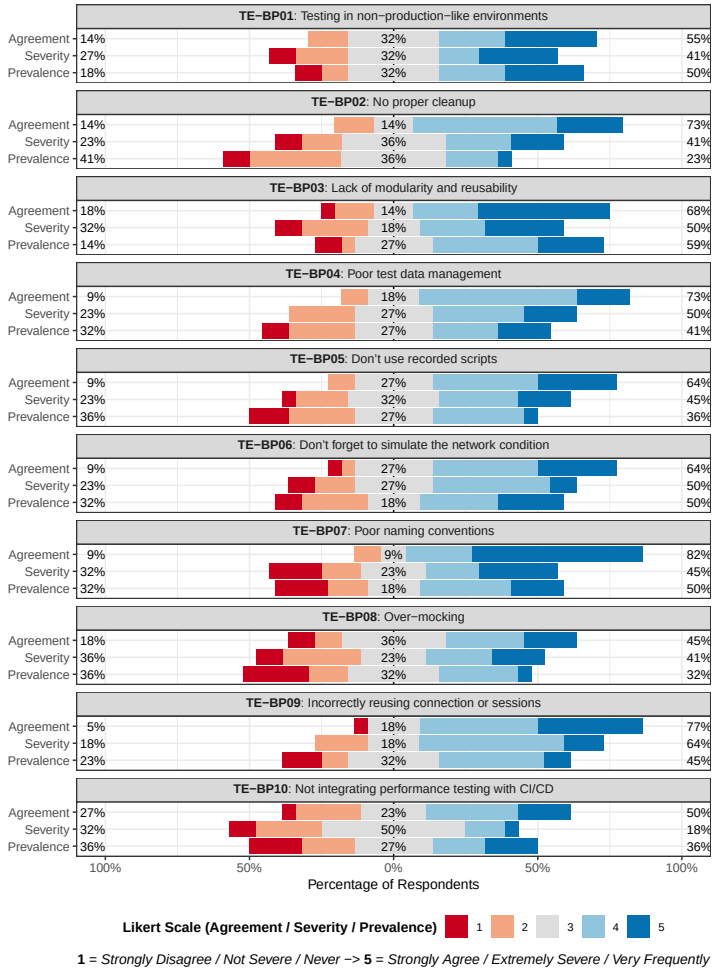

Fig. 6. Survey response distribution highlighting perceived agreement, severity, and prevalence of Test Execution (**TE-BP**) bad practices.

making it difficult to isolate the impact of specific test conditions. Improper cleanup is closely related to TE-BP03 and TE-BP04, since residual data can undermine efforts to structure test logic clearly and maintain consistent, reusable components.

Practitioners' Perceptions. Survey results indicate strong recognition of the issue: 73% of participants agreed or strongly agreed that this constitutes a bad practice. Regarding severity, 41% rated it very or extremely severe, 36% rated it moderately severe, and 23% rated it low. Perceived prevalence was more varied: 41% reported frequent occurrence, 36% occasional, and 23% rarely or never, suggesting that while practitioners acknowledge the risks, the actual frequency of encountering this problem varies across contexts. Insights from the expert interviews provide additional nuance. Experts noted that partial or delayed cleanup is common and sometimes intentional, particularly for simulating realistic environments with preexisting data or long-running scenarios. This approach helps evaluate system performance under more realistic conditions but introduces risks such as

hidden dependencies, confusion over test state, and potential impacts on modularity and reliability. Experts recommended a progressive strategy: start tests with a clean state to isolate potential issues, then gradually introduce more realistic system conditions as understanding improves. This balances the need for realism with the need to maintain clarity and control in performance testing.

5.2.3 🏷 *Lack of Modularity and Reusability (TE-BP03).*

Description. Another recurring issue, described in 13 documents, is insufficient modularization and reuse of test code and data. Poor structuring leads to duplicated logic, inconsistencies across test scripts, and challenges in reproducing results. Modular and reusable components are essential for maintaining consistency and reducing maintenance effort across performance test suites.

Practitioners' Perceptions. The survey shows that most practitioners recognize it as a problematic practice (68%), with a strong majority agreeing on its relevance. Participants also rated it as fairly severe (59%), reflecting concerns about maintainability and the long-term impact on test suites. Opinions on how often this issue occurs were more varied, suggesting that while the problem is acknowledged, its prevalence may depend on project context, team practices, or the expected lifetime of the system under test. Findings from expert interviews highlighted that modularity and reusability are crucial for long-lived, product-oriented systems, where tests evolve in tandem with the application. In short-term or one-off projects, less structured approaches may be sufficient. While experienced engineers typically adopt modular designs when possible, a lack of modularity can become a liability in sustained development. Practices such as BDD (Behavior-Driven Development) with reusable steps naturally support maintainable test suites. Overall, the experts emphasized that although modularity and reusability are valuable for general software and test quality, they do not represent a strict bad practice in performance testing.

5.2.4 🏷 *Poor Test Data Management (TE-BP04).*

Description. Another critical issue relates to test data handling. Poor management of test data is the most frequently reported execution-related smell. We identified 36 documents highlighting issues such as static endpoints, hardcoded credentials, or missing operation/request relation.

Practitioners' Perceptions. Survey results show strong agreement on its relevance: 73% of participants agreed or strongly agreed that this constitutes a bad practice. Half of the respondents rated it as very or extremely severe, and 27% rated it as moderately severe. Prevalence was perceived as moderately high, with 41% reporting frequent occurrence, 27% occasional, and 32% rarely or never. Expert interviews reinforced these findings. Both experts emphasized that the lack of handling in test data is a clear bad practice. Especially in the context of dynamic correlations between requests, such as session tokens or access keys. Ignoring these dependencies was described as a clear mistake, since it produces unrealistic or broken test scenarios. While many modern frameworks (e.g., Locust) make extracting and reusing dynamic values straightforward, others (e.g., Postman) complicate this process and increase the risk of misaligned tests. Although some testers might intentionally omit correlation to simulate chaotic conditions, experts stressed that, in most cases, proper correlation management is indispensable for accurate and reliable results.

5.2.5 🏷 *Do Not Use Recorded Scripts (TE-BP05).*

Description. Several documents also warn against the uncritical use of recorded scripts. While recording provides a quick way to bootstrap test scenarios, the resulting scripts often contain noise, redundant steps, and inadequate handling of dynamic values, correlations, or think times. Using them as-is results in unrealistic workloads. Moreover, fixing recorded scripts to ensure

reliability typically requires substantial manual effort, thereby negating the initial advantage of rapid generation.

Practitioners' Perceptions. Survey responses show that 64% of participants agreed or strongly agreed that this is a bad practice. In terms of severity, opinions were more evenly distributed: 45% rated it very or extremely severe, 32% rated it moderately severe, and 23% rated it low. Prevalence was mixed as well, with 36% reporting frequent occurrence, 27% occasional, and 36% rarely or never. Expert insights suggest that relying on recorded scripts "as-is" is problematic, though it is often considered low severity. Two experts stressed that while recorded scripts rarely break tests, they undermine maintainability, creating technical debt and extra work when applications evolve. Another expert highlighted the lack of realism, comparing recorded scripts to "microwave meals" — quick but lacking the customization of handcrafted tests. While refining scripts requires more effort and skill, it ensures fidelity in complex scenarios (e.g., involving databases or external systems). Overall, experts agreed that recorded scripts are acceptable only as a starting point, but relying on them without refinement reduces test realism and long-term value.

5.2.6 🏷 *Do Not Forget to Simulate the Network Condition (TE-BP06).*

Description. The simulation of network condition threats to the realism of performance testing. We found 15 documents emphasizing that overlooking network variability (**TE-BP06**) leads to overly optimistic and unreliable performance assessments. Many tests are executed only in fast LAN environments, ignoring the impact of latency, bandwidth limitations, or unstable connections. This omission conceals bottlenecks that may significantly impact user experience in production, particularly for web and mobile applications.

Practitioners' Perceptions. Survey results indicate strong practitioner consensus: 65% of the practitioners agreed or strongly agreed that these constitute bad practices, with severity and prevalence rated high by about half of practitioners. Regarding network conditions, experts emphasized that neglecting to account for latency, bandwidth constraints, or unstable links can obscure real-world bottlenecks, especially in global applications where infrastructure quality varies (e.g., Europe vs. India). Without such considerations, systems may appear to perform well under ideal laboratory conditions but fail under realistic use.

5.2.7 🏷 *Poor Naming Conventions (TE-BP07).*

Description. Poor naming conventions are a concern identified in 9 documents. It may seem trivial, but it carries practical consequences: when test scripts use generic labels such as HTTP `Request 1` (the default nomenclature in Apache JMeter), they become difficult to maintain, interpret, and debug, particularly in large projects or when results are shared across teams.

Practitioners' Perceptions. Survey results show strong agreement that this is a bad practice: 82% of respondents agreed or strongly agreed, while half rated it as severe or extremely severe. Prevalence was also substantial, with nearly half of the participants reporting that it is common in practice. Experts consistently framed poor naming as a general concern across software and test quality rather than as something specific to performance testing. Nonetheless, they stressed its practical consequences: vague or meaningless names (e.g., "HTTP Request 1/2" or arbitrary codes like "ABC") hinder understanding of what each test action does and why it exists. This ambiguity propagates into performance dashboards and reports, where it becomes difficult—or even impossible—to link spikes, errors, or slowdowns to meaningful business actions without an external "legend." One expert noted that weak names often reflect a deeper misunderstanding of the system, undermining design,

review, and maintainability. Another likened it to naming variables as "1" or "2" in programming: technically valid, but practically useless.

5.2.8 🏷 *Over-Mocking (TE-BP08).*

Description. Other problems concern the realism and interpretability of test scripts, though they were less frequently reported in the literature. Over-mocking dependencies, as noted in six documents, can strip away critical parts of the workload and obscure real bottlenecks, because excessive reliance on mocked services fails to reflect the complexity of production environments.

Practitioners' Perceptions. Survey results reveal that 45% of the practitioners agreed or strongly agreed, with moderate severity and prevalence. These results suggest that, although these practices are recognized as risky, they are often mitigated or tolerated due to practical constraints. Experts emphasized that, whenever possible, performance tests should be conducted in production-like environments with minimal mocking to ensure realistic, actionable results, while acknowledging that practical trade-offs sometimes make strict adherence challenging. Indeed, one of the experts further clarified that the impact of mocking is strongly dependent on the testing objective: when the goal is to obtain a qualitative assessment (e.g., whether the system can sustain expected load conditions), the use of mocks may be acceptable to provide approximate insights. However, when aiming for quantitative measurements, such as precise response times, excessive mocking becomes detrimental, as it obscures the real behavior of external dependencies.

5.2.9 🏷 *Reusing Connections or Sessions Incorrectly (TE-BP09).*

Description. Similar to the simulation of the network condition (**TE-BP06**), the reuse of sessions also threatens the realism of performance testing. The incorrect reuse of sessions or connections across virtual users, as reported in five documents, results in unrealistic load distribution and resource contention that do not accurately reflect real user behavior. Although only five documents highlighted these pitfalls, they point to subtle yet consequential errors that can invalidate performance testing results.

Practitioners' Perceptions. Survey results indicate strong practitioner consensus: 77% of participants agreed or strongly agreed that these constitute bad practices. At the same time, 64% of respondents consider the practice severe, and 32% consider it frequent. Experts warned that reusing sessions or connections across virtual users distorts load simulation: it can either artificially reduce contention or create bottlenecks unrelated to actual user behavior. While modern architectures mitigate some risks, improper handling of sessions, cookies, or protocols remains a clear source of misleading results.

5.2.10 🏷 *Not Integrating With CI/CD (TE-BP10).*

Description. An organizational challenge is the lack of integration into CI/CD pipelines, mentioned in 32 documents. When performance testing is performed manually, it loses its preventive role and becomes reactive, with regressions detected late in the development lifecycle.

Practitioners' Perceptions. Survey results were mixed: 50% of respondents agreed or strongly agreed that this is a bad practice, 23% were neutral, and 27% disagreed. Severity was evenly distributed, with approximately one-third of participants rating it as low, medium, or high. Prevalence also varied: while half reported occasional occurrence, only 18% considered it rare, and 32% reported frequent occurrence. These results suggest that practitioners do not consistently view the lack of CI/CD integration as a clear bad practice. Insights from expert interviews clarify this ambivalence. Experts noted that integration with CI/CD is not universally necessary, and its value depends

strongly on context. Embedding performance tests directly in the pipeline can provide continuous feedback. Still, it also introduces practical challenges: such tests can be resource-intensive, require complex monitoring setups, and significantly slow build times if they run for hours. As two experts emphasized, performance testing within CI/CD needs to be carefully designed and coordinated with the right stakeholders to ensure meaningful results without disrupting development workflows. Instead of always embedding tests in pipelines, experts suggested hybrid strategies: running lightweight smoke or regression performance tests in CI/CD for quick feedback, while scheduling more extensive load or stress tests in dedicated environments outside of the pipeline. In this way, performance testing remains systematic and repeatable without overburdening the CI/CD process.

Table 12. Performance testing bad practices identified in the Test Analysis phase. The last column refers to the number of sources associated with the specific bad practice, and percentages are computed based on the entire corpus of selected sources.

ID	Bad Practice	Description	#Documents
TA-BP01	Ignoring warm-up phases	Measuring performance from the first request without giving the system time to warm up (e.g., JIT compilation, caching), resulting in artificially bad metrics.	10 (2%)
TA-BP02	Lack of test repeatability	Tests are non-deterministic when factors such as caching, background tasks, or external dependencies aren't controlled, making results difficult to reproduce across environments. Use relative assertions to mitigate variability.	18 (4%)
TA-BP03	Rigid assertions	Using fixed expectations (e.g., 200 ms average response time) makes tests brittle. Prefer flexible checks such as percentile ranges.	20 (5%)
TA-BP04	No assertion or response validation	Measuring performance without checking if the application responded correctly.	25 (6%)
TA-BP05	Excessive Listeners/Logging	Using too many listeners (e.g., View Results Tree) during load tests inflates memory and CPU usage, skewing performance results.	11 (3%)
TA-BP06	Lack of baseline comparison	Running tests without reference metrics or historical benchmarks makes it difficult to identify performance regressions.	21 (5%)
TA-BP07	Skipping test validation	Executing long tests without verifying early results can waste time if the test is already failing.	8 (2%)
TA-BP08	High thread count on underpowered machines	Generating high load from underpowered machines can bottleneck the load generator itself, skewing test results rather than stressing the system under test.	13 (3%)
TA-BP09	Ignore test result variability	Take one test run as the final truth.	12 (3%)
TA-BP10	Inadequate resource monitoring	Failing to track system resources (CPU, memory, disk, network) and key performance metrics (response time, throughput, errors).	38 (9%)

5.3 Test Analysis

As shown in Table 12, during the test analysis phase, 10 bad practices were identified that undermine the reliability and interpretability of performance test outcomes. In addition, Figure 7 presents the survey results.

5.3.1 🏷 *Ignoring Warm-up Phases (TA-BP01).*

Description. One recurring bad practice is ignoring warm-up phases, reported in ten documents. Measuring from the very first operations/requests captures transient effects such as just-in-time compilation, cache initialization, or service bootstrapping. This can artificially inflate response times and misrepresent steady-state performance, leading to misguided optimization efforts.

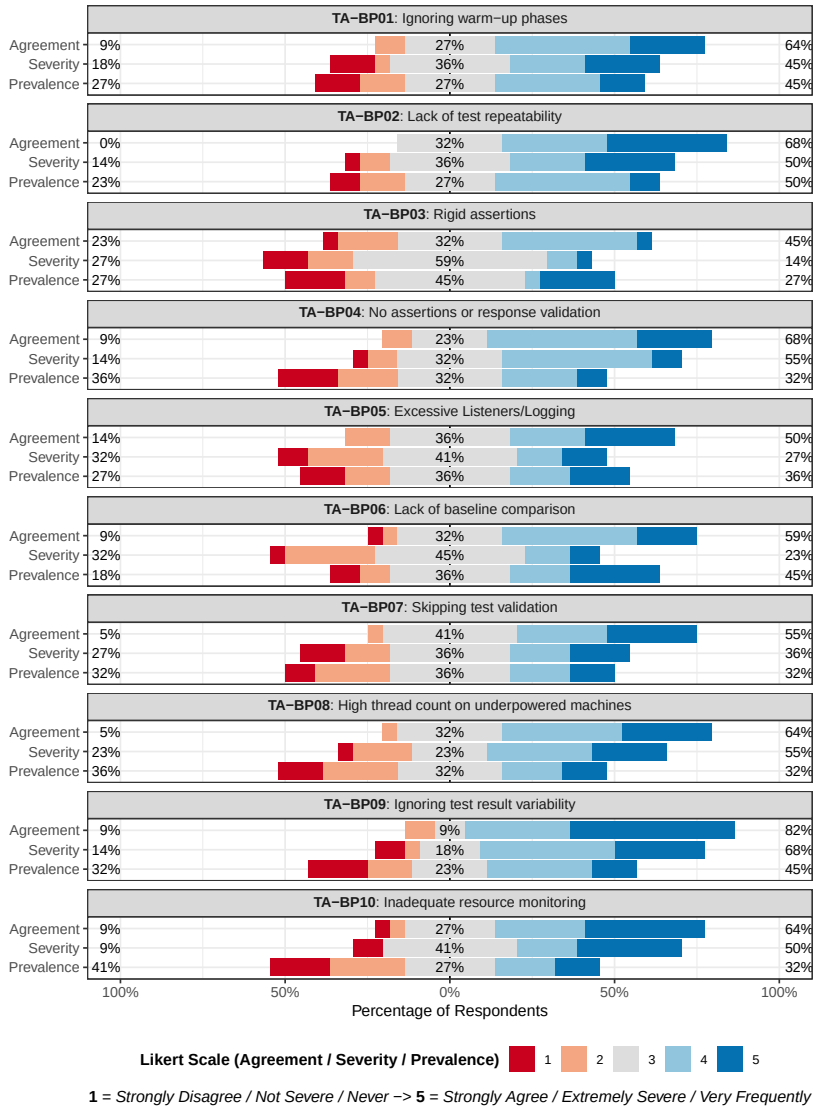

Fig. 7. Survey response distribution highlighting perceived agreement, severity, and prevalence of Test Analysis (**TA-BP**) bad practices.

Practitioners' Perceptions. The survey results show that 64% of respondents agreed or strongly agreed that this is a bad practice, 27% were neutral, and 9% disagreed. Severity was rated as high by 45%, medium by 36%, and low by 18%. Prevalence was mixed: 45% reported frequent occurrence, 27% occasional, and 27% rare. This suggests that while practitioners are aware of the risks of neglecting warm-up periods, the extent to which this affects projects may depend heavily on context and testing environments. Expert insights confirm that ignoring warm-up phases is problematic, as measuring performance while the system is still "cold" can produce misleading results (e.g., inflated latencies after serverless cold starts). However, experts also emphasized context: if cold starts are

part of the expected workload (e.g., all employees logging in simultaneously at the start of the day), they must be explicitly included in the testing strategy. Overall, warm-up phases should be excluded from general performance metrics unless they directly represent real-world usage patterns.

5.3.2 🏷 *Lack of Test Repeatability (TA-BP02).*

Description. The lack of test repeatability, mentioned in 18 documents, is another common problem. When factors such as uncontrolled caching, background processes, or unstable external dependencies are not managed, results vary significantly between runs. This non-determinism makes it difficult to distinguish genuine performance regressions from random fluctuations.

Practitioners' Perceptions. 68% of participants agreed or strongly agreed it is a bad practice, with none disagreeing. Severity was rated as high by 50%, medium by 27%, while prevalence was frequent by 50%, occasional by 36%, and rare by 14%. Expert insights highlight that repeatability is essential for trustworthy performance testing. Complex or poorly controlled tests often produce non-deterministic results, undermining confidence in the outcomes. To improve repeatability, experts emphasize the importance of modular test design, proper cleanup between runs, and control of variables that can influence results, such as caching behavior, background tasks, and external dependencies.

5.3.3 🏷 *Do Not Use Rigid Assertions (TA-BP03).*

Description. Rigid assertions, such as requiring each operation/request to complete under a fixed 200 ms threshold, were mentioned in 20 documents. These brittle checks can flag acceptable systems as failing due to occasional outliers.

Practitioners' Perceptions. Survey results show that 45% of respondents agreed or strongly agreed that the practice is relevant. In terms of severity, participants considered the practice moderately to highly severe (45% high, 27% medium). Regarding prevalence, the practice was reported by 27% of respondents and by 59% occasionally. Experts acknowledged that rigid, hardcoded assertions (e.g., "must respond within 200 ms") are common in early test setups, especially when SLA definitions are unclear. However, they emphasized that such approaches can be misleading: a few slow operations/requests may skew results, whereas most users experience acceptable performance. Using relative or percentile-based thresholds (e.g., 90th percentile) and descriptive statistics, such as medians, better reflect real-world behavior and provide a user-centric view of system performance.

5.3.4 🏷 *No Assertion or Response Validation (TA-BP04).*

Description. As reported in 25 documents, omitting assertions or response validation altogether risks reporting excellent response times while the system is silently returning error messages instead of valid results. Both practices compromise the validity of performance testing.

Practitioners' Perceptions. The survey results show strong agreement with *TA-BP03*, as they both concern the accuracy and reliability of performance test assertions. In particular, 68% of respondents agreed or strongly agreed that this practice is relevant. In terms of severity, participants considered the practice moderately to highly severe (55% high, 32% medium). Regarding prevalence, the practice was reported as frequent by 36% and occasional by 32% of respondents. Experts stressed that performance testing should not only measure execution times or throughput but also validate the correctness of responses. For example, checking HTTP status codes alone is insufficient, as systems may return a 200 response code even when the payload contains errors. Validation should include schema checks and semantic verification to ensure that responses are correct under load.

5.3.5 🏷 *Excessive Listeners/Logging (TA-BP05).*

Description. Associated with testing infrastructure, primarily Apache JMeter usage. In particular, we identified 11 documents that mention excessive listeners and logging as bad practices. It adds significant overhead during execution, skewing response times and throughput. *TA-BP10* is a more generic version of this bad practice.

Practitioners' Perceptions. Survey results show that 50% of respondents agreed or strongly agreed that this is a bad practice, 27% agreed that it is severe, and 36% considered it prevalent. Experts highlighted that overusing listeners or logging can consume CPU and memory, potentially turning the load generator itself into a bottleneck. While in some setups the performance impact was minor due to external monitoring or offloaded metric collection, it remains critical to ensure that the generator is not constrained, as tools such as Locust can signal when the expected load is not met due to resource limits.

5.3.6 🏷 *Lack of Baseline Comparison (TA-BP06).*

Description. Analysis can also suffer from missing context. The absence of a baseline comparison underscores the importance of historical reference points for evaluating performance changes. The lack of a baseline comparison, identified in 21 documents, prevents teams from determining whether performance is improving, degrading, or remaining constant.

Practitioners' Perceptions. Survey results indicate strong agreement on its relevance: 59% of participants agreed or strongly agreed it is a bad practice, 32% were neutral, and 9% disagreed. Severity was considered high by 45% of respondents, medium by 36%, and low by 18%, while prevalence was frequent by 32%, occasional by 45%, and rare by 23%. This suggests that although baselines are valued for detecting regressions and ensuring continuity, they are not always systematically maintained in performance testing workflows. Experts highlighted that baseline comparisons are critical for detecting regressions or unintended performance degradations, such as changes in database indexing or system configuration. They emphasized that historical measurements provide a reference point for assessing whether modifications improve or degrade performance and support meaningful discussions with developers. However, they also noted that in rapidly evolving systems, strict reliance on old baselines can be misleading: introducing new components or features may naturally alter throughput or response times, rendering comparisons with outdated baselines less informative. In such contexts, establishing updated baselines as the system evolves is essential. Overall, while not always mandatory for exploratory or targeted testing, baseline comparisons remain a fundamental practice for reliable performance assessment and regression detection.

5.3.7 🏷 *Skipping Test Validation (TA-BP07).*

Description. Skipping test validation, cited in eight documents, results in long endurance tests being executed even when early results show the test is invalid due to errors or misconfiguration, wasting resources and time. This practice concerns insufficient validation, similarly to TA-BP10.

Practitioners' Perceptions. Survey results show that 55% of participants agreed or strongly agreed that this is a bad practice, with 41% neutral and only 5% disagreeing. Severity was rated high by 36%, medium by 36%, and low by 27%, while prevalence was more evenly distributed. Experts emphasized that continuous observation is crucial, as long-running tests may hide intermittent or gradually emerging issues. For instance, a system might perform well initially but degrade due to failing nodes or dynamically replaced resources. Real-time monitoring helps detect fluctuations in response times, slowdowns, or crashes, allowing early intervention. However, they also noted that skipping early validation can sometimes be a deliberate strategy: allowing a test to run to completion can reveal how the system behaves under sustained stress or repeated failure scenarios. Thus, while

skipping test validation can risk masking problems, it may also provide valuable insights when used thoughtfully within a testing strategy.

5.3.8 🏷 *High Thread Count on Underpowered Machines (TA-BP08).*

Description. Using high thread counts on underpowered machines causes the load generator, not the system under test, to become the bottleneck. We found this practice in 13 documents. In such cases, results reflect the limitations of the testing setup rather than the application.

Practitioners' Perceptions. Survey results show strong agreement on its relevance as a bad practice, with 64% of respondents agreeing or strongly agreeing, 32% neutral, and only 5% disagreeing. Severity was rated high by 55%, medium by 23%, and low by 23%, while prevalence was relatively balanced. Experts have confirmed that running high-load tests on the same machine that hosts the application can distort results, as the load generator itself may become the bottleneck. In practice, they mitigated this by using separate machines or sufficiently powerful virtual environments to ensure that performance measurements reflect the target system's behavior rather than limitations of the testing infrastructure. This approach ensures that the system is properly stressed, producing reliable and meaningful performance insights.

5.3.9 🏷 *Ignore Test Result Variability (TA-BP09).*

Description. As mentioned in 12 documents, this practice leads to relying on a single run as the ground truth, overlooking natural fluctuations that may significantly influence conclusions.

Practitioners' Perceptions. Survey results show that 82% of respondents agreed or strongly agreed it is a bad practice, 9% were neutral, and 9% disagreed; severity was high for 68%, medium for 18%, and low for 14%, with prevalence frequently reported by 45%, occasionally by 23%, and rarely by 32%. Experts highlight the need to account for natural variability in performance results. Relying on a single test execution or overlooking fluctuations can lead to misleading conclusions. Monitoring multiple runs over time and analyzing their variation yields more accurate, stable, and actionable insights. This issue is closely related to TA-BP02, as repeatable test setups facilitate reliable detection and interpretation of variability.

5.3.10 🏷 *Inadequate Resource Monitoring (TA-BP10).*

Description. The most frequently reported bad practice is inadequate resource monitoring, mentioned in 38 documents. Without tracking CPU, memory, disk, and network metrics in parallel with response times, testers can observe symptoms (e.g., slower responses) but miss the root causes (e.g., memory leaks, I/O bottlenecks), yielding results of limited value to stakeholders.

Practitioners' Perceptions. Survey results show that 64% of respondents agreed or strongly agreed that this bad practice is relevant, while 50% considered it extremely severe and 32% considered it prevalent. reflecting that the occurrence depends on tool configurations and testing context. Experts emphasized that neglecting to track system metrics can obscure early signs of performance degradation, such as steadily increasing RAM usage or CPU saturation. Real-time monitoring allows testers to detect trends and intervene before critical failures occur.

6 Discussion, Further Analyses, and Implications

In this section, we first discuss how our findings address the two research questions. Then, we discuss their implications for researchers, practitioners, tool developers, and educators.

6.1 Addressing the Research Questions

Across the two research questions, our study provides a concrete, empirically grounded picture of how bad practices emerge and persist throughout the software performance testing lifecycle. The gray literature review ($\mathbf{RQ}_1$) identified 29 recurring bad practices across 415 practitioner-oriented documents, encompassing the phases of test design, execution, and analysis. These practices compromise realism, methodological rigor, and interpretability, ranging from unrealistic user behavior and static inputs to poor environment representativeness and inadequate resource monitoring.

In the *test design* phase, the most prevalent issues concern unrealistic or overly narrow workload specifications. Many sources described tests built around non-realistic user behaviors, ignored think times, or a single "happy-path" scenario, all of which limit the ability to capture real usage variability. Additional problems, such as hardcoded or static input data, abrupt ramp-up and cool-down configurations, insufficient planning, and outdated or unmaintained scripts, further reduce methodological rigor and increase the likelihood of producing misleading results.

During the *test execution* phase, the primary shortcomings are related to the lack of environmental fidelity and inadequate data handling practices. Tests are frequently run on simplified, non-production-like environments and rely on static data, unrefined recorded scripts, or incomplete cleanup between runs, leading to distorted results. Other recurring issues include insufficient modularity, the absence of network-condition simulation, and incorrect session reuse. The lack of integration with CI/CD indicates that performance testing is still often treated as an ad hoc activity rather than a continuous quality practice.

In the *test analysis* phase, the gray literature highlighted gaps that directly affect interpretability and trustworthiness. Common pitfalls include ignoring warm-up periods, omitting resource monitoring, making overly rigid assertions, failing to include baseline comparisons, and inadequate validation of early test signals. Additional issues, such as excessive logging or improper load configurations, stress the load generator rather than the system under test, thereby obscuring the identification of true performance bottlenecks.

Practitioners ($\mathbf{RQ}_2$) reported broad awareness of these pitfalls but also revealed variability in how these practices are perceived and addressed. Practitioners strongly validated the need for realistic and diversified workloads, consistently recognizing the harms associated with unrealistic user behaviors, static inputs, and scenario oversimplification. However, experts also emphasized that some practices, such as omitting think times or simplifying setups, may be justified in specific goals (e.g., stress testing rather than user emulation). Similarly, during execution, practitioners acknowledged the risks associated with non-production environments and poor data management, but noted that infrastructural and organizational constraints often necessitate trade-offs. Finally, in the analysis phase, respondents consistently emphasized the importance of repeatability, variability assessment, resource monitoring, and appropriate validation, yet highlighted that these principles, though widely understood, are not systematically applied in practice.

From a qualitative perspective, the interviewees discussed concrete trade-offs they routinely face. For example, they reflected on the tension between realism and execution cost: increasing workload diversity and realism improves test representativeness, but often at the expense of longer execution times and more complex maintenance. In practice, they reported adopting incremental strategies, starting from simplified scenarios and progressively increasing realism as needed. Similarly, they described balancing comprehensive scenario coverage against test maintainability, noting that overly elaborate test suites can become brittle and difficult to evolve; to mitigate this, they emphasized the importance of modular test design and reuse.

Additionally, they shared lessons learned from past mistakes, such as initially over-relying on simplified "happy-path" scenarios that later failed to capture production bottlenecks, or introducing

overly aggressive stress configurations that generated misleading bottlenecks unrelated to real usage. These experiences led practitioners to adopt more context-aware testing strategies, where workload design and test configuration are aligned with specific performance goals. In several cases, they emphasized that the effectiveness of these strategies depends on contextual factors such as system architecture, team maturity, and integration with CI/CD processes.

In conclusion, our results reveal a persistent gap between recommended principles and the testing practices employed in real-world projects. Practitioners generally recognize the risks of these bad practices, constraints related to time, tooling, skills, and organizational maturity often lead to pragmatic compromises. While these reflections offer qualitative, experience-driven insights rather than systematically validated best practices, we believe they provide a valuable real-world perspective on how the identified bad practices emerge, evolve, and are pragmatically addressed.

Table 13. Performance Testing Bad Practices manually analyzed with the corresponding identification criteria.

ID	Bad Practice	Identification Criteria	#Violations
TD-BP04	Ignoring think times	Verify whether at least one timer element (e.g., Constant Timer, Gaussian Random Timer) is present between any two subsequent requests.	23
TD-BP07	Improper ramp up and cool down	Analyze 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.	38
TE-BP07	Poor naming conventions	Check whether sample requests retain the default JMeter names (e.g., "HTTP Request") instead of being renamed with meaningful identifiers.	33
TA-BP03	Rigid assertions	Identify 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").	39
TA-BP04	No assertion or response validation	Verify that each sampler in the Apache JMeter test plan includes at least one response assertion.	27
TA-BP05	Excessive Listeners/Logging	Verify whether the test plan includes more than three listeners (e.g., View Results Tree, Summary Report, Aggregate Report, etc.).	6

6.2 Performance Testing Bad Practices: Are They Relevant in Practice?

The results presented in the previous sections are primarily grounded in practitioner perceptions collected through surveys and interviews. While such perception-based evidence is valuable to understand how bad practices are experienced and prioritized in practice, it does not directly demonstrate whether these practices actually occur in real-world systems. To complement this perspective, we performed a targeted manual analysis of performance testing artifacts from open-source repositories, with the goal of assessing whether the identified bad practices can be observed in practice. It is worth remarking that this analysis does not aim to provide a fully-fledged large-scale validation of the identified bad practices. Rather, it is intended as an initial empirical step to assess their presence in real-world artifacts. Indeed, a comprehensive validation would require the design and implementation of automated detection mechanisms, together with large-scale and potentially longitudinal analysis across multiple projects and system versions - analyses that are beyond the scope of this study and part of our future research agenda.

In particular, we manually inspected 33 performance test cases collected from 72 open-source projects in the E2EGit dataset [25]. This dataset was selected because it provides a curated and publicly available collection of real-world end-to-end testing artifacts, including performance-related test cases. From this dataset, we selected a subset of test cases based on the availability of performance testing artifacts and their suitability for manual inspection. We focused on six

bad practices (TD-BP04, TD-BP07, TE-BP07, TA-BP03, TA-BP04, and TA-BP05) that can be identified directly from the test artifacts without requiring external contextual information. The manual analysis was jointly conducted by the first two authors, who collaboratively inspected the selected test cases and identified occurrences of the considered bad practices. The identification process involved iterative discussion to ensure a consistent and shared interpretation of the analyzed artifacts. To support transparency and replicability, Table 13 reports the criteria used to identify each bad practice within the analyzed artifacts.

Within the analyzed subset, we identified 166 instances of the selected bad practices. The results confirm that these practices concretely occur in real-world performance test artifacts and are not limited to theoretical or perception-based observations. In particular, practices related to workload configuration, such as improper ramp-up/ramp-down settings (TD-BP07) and ignoring think times (TD-BP04), were frequently observed. Similarly, validation-related issues, including lack of response validation (TA-BP04) and rigid assertions (TA-BP03), were consistently identified across multiple test cases. Structural issues, such as the use of default request names (TE-BP07) and excessive listeners (TA-BP05), were also observed, sometimes with multiple occurrences within the same artifact. These findings provide initial empirical evidence that several of the identified bad practices manifest in real-world settings and affect different aspects of performance test design and execution. Notably, the observed practices span multiple phases of the performance testing process, suggesting that issues are not confined to a single stage but rather emerge across design, execution, and analysis activities. The presence of multiple occurrences of the same bad practice within individual artifacts further indicates that these issues are not isolated incidents, but may reflect recurring patterns in how performance tests are implemented. This observation is consistent with the perception-based results reported earlier, where several of these practices were rated as both prevalent and impactful. At the same time, the types of practices observed - particularly those related to workload configuration and validation - highlight concrete areas where performance testing may be systematically weakened. For example, improper workload modeling or missing validation logic can directly compromise the reliability and interpretability of test results, potentially leading to incorrect conclusions about system performance.

While this analysis does not constitute a comprehensive large-scale validation, it strengthens the link between practitioner perceptions and observable evidence in real-world artifacts. In particular, it suggests that the bad practices identified through the mixed-methods approach are not merely perceived issues, but correspond to concrete patterns that can be found in practice. We consider this analysis as an initial step toward a broader empirical validation, which would require automated detection and large-scale repository mining.

6.3 Further Qualitative Insights

The insights from the semi-structured interviews highlighted additional considerations that, while preliminary and confined to the sample, offer potentially valuable insights into how practitioners deal with performance testing in practice. In this section, we organize these insights by testing phase and illustrate them with representative examples.

Test Design. Across several design-related issues, practitioners emphasized the importance of aligning test scenarios with realistic usage and explicit testing goals. A recurring recommendation is to adopt an incremental approach to realism, starting from simplified workloads and progressively introducing variability. For instance, for Hardcoded or Static Input (TD-BP03), interviewees highlighted the importance of generating dynamic and diverse test data to avoid misleading caching effects or resource conflicts, especially in API-based systems. At the same time, they noted that static inputs may still be appropriate in controlled scenarios explicitly designed to evaluate

caching behavior. Similarly, issues such as Single Scenario Focus (TD-BP02) can be mitigated by diversifying test scenarios beyond the "happy path," while maintaining modularity to ensure long-term maintainability.

Test Execution. In the execution phase, practitioners highlighted the need to balance realism with feasibility. A key strategy is to make execution environments as representative as possible, while explicitly accounting for deviations when production-like setups are not available. For example, in the case of Testing in Non-production-like Environments (TE-BP01), practitioners suggested interpreting results in light of infrastructural differences rather than assuming direct comparability. Similarly, Poor Test Data Management (TE-BP04) can be mitigated through proper handling of dynamic data and request correlations, while issues such as No Proper Cleanup (TE-BP02) require disciplined setup and teardown procedures to ensure repeatable executions.

Test Analysis. For the analysis phase, practitioners emphasized strategies to improve robustness and interpretability of results. A central recommendation is to avoid relying on single test executions and instead analyze variability across multiple runs. For instance, Ignoring Test Result Variability (TA-BP09) can be mitigated by performing repeated tests and comparing distributions rather than individual outcomes. Similarly, Lack of Baseline Comparison (TA-BP06) highlights the importance of maintaining reference metrics to detect regressions, while Inadequate Resource Monitoring (TA-BP10) can be addressed by complementing performance metrics with system-level observations to identify root causes.

6.4 Implications for Researchers, Practitioners, Tool Developers, and Educators

Our findings carry implications for multiple stakeholders involved in performance testing.

For researchers, the proposed taxonomy provides a structured vocabulary for examining the quality of performance testing and highlights several open questions. These include understanding the trade-offs practitioners make between simplicity and methodological rigor, and why certain bad practices persist despite widespread awareness of their risks. Further research should investigate interventions to foster better practices—such as evaluating the effectiveness of training initiatives, developing tool support for guided test design, or creating automated detectors for problematic patterns. The absence of explicit commit-level signals when bad practices are removed also raises questions about whether developers address these issues consciously or incidentally. More broadly, the socio-technical factors that influence the adoption or persistence of poor practices, such as team roles, organizational constraints, and tool limitations, warrant further empirical investigations.

For practitioners, the results underscore the risk of overlooking issues that may appear minor, e.g., poor naming or insufficient analysis of result variability, but that gradually erode the interpretability, reliability, and maintainability of performance tests. The persistence of practices such as missing assertions or unrealistic workloads suggests that performance testing is still often approached primarily as a load-generation exercise rather than as a structured, iterative process that requires realism, validation, and continuous monitoring. Greater awareness of common pitfalls can help teams avoid design and execution choices that undermine the trustworthiness of performance assessments and, ultimately, decision-making.

The study also has implications for tool developers. The recurrence of bad practices in open-source artifacts reveals opportunities to integrate proactive guidance directly into performance testing frameworks. Several issues, such as a lack of validation, hard thresholds, improper ramp-up configurations, or inadequate handling of correlations, are amenable to automated detection. Integrating lightweight linters, warnings, or best-practice checklists, analogous to static analysis tools in programming, could reduce cognitive load for testers and decrease the likelihood of

introducing systematic errors. Improvements in usability, particularly around naming conventions, parameterization, and validation workflows, may further help mitigate recurring pitfalls.

The findings also underscore the need for more comprehensive educational efforts. Although practitioners generally recognize many poor practices, our analyses show that these issues persist in real-world artifacts, suggesting a persistent gap between theoretical knowledge and practical application. Educational programs and training should therefore emphasize not only the mechanics of executing load tests, but also the principles of designing realistic workloads, validating functional correctness, and interpreting performance results responsibly. Embedding these concepts in curricula and professional development activities could help bridge the gap between awareness and practice, fostering more reliable and methodologically sound performance testing.

7 Threats to Validity

As with any empirical study, our results are subject to several validity threats. We discuss them below, along with the main types of validity [73].

Internal Validity. For the literature review, the primary concern is the potential for bias during the selection and classification of documents. Our search relied on Google queries, and all searches were conducted in a consistent manner using an external SERP API. In this way, we reduced personalization effects and mitigated potential bias due to browsing history or prior interactions. However, search results may still be influenced by factors such as geographic location or temporal variations in indexing, potentially affecting the set of retrieved sources. 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 taxonomy is based on interpretation. To reduce subjectivity, the classification was iteratively discussed among authors and validated in multiple rounds.

Construct Validity. For the survey, construct validity may be threatened by the phrasing and interpretation of the questions. Ambiguous wording or technical jargon could lead to misunderstandings, while self-reported answers may not accurately reflect actual practices. To mitigate these threats, each bad practice was presented not only by its label but also through a short explanatory description written in clear and accessible language, complemented with brief examples to support a consistent interpretation across participants. In addition, we followed established guidelines for survey design in software engineering and piloted the questionnaire with a group of researchers and practitioners. The pilot phase allowed us to identify and refine potentially ambiguous formulations. For instance, descriptions of practices such as "Poor test data management" were revised to explicitly distinguish between concepts like test data variation and parameterization, thereby reducing the risk of misinterpretation. Despite these mitigation steps, differences in participants' backgrounds and levels of expertise may still have influenced how certain concepts were understood, and thus some degree of interpretation bias cannot be entirely excluded.

Given the subjective nature of survey-based data, a potential concern is that the aggregated results may be influenced by systematic response behaviors, such as participants consistently agreeing or disagreeing with most items, or relying on extreme values across the questionnaire. Such patterns could bias the interpretation of agreement, severity, and prevalence of the identified bad practices. To investigate this aspect, we analyzed individual response profiles across the three dimensions considered in the study (agreement, severity, and prevalence). Specifically, we examined whether any participant consistently selected the same extreme Likert-scale value (i.e., 1 or 5) for a large proportion of the bad practices. We defined a threshold corresponding to more than 80% of the items (i.e., 24 out of 29 practices) within each dimension. In addition, we assessed whether any

participant exhibited a systematic tendency to disagree (values 1–2) or agree (values 4–5) with most of the practices. The analysis shows that no participant met the extreme-response criterion in any of the considered dimensions. Furthermore, no respondent consistently disagreed or agreed with more than 25 practices overall. This indicates that participants did not exhibit uniform or polarized response behaviors across the questionnaire. These findings suggest that the aggregate results are not driven by a small subset of participants with systematic agreement or disagreement patterns. Instead, the responses reflect a more nuanced distribution across participants, supporting the robustness of the observed trends. This additional analysis increases confidence that the reported findings are not significantly affected by response bias or extreme answering strategies.

Another potential threat concerns the completeness of the proposed taxonomy, as it is possible that relevant bad practices were not identified during the study. To partially mitigate this risk, participants were given the opportunity to suggest additional bad practices through an open-ended question included at the end of the survey. In addition, during the semi-structured interviews, participants were explicitly encouraged to reflect on the completeness of the taxonomy, to challenge existing practices, and to suggest potential missing ones. However, neither the open-ended responses nor the interviews revealed additional distinct practices requiring an extension of the taxonomy. Despite these mitigation steps, the taxonomy should not be considered exhaustive. Rather, it represents an empirically grounded synthesis derived from gray literature analysis and practitioner feedback. It is therefore possible that additional bad practices may emerge in different contexts, domains, or experience levels. Future studies may refine, extend, or challenge the proposed taxonomy by incorporating broader datasets and more diverse practitioner populations.

An additional point of discussion concerns the data cleaning process, which may have failed to exclude responses from participants who did not take the task seriously or lacked sufficient expertise to complete the survey. In this respect, the paper's first author reviewed each response and validated it. The second author then confirmed the operations performed. Such a double-check gives us confidence that we have not overlooked answers that might have biased our conclusions.

For the interviews, construct validity is influenced by the framing of questions and the potential for interviewer bias in follow-up questions. To mitigate this, we designed a semi-structured interview protocol, keeping questions open-ended and carefully avoiding leading participants toward specific answers. All interview recordings and transcripts (with personal information removed) are included in our replication package [23] to ensure transparency and to allow others to re-examine our interpretations and replicate our study. Additional threats to validity may arise from how the concept of bad practice is interpreted and measured. Although we provided participants with descriptions derived from the gray literature, practitioners may still interpret what constitutes a bad practice differently depending on their organizational context, system type, and testing goals (e.g., user-oriented load testing versus stress testing). As a result, agreement, severity, and prevalence ratings may reflect context-dependent judgments rather than universally applicable assessments. Moreover, our survey and interviews rely on self-reported perceptions, which capture practitioners' beliefs and experiences but may not always correspond to actual performance testing practices enacted in industrial settings. To mitigate these threats, we grounded the taxonomy in a large corpus of practitioner-oriented sources, provided clear descriptions for each bad practice, and complemented survey data with follow-up interviews aimed at clarifying participants' interpretations and contextual assumptions.

External Validity. Regarding the generalizability of our findings, the gray literature review is limited to publicly accessible practitioner platforms and may not reflect private discussions or industry-internal practices. Similarly, the survey respondents, although diverse, may not be representative of all developers engaged in performance testing, particularly those in regulated or

proprietary environments. 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 and practitioner interviews, would be valuable to deepen our perspective and validate our findings in other settings. Furthermore, the frequent occurrence of bad practices even in this limited subset suggests that expanding to larger or more diverse datasets would likely increase the absolute number of observed violations, thereby reinforcing rather than weakening our findings.

Additionally, the survey engaged 22 practitioners; this sample size may not fully capture the diversity of performance testing practices across different organizations, domains, or project contexts. As a result, the identified bad practices and their perceived criticality may not be representative of all software performance testing settings. We acknowledge this limitation. At the same time, it is worth noting that our participants belong to a hard-to-reach practitioner population with substantial experience in performance testing, which supports the relevance and depth of the collected insights. Similarly, we were only able to interview three practitioners, which further limits the generalizability of the qualitative findings. Nevertheless, these interviews were intended to complement the survey results by providing deeper contextual understanding rather than statistical representativeness. To partially mitigate this limitation, we triangulated the survey and interview findings with a preliminary manual analysis of real-world performance testing artifacts, providing an initial form of empirical grounding for the identified bad practices. While this additional step strengthens the connection between perception-based evidence and observed practices, it does not constitute a comprehensive, large-scale validation. Further replications would be desirable to confirm and extend our findings across different industrial contexts and practitioner populations. To ease replication and support transparency, our online appendix [23] includes the full survey instrument, the interview guide, and the taxonomy of identified bad practices.

Another potential threat concerns the experience distribution of the survey participants. A substantial portion of respondents reported limited experience in performance testing, which may influence their assessment of the severity and prevalence of the identified bad practices. In particular, less experienced practitioners may have limited exposure to complex performance testing scenarios, potentially affecting the generalizability of these perceptions. More specifically, they may overestimate the severity of certain issues due to limited exposure to diverse contexts or underestimate long-term or systemic implications that typically emerge with more extensive field experience. At the same time, this distribution reflects realistic development contexts, where performance testing activities might be carried out by practitioners such as software developers or QA engineers, rather than exclusively by dedicated performance testing specialists. Therefore, our findings capture a broader practitioner perspective, although they may not fully represent the views of highly specialized performance testing experts.

To mitigate this limitation, we complemented the survey with semi-structured interviews involving practitioners with substantially more experience in performance testing, whose background ranges from several years to more than a decade in performance and quality engineering roles. These interviews provide deeper qualitative insights into how more experienced practitioners interpret the severity, prevalence, and contextual relevance of the identified bad practices, allowing us to better contextualize and contrast the survey findings across different levels of experience. In addition, we complemented this analysis with a preliminary manual inspection of real-world artifacts, providing an initial empirical grounding of the results. Nevertheless, the results related to severity and prevalence should be interpreted with caution, as they reflect the perspectives of the sampled practitioners rather than a consensus across all experience levels. Future work should involve larger and more experience-diverse samples, particularly including senior performance engineers and architects, to further assess how perceptions of severity and prevalence may vary across different career stages and industrial contexts.

Conclusion Validity. Finally, threats to conclusion validity concern the inferences we draw from the data. Part of our interpretation is based on manual classification and subjective judgment, which may introduce bias. This 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 as part of our replication package appendix [23] may serve as a base for further replications.

Finally, a potential threat concerns the interpretation of the frequency with which bad practices are reported across the analyzed gray literature sources. While we report the number of documents discussing each practice to provide an indication of how frequently it is mentioned, this measure should be interpreted with caution. Gray literature sources may vary substantially in terms of scope, depth, and level of detail, and therefore a higher number of documents does not necessarily reflect a greater importance or impact of a given practice. Rather, these counts provide an indicative view of how often practices are discussed within the collected sources, and are not intended as a precise measure of their relative significance.

8 Conclusion

The ultimate goal of our work was to develop a deeper understanding and provide a structured foundation of bad practices in software performance testing. Using a two-phase mixed-method approach, we analyzed evidence from gray literature, surveys, and semi-structured interviews to capture the breadth and depth of bad practices affecting performance testing activities.

In detail, based on an analysis of 415 documents from a gray literature review, we developed a taxonomy that systematically describes and categorizes 29 recurring bad practices across the phases of test design, execution, and analysis. Afterwards, we surveyed 22 practitioners and conducted semi-structured interviews with three additional practitioners, which together contextualized the identified practices. Overall, the practitioners agreed with the taxonomy, though perceptions of severity and prevalence varied.

To sum up, our paper provides the following contributions:

- A taxonomy that defines bad practices in software performance testing;
- Empirical evidence on practitioners' experiences and perceptions of such bad practices;
- A publicly available replication package including all sources, questions, and results used in our study, to ensure transparency and foster future research [23].

The main conclusions of this study represent the input for future research rather than providing direct solutions. First, we plan to design and implement automated tools to support the identification of bad practices in software performance testing. This will require the formalization of the identified bad practices into precise and operational criteria, as well as the development of automated detection mechanisms capable of analyzing heterogeneous performance testing artifacts. Such tools would enable large-scale empirical studies aimed at assessing the prevalence, evolution, and potential impact of these bad practices across projects and over time. In turn, this would allow us to complement the perception-based and qualitative evidence presented in this work with quantitative insights derived from real-world software repositories. Furthermore, we envision extending this line of research toward the design and evaluation of tool-supported techniques not only for detecting but also for refactoring performance testing bad practices. In this context, the taxonomy proposed in this paper represents a necessary foundation for enabling prescriptive and evaluative approaches, supporting practitioners in systematically improving the quality of their performance testing artifacts.

We also intend to explore socio-technical factors (e.g., team roles and tool limitations) to understand the conditions under which such bad practices emerge and how they can be mitigated. Finally, we aim to investigate how bad practices manifest themselves in particular performance testing scenarios, such as load, stress, or smoke testing, and across different classes of software systems. Such analyses would enable the refinement of the proposed taxonomy and support the development of more targeted guidelines and mitigation strategies tailored to specific contexts.

Acknowledgments

This work has been partially supported by the Italian PNRR MUR project Future Artificial Intelligence Research - FAIR (PE0000013-FAIR) and the European HORIZON-KDT-JU-2023-2-RIA research project MATISSE "Model-based engineering of Digital Twins for early verification and validation of Industrial Systems" (grant 101140216-2, KDT232RIA_00017).

References

[1] Rabiya Abbas, Zainab Sultan, and Shahid Nazir Bhatti. 2017. Comparative analysis of automated load testing tools: Apache JMeter, Microsoft Visual Studio (TFS), LoadRunner, Siege. In *2017 international conference on communication technologies (comtech)*. IEEE, 39–44.

[2] Angelo Afeltra, Alfonso Cannavale, Fabiano Pecorelli, Valeria Pontillo, and Fabio Palomba. 2024. A Large-Scale Empirical Investigation Into Cross-Project Flaky Test Prediction. *IEEE Access* (2024).

[3] Dr SM Afroz, N Elezabeth Rani, and N Indira Priyadarshini. 2011. Web application-A study on comparing software testing tools. *International Journal of Computer Science and Telecommunications* 2, 3 (2011), 1–6.

[4] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D Newman, Abdullatif Ghallab, and Stephanie Ludi. 2021. Test smell detection tools: A systematic mapping study. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*. 170–180.

[5] Karen M Alsante, Linda Martin, and Steven W Baertschi. 2003. A stress testing benchmarking study. *Pharmaceutical technology* 27, 2 (2003), 60–73.

[6] Dorine Andrews, Blair Nonnecke, and Jennifer Preece. 2007. Conducting research on the internet:: Online survey design, development and implementation guidelines. (2007).

[7] Anuja Arora and Madhavi Sinha. 2012. Web application testing: A review on techniques, tools and state of art. *International Journal of Scientific & Engineering Research* 3, 2 (2012), 1.

[8] Ermanno Battista, Sergio Di Martino, Sergio Di Meglio, Fabio Scippacercola, and Luigi Libero Lucio Starace. 2023. E2E-Loader: A Framework to Support Performance Testing of Web Applications. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 351–361.

[9] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2012. 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, 56–65.

[10] Cecilio Cannavacciuolo and Leonardo Mariani. 2022. Smoke testing of cloud systems. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 47–57.

[11] Lidiany Cerqueira, João Pedro Bastos, Danilo Neves, Glauco Carneiro, Rodrigo Spinola, Sávio Freire, Jose Santos, and Manoel Mendonça. 2025. Exploring Empathy in Software Engineering: Insights from a Grey Literature Analysis of Practitioners' Perspectives. *ACM Trans. Softw. Eng. Methodol.* (July 2025). doi:10.1145/3748721 Just Accepted.

[12] Vandana Chandel, Shilpa Patial, and Sonal Guleria. 2013. Comparative Study of Testing Tools: Apache JMeter and Load Runner. *International Journal of Computing and Corporate Research* 3, 3 (2013), 1–7.

[13] Jinfu Chen, Weiyi Shang, Ahmed E Hassan, Yong Wang, and Jiangbin Lin. 2019. 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, 669–681.

[14] Tse-Hsun Chen, Mark D. Syer, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2017. 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)*. 243–252. doi:10.1109/ICSE-SEIP.2017.26

[15] J. Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[16] Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. 2021. Web Application Testing: Using Tree Kernels to Detect Near-duplicate States in Automated Model Inference. In *Proceedings of the 15th ACM/IEEE*

International Symposium on Empirical Software Engineering and Measurement (ESEM). 1–6.
[17] John W Creswell. 1999. Mixed-method research: Introduction and application. In *Handbook of educational policy*. Elsevier, 455–472.
[18] Mariela Curiel and Ana Pont. 2018. Workload generators for web-based systems: Characteristics, current status, and challenges. *IEEE Communications Surveys & Tutorials* 20, 2 (2018), 1526–1546.
[19] Sergio Di Meglio and Luigi Libero Lucio Starace. 2024. Evaluating Performance and Resource Consumption of REST Frameworks and Execution Environments: Insights and Guidelines for Developers and Companies. *IEEE Access* (2024).
[20] Sergio Di Meglio, Luigi Libero Lucio Starace, and Sergio Di Martino. 2023. Starting a New REST API Project? A Performance Benchmark of Frameworks and Execution Environments.. In *IWSM-Mensura*.
[21] Sergio Di Meglio, Luigi Libero Lucio Starace, and Sergio Di Martino. 2025. 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.
[22] Sergio Di Meglio, Luigi Libero Lucio Starace, and Sergio Di Martino. 2026. Web app performance testing in industrial contexts: Supporting workload generation with E2E-Loader++. *Journal of Systems and Software* 232 (2026), 112684. doi:10.1016/j.jss.2025.112684
[23] Sergio Di Meglio, Luigi Libero Lucio Starace, Valeria Pontillo, Luana Martins, Fabio Palomba, and Dario Di Nucci. 2025. A Taxonomy of Bad Practices in Web Performance Testing: Insights from Grey Literature and Practitioner Validation — zenodo.org. https://zenodo.org/records/19929942. [Accessed 30-04-2026].
[24] Sergio Di Meglio, Luigi Libero Lucio Starace, Valeria Pontillo, Ruben Opdebeeck, Coen De Roover, and Sergio Di Martino. 2025. E2EGit: A Dataset of End-to-End Web Tests in Open Source Projects. In *Proceedings of the 22nd International Conference on Mining Software Repositories*. 836–840. doi:10.1109/MSR66628.2025.00121
[25] Sergio Di Meglio, Luigi Libero Lucio Starace, Valeria Pontillo, Ruben Opdebeeck, Coen De Roover, and Sergio Di Martino. 2025. 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, 10–15.
[26] Sergio Di Meglio, Luigi Libero Lucio Starace, Valeria Pontillo, Ruben Opdebeeck, Coen De Roover, and Sergio Di Martino. 2025. 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.
[27] Sergio Di Meglio, Luigi Libero Lucio Starace, Valeria Pontillo, Ruben Opdebeeck, Coen De Roover, and Sergio Di Martino. 2026. Investigating the adoption and maintenance of web GUI testing: Insights from GitHub repositories. *Information and Software Technology* 189 (2026), 107928. doi:10.1016/j.infsof.2025.107928
[28] Kit Eaton. 2012. How One Second Could Cost Amazon $1.6 Billion In Sales. https://web.archive.org/web/20221006004855/https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales Last accessed: Oct. 24, 2022.
[29] Tommaso Fulcini, Giacomo Garaccione, Riccardo Coppola, Luca Ardito, and Marco Torchiano. 2022. 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*. 17–24.
[30] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. 2016. The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature. In *Proceedings of the 20th international conference on evaluation and assessment in software engineering*. 1–6.
[31] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and software technology* 106 (2019), 101–121.
[32] Vahid Garousi and Barış Küçük. 2018. Smells in software test code: A survey of knowledge in industry and academia. *Journal of systems and software* 138 (2018), 52–81.
[33] Giammaria Giordano, Gerardo Festa, Gemma Catolino, Fabio Palomba, Filomena Ferrucci, and Carmine Gravino. 2024. On the adoption and effects of source code reuse on defect proneness and maintenance effort. *Empirical Software Engineering* 29, 1 (2024), 20.
[34] Giammaria Giordano, Giulia Sellitto, Aurelio Sepe, Fabio Palomba, and Filomena Ferrucci. 2023. The yin and yang of software quality: On the relationship between design patterns and code smells. In *2023 49th Euromicro conference on software engineering and advanced applications (SEAA)*. IEEE, 227–234.
[35] Giovanni Grano, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Harald C Gall. 2019. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software* 156 (2019), 312–327.
[36] T. Hall and V. Flynn. 2001. Ethical issues in software engineering research: a survey of current practice. *Empirical Software Engineering* 6, 4 (2001), 305–317.
[37] Siw Elisabeth Hove and Bente Anda. 2005. Experiences from conducting semi-structured interviews in empirical software engineering research. In *11th IEEE International Software Metrics Symposium (METRICS'05)*. IEEE, 10–pp.

[38] Alexandru Iosup, Simon Ostermann, M Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. 2011. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed systems* 22, 6 (2011), 931–945.

[39] Zhen Ming Jiang and Ahmed E. Hassan. 2015. A Survey on Load Testing of Large-Scale Software Systems. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1091–1118. doi:10.1109/TSE.2015.2445340

[40] Mitashree Kalita and Tulshi Bezboruah. 2011. Investigation on performance testing and evaluation of prewebd: A. net technique for implementing web application. *IET software* 5, 4 (2011), 357–365.

[41] Barbara A Kitchenham and Shari Lawrence Pfleeger. 2002. Principles of survey research part 2: designing a survey. *ACM SIGSOFT Software Engineering Notes* 27, 1 (2002), 18–20.

[42] Diwakar Krishnamurthy, Jerome A Rolia, and Shikharesh Majumdar. 2006. A synthetic workload generation technique for stress testing session-based systems. *IEEE Transactions on Software Engineering* 32, 11 (2006), 868–882.

[43] Philipp Leitner and Cor-Paul Bezemer. 2017. An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (L'Aquila, Italy) *(ICPE '17)*. Association for Computing Machinery, New York, NY, USA, 373–384. doi:10.1145/3030207.3030213

[44] Haroon Malik, Bram Adams, and Ahmed E. Hassan. 2010. Pinpointing the Subsystems Responsible for the Performance Deviations in a Load Test. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*. 201–210. doi:10.1109/ISSRE.2010.43

[45] Vincenzo Marrazzo. [n. d.]. How to handle correlation in jmeter: Blazemeter by perforce. https://www.blazemeter.com/blog/correlation-in-jmeter#what

[46] Christopher Marshall, Pearl Brereton, and Barbara Kitchenham. 2015. Tools to support systematic reviews in software engineering: a cross-domain survey using semi-structured interviews. In *Proceedings of the 19th international conference on evaluation and assessment in software engineering*. 1–6.

[47] Luana Martins, Heitor Costa, Márcio Ribeiro, Fabio Palomba, and Ivan Machado. 2023. Automating test-specific refactoring mining: A mixed-method investigation. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 13–24.

[48] Luana Martins, Valeria Pontillo, Heitor Costa, Filomena Ferrucci, Fabio Palomba, and Ivan Machado. 2025. Test code refactoring unveiled: where and how does it affect test code quality and effectiveness? *Empirical Software Engineering* 30, 1 (2025), 27.

[49] M. L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.

[50] Courtney A McKim. 2017. The value of mixed methods research: A mixed methods study. *Journal of mixed methods research* 11, 2 (2017), 202–222.

[51] Jefferson Seide Molléri, Kai Petersen, and Emilia Mendes. 2016. Survey guidelines in software engineering: An annotated review. In *Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement*. 1–6.

[52] Karen H Morin. 2013. Value of a pilot study. *Journal of Nursing Education* 52, 10 (2013), 547–548.

[53] Shiva Nejati, Stefano Di Alesio, Mehrdad Sabetzadeh, and Lionel Briand. 2012. Modeling and analysis of CPU usage in safety-critical embedded systems to support stress testing. In *Model Driven Engineering Languages and Systems: 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30–October 5, 2012. Proceedings 15*. Springer, 759–775.

[54] Ramakanth P., Kalpan Bhargav, and M Tech. 2012. A Survey on Performance Testing Approaches of Web Application and Importance of WAN Simulation in Performance Testing. *International Journal on Computer Science and Engineering* 4 (05 2012).

[55] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. 2018. Automatic test smell detection using information retrieval techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 311–322.

[56] Shravan Pargaonkar. 2023. A comprehensive review of performance testing methodologies and best practices: software quality engineering. *International Journal of Science and Research (IJSR)* 12, 8 (2023), 2008–2014.

[57] Fabiano Pecorelli, Giovanni Grano, Fabio Palomba, Harald C Gall, and Andrea De Lucia. 2024. Toward granular search-based automatic unit test case generation. *Empirical Software Engineering* 29, 4 (2024), 71.

[58] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. 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*. 1650–1654.

[59] Valeria Pontillo, Dario Amoroso d'Aragona, Fabiano Pecorelli, Dario Di Nucci, Filomena Ferrucci, and Fabio Palomba. 2024. Machine learning-based test smell detection. *Empirical Software Engineering* 29, 2 (2024), 55.

[60] Valeria Pontillo, Luana Martins, Ivan Machado, Fabio Palomba, and Filomena Ferrucci. 2025. An empirical investigation into the capabilities of anomaly detection approaches for test smell detection. *Journal of Systems and Software* 222 (2025), 112320.

[61] Valeria Pontillo, Fabio Palomba, and Filomena Ferrucci. 2022. Static test flakiness prediction: How far can we go? *Empirical Software Engineering* 27, 7 (2022), 187.

[62] Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci, and Fabio Palomba. 2025. When code smells meet ML: on the lifecycle of ML-specific code smells in ML-enabled systems. *Empirical Software Engineering* 30, 5 (2025), 139.

[63] Filippo Ricca and Andrea Stocco. 2021. Web test automation: Insights from the grey literature. In *International Conference on Current Trends in Theory and Practice of Informatics*. Springer, 472–485.

[64] Sanjay Tyagi Rina. 2013. A comparative study of performance testing tools. *International Journal of Advanced Research in Computer Science and Software Engineering (IJARCSSE)* 3, 5 (2013), 1300–1307.

[65] Renaud Rwemalika, Sarra Habchi, Mike Papadakis, Yves Le Traon, and Marie-Claude Brasseur. 2023. Smells in system user interactive tests. *Empirical Software Engineering* 28, 1 (2023), 20.

[66] Mahnaz Shams, Diwakar Krishnamurthy, and Behrouz Far. 2006. A Model-Based Approach for Testing the Performance of Web Applications. In *Proceedings of the 3rd International Workshop on Software Quality Assurance* (Portland, Oregon) *(SOQUA '06)*. Association for Computing Machinery, New York, NY, USA, 54–61. doi:10.1145/1188895.1188909

[67] Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2015. Automated Detection of Performance Regressions Using Regression Models on Clustered Performance Counters. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (Austin, Texas, USA) *(ICPE '15)*. Association for Computing Machinery, New York, NY, USA, 15–26. doi:10.1145/2668930.2688052

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

[69] Elvys Soares, Márcio Ribeiro, Rohit Gheyi, Guilherme Amaral, and André Santos. 2022. Refactoring test smells with junit 5: Why should developers keep up-to-date? *IEEE Transactions on Software Engineering* 49, 3 (2022), 1152–1170.

[70] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the relation of test smells to software code quality. In *2018 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 1–12.

[71] Muhammad Dhiauddin Mohamed Suffian and Fairul Rizal Fahrurazi. 2012. Performance testing: Analyzing differences of response time between performance testing tools. In *2012 International Conference on Computer & Information Science (ICCIS)*, Vol. 2. IEEE, 919–923.

[72] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 4–15.

[73] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2012. *Experimentation in software engineering*. Vol. 236. Springer.

[74] Bo Yang, Zhenchang Xing, Xin Xia, Chunyang Chen, Deheng Ye, and Shanping Li. 2021. UIS-hunter: Detecting UI design smells in Android apps. In *2021 IEEE/ACM 43rd international conference on software engineering: companion proceedings (ICSE-companion)*. IEEE, 89–92.

[75] Keith Yorkston. 2021. *Performance Testing Tasks*. Apress, Berkeley, CA, 195–354. doi:10.1007/978-1-4842-7255-8_4