On the Practical Adoption of a Static Performance Anti-Pattern Detector: An Industrial Case Study

Lizhi Liao
University of Guelph
Guelph, Canada
lizhi.liao@uoguelph.ca

Weiyi Shang
University of Waterloo
Waterloo, Canada
wshang@uwaterloo.ca

Catalin Sporea, Andrei Toma, Sarah Sajedi
ERA Environmental
Montréal, Canada

***Abstract*—Performance anti-patterns, which are recurring design and implementation flaws that lead to potential performance issues (e.g., high latency, resource exhaustion, or pool scalability), have gained significant attention in recent years. Identifying and mitigating these anti-patterns early in development is also increasingly recognized as an effective means of proactive performance assurance. Despite considerable research on performance anti-patterns, their practical adoption in industrial contexts remains limited. In addition, practitioners' perceptions, expectations, and experiences with these techniques in real-world settings are still insufficiently explored. To bridge this gap, we conduct an industrial case study in which we first investigate the perception and expectations on performance anti-pattern detection from the perspective of real-world practitioners. Based on these valuable insights, we then design and develop a static performance anti-pattern detector to proactively assist our industrial collaborator in ensuring and improving the performance of their large-scale software system. Furthermore, we gather feedback from developers who have used our detector and discuss their perspectives on its usability, effectiveness, and areas for future enhancement. By addressing this gap, we aim to provide practical insights into how practitioners perceive, expect, and experience performance anti-pattern detection. We anticipate that our work can contribute to improving performance anti-pattern detection and facilitate its wider adoption in real-world industrial contexts.**

***Index Terms*—Software performance, performance anti-pattern, static and dynamic analysis, industrial case study**

I. Introduction

Performance issues pose a critical challenge to modern software systems, since they directly impact system efficiency, responsiveness, and reliability [1]–[3]. If left unaddressed, such issues can lead to severe repercussions, like increased operational costs, compromised user satisfaction, or even business losses [4]–[6]. This is particularly crucial for large-scale enterprise applications, like Facebook, LinkedIn, and YouTube, which provide essential services to a large client base globally. Given the critical nature of these services, ensuring high performance is of utmost importance [7], [8].

To ensure performance, performance assurance activities are typically conducted to identify and eliminate performance issues in large software systems [9], [10]. In particular, current practices often rely on system performance testing, which is conducted after the system has been built, integrated, and deployed to the testing or production environment to identify performance issues [11]–[14]. However, such reactive approaches typically incur significant time and resource overheads, as performance issues are detected, located, and resolved only in the late stages of development (e.g., during final system testing), often resulting in project delays and increased costs [15]–[17].

In recent years, extensive research has emphasized the importance of detecting and resolving software performance anti-patterns, as it enables a proactive approach to identifying performance issues early in the development process [7], [18]–[24]. These recurring design and implementation flaws, such as inefficient algorithms [18] or poor caching strategies [7], can significantly degrade performance if not addressed in time. By uncovering these flaws early in development, developers can address potential issues before they actually affect system performance and user experience. However, despite advances in research, the practical adoption of performance anti-pattern detection in real-world industrial contexts remains limited. Practitioners often exhibit limited awareness of performance anti-patterns during the code writing or review process, and as a result, these issues often go undetected until after deployment [25]–[27]. Furthermore, there is also a lack of prior research exploring the real-world perceptions, expectations, or providing sufficient practical feedback on adopting performance anti-pattern detection in industrial settings. This gap between academic research and industrial practice poses a significant barrier to the development of performance anti-pattern detection solutions that align closely with real-world needs, thereby limiting their broader adoption in practice.

In this paper, we aim to address this gap by conducting an industrial case study on the practical adoption of performance anti-pattern detection. In particular, we begin by performing a preliminary study to investigate the experiences, concerns, and expectations of practitioners when adopting performance anti-pattern detection in real-world practices. Building upon these valuable insights, we then design and develop a performance anti-pattern detector seamlessly integrated into the existing workflow of our industrial collaborator, specifically within the developers' integrated development environment (IDE), to proactively assist in enhancing the performance of their large-scale software system. Our detector combines static code analysis with dynamic impact analysis, offering a convenient, efficient, and effective solution for identifying and resolving performance anti-patterns early in development. Furthermore, we collect and analyze the valuable feedback from real-world developers who have used our detector, highlighting

their perspectives on its usability, effectiveness, and potential areas for future enhancement. We believe that our study can provide software practitioners and researchers with helpful insights into improving performance anti-pattern detection and its broader adoption into real-world industrial practices.

The main contributions of this study are twofold:

- We conduct a preliminary study on the experiences, concerns, and expectations of practitioners regarding performance anti-pattern detection. It aims to provide a comprehensive understanding of the factors limiting practical adoption and offers insights into real-world needs.
- Building upon the insights, we perform an industrial case study in which we design and develop a performance anti-pattern detector in a real-world context with our industrial collaborator. We also evaluate our detector's usability and effectiveness based on practitioner feedback, as well as identify key areas for further improvements.

Paper organization. Section II presents our preliminary study on practitioners' experiences, concerns, and expectations in adopting performance anti-pattern detection. Section III details our industrial case study, including the industrial background and design of our performance anti-pattern detector. Section IV discusses the adoption results and real-world feedback from developers regarding our detector. Related work and threats to validity are discussed in Section V and Section VI, respectively. Finally, Section VII concludes the paper.

II. Preliminary study on practitioner perspectives

Performance anti-pattern detection aims to automate the identification of common performance issues and improve software performance. Despite considerable research (e.g., [7], [28]–[32]) on performance anti-pattern detection in recent years, the adoption of these techniques in real-world software engineering practices remains scarce [25]–[27]. This lack of widespread adoption indicates that current research may not fully align with the practical requirements and expectations of practitioners in addressing performance anti-patterns. Therefore, in this section, we aim to fill this gap by exploring practitioners' perspectives on performance anti-pattern detection and their expectations. In particular, our study focuses on addressing the following preliminary study questions (PQs):

- **PQ-01: What are practitioners' current approaches and perspectives on performance anti-pattern detection?** We aim to understand how practitioners currently detect performance anti-patterns and their perspectives on the concerns and benefits of performance anti-pattern detection. Understanding these aspects will help identify gaps in existing practices and better recognize practitioners' expectations.
- **PQ-02: Who are the primary users and focus areas for performance anti-pattern detection?** We seek to explore the roles (e.g., developers or testers) and expertise levels (e.g., junior or senior) of potential users, as well as the types of performance anti-patterns that are of the most concern. Gaining these key insights will help tailor detection tools to specific roles, refine the focus areas, and optimize their effectiveness in real-world applications.
- **PQ-03: What are practitioners' expectations for performance anti-pattern detection?** We intend to identify the capabilities practitioners prioritize in performance anti-pattern detection, such as detection speed, integration with existing workflows, detection granularity, and expected features. Understanding these perspectives will help develop user-friendly and efficient tools that better align with practitioners' needs and improve adoption.

A. Study design

To answer our PQs, we adopt a survey-based approach to gather insights from a broad community of software practitioners. The survey aims to capture their experiences, concerns, and expectations in performance anti-pattern detection.

Questionnaire development. We follow Kitchenham and Pfleeger's guidelines [33] for personal opinion surveys and design our online survey using Microsoft Forms [34]. The first author is responsible for formulating the initial draft of the questionnaire, while the remaining authors validate and improve it. To provide participants with a shared understanding, the survey begins with a brief background on performance anti-patterns and the context of this study. Our questionnaire consists of closed-ended questions (single-choice or multiple-choice), with each question also allowing optional open-ended responses to capture qualitative insights alongside quantitative data. Our questions are informed by prior studies [35], [36], and each is explicitly designed to address a specific PQ. Furthermore, before launching the survey, we conduct a pilot study with three participants who go through the survey and suggest improvements, like refining the wording, simplifying certain questions for clarity, and adjusting some response options to better capture the diversity of participant views. Based on feedback, we then make revisions to the questionnaire. The list of questions included in our survey is detailed in Table I.

Participant selection. In our study, we initially invite participants from our personal networks with attention to ensuring a diverse and relevant sample of software practitioners. For example, we include both senior practitioners, who have extensive experience in the field, and junior ones, who may provide fresh perspectives. We also invite participants through the connections of initial respondents and relevant professional platforms such as LinkedIn and X. In total, 97 potential participants were invited, and 22 completed our questionnaire, resulting in a completion rate of 22.7%. To facilitate research sharing and transparency, the full survey questionnaire and participants' responses are shared in our replication package [37].

B. Data analysis

We employ a mixed-methods approach to analyze the survey results. In particular, we conduct quantitative analysis of closed-ended responses by examining their distributions to identify trends. It is worth noting that participants could select multiple options for some questions; therefore, the total percentages may exceed 100%. We also perform qualitative

TABLE I
SURVEY QUESTIONNAIRE ON STUDYING PRACTITIONER PERSPECTIVES ON PERFORMANCE ANTI-PATTERN DETECTION

ID	PQ	Question
SQ-01	PQ-01	What is your current approach to the performance anti-pattern detection and localization?
SQ-02	PQ-01	How long does it usually take you to detect and locate a performance anti-pattern in hours?
SQ-03	PQ-01	What benefits do you think automated performance anti-pattern detection tools will provide?
SQ-04	PQ-01	What harms do you think automated performance anti-pattern detection tools will provide?
SQ-05	PQ-01	What concerns do you think about adopting automated performance anti-pattern detection tools?
SQ-06	PQ-01	Will you try to use a performance anti-pattern detection tool if it is available?
SQ-07	PQ-02	Which category of software engineers do you believe would benefit the most from the performance anti-pattern detection?
SQ-08	PQ-02	For which software development role do you think performance anti-pattern detection tools would be most advantageous?
SQ-09	PQ-02	Which areas do you think performance anti-pattern detection tools should primarily focus on?
SQ-10	PQ-02	Which type of anti-patterns do you prefer to detect in performance anti-pattern detection tools?
SQ-11	PQ-02	Which error (false positive or false negative) do you consider to be more severe than the other?
SQ-12	PQ-03	What is the expected time (in minutes) for the tool to detect performance anti-patterns in a project?
SQ-13	PQ-03	How would you like to integrate the performance anti-pattern detection tools in your project?
SQ-14	PQ-03	What level of granularity would best suit your development process when adopting performance anti-pattern detection tools?
SQ-15	PQ-03	How often would you like performance anti-pattern detection to occur?
SQ-16	PQ-03	How would you prefer performance anti-pattern detection to be applied?
SQ-17	PQ-03	Please prioritize the expected features for performance anti-pattern detection tools.

Note 1: “SQ” stands for “Survey Question”, and “PQ” represents the “Preliminary Study Question” that each survey question is designed to address.
Note 2: For brevity, background questions, as well as all answer options, are not presented here and can be found in our replication package [37].

analysis of open-ended inputs by categorizing and interpreting responses to gain additional insights beyond predefined options. Moreover, we explore potential relationships between different aspects. For instance, we analyze which types of performance anti-patterns practitioners prefer to detect and how these preferences vary according to factors such as participants’ experience levels. By integrating these analysis methods, we ensure a comprehensive understanding of the perceptions and expectations of practitioners in performance anti-pattern detection, capturing both common perspectives and diverse insights.

C. Results and implications

Demographics and background. According to the survey results, the majority of respondents are male, accounting for 72.7% of the total sample, while 22.7% are female, and 4.5% identify as non-binary or other gender options. The respondents are aged from 25 to 59 years old, with a median age of 40. Regarding work experience, respondents generally have substantial experience in software engineering, ranging from 2 to 30 years, with a mean of around 13 years. The participants also display a wide range of roles in the software engineering domain. The majority of respondents identify their role as developer, accounting for 68.2%, followed by software development manager (36.4%), architect (22.7%), tester (18.2%), and researcher (18.2%). Respondents mainly engage in backend development (68.2%), followed by web development (54.5%), with relatively smaller proportions involved in mobile development (22.7%), research projects (18.2%), and AI/ML/data science projects (13.6%). Overall, this variety in backgrounds, experience levels, and work characteristics further ensures the diversity of our survey participants.

PQ-01: What are practitioners’ current approaches and perspectives on performance anti-pattern detection? Our first PQ intends to investigate practitioners’ current practices in addressing performance anti-patterns and their perspectives on adopting automated detection tools in real-world contexts.

According to the survey results, the primary approach used by respondents for detecting performance anti-patterns is manual code review, employed by 81.8% of the participants. The second most common approach is analyzing performance testing results (e.g., logs, traces, or performance metrics), used by 50.0% of the participants. Furthermore, 45.5% of participants engage in discussions with peers, while 31.8% utilize automated tools (e.g., GitHub Copilot [38]) to avoid performance anti-patterns when coding. A small portion (9.1%) have not specifically dealt with performance-related issues before (**F[1]-01**). This finding aligns with previous research by Ghanavati et al. [39], which also reports that developers heavily rely on manual inspection to identify software defects. Figure 1 illustrates the distribution of the time practitioners usually spend addressing performance anti-patterns (cf. SQ-02). The reported duration ranges from 0 to 16 hours, with an average of approximately 3 hours (**F-02**). This may suggest that, although most practitioners possess a certain level of skill and experience, some complex performance issues still require a considerable amount of time to detect and resolve.

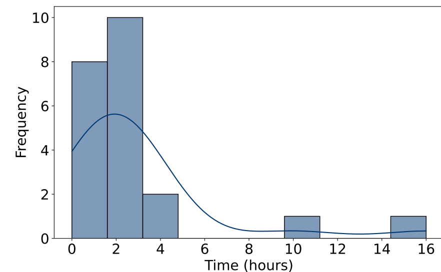

Fig. 1. Time distribution for addressing performance anti-patterns

Regarding the benefits of using automated performance anti-pattern detection tools, 95.5% of participants believe that these tools can improve system efficiency, while 77.3% consider them capable of enhancing developer productivity. Additionally, a relatively small portion of participants (36.4%) mention other benefits, such as aiding in the training of new software engineers (**F-03**). At the same time, participants also highlight potential harms of using these tools. The majority (72.7%) indicate that such tools might create a false sense of performance and overlook issues not detected. Moreover,

[1]“F” hereafter denotes “Finding”.

40.9% of participants feel that over-reliance on such tools could lead to skill degradation among developers, while 22.7% note potential time and resource costs associated with learning new techniques or tools. Smaller portions of participants also raise concerns about deprivation of problem-solving satisfaction (9.1%) and feeling overwhelmed by false positives (4.5%) (**F-04**). In terms of the major concerns of performance anti-pattern detection, respondents mainly focus on detection accuracy (68.2%), integration into existing workflow (45.5%), lack of proper detection tools (36.4%), performance overhead (36.4%), and risk of disclosing proprietary intellectual property (e.g., codebase) to a third-party tool (31.8%). Some participants (18.2%) also mention that experienced engineers may opt to skip using these tools due to their confidence and familiarity with manual methods. Additionally, a small portion (4.5%) note that non-performance-critical parts of the system may require significant time to update according to the detection results (**F-05**).

The survey results also indicate that while a certain proportion of practitioners, accounting for 27.3%, remain cautious, the majority of respondents, accounting for 72.7%, express a willingness to try using performance anti-pattern detection tools (**F-06**). We further investigate practitioners' willingness to adopt such tools and the relationship with their current practices. We find that practitioners who rely on manual and time-consuming processes for detecting performance anti-patterns are more likely to adopt automated tools (**F-07**). This finding underscores the pain points faced by practitioners and highlights the potential of automated tools to save time and effort while enhancing performance.

> Implication 1: Practitioners mainly rely on manual code reviews for performance anti-pattern detection (**F-01**), which is labor-intensive and time-consuming (**F-02**). While attitudes toward automated tools are mixed (**F-03, F-04**), most practitioners are willing to adopt them (**F-06, F-07**) if they are available and capable of addressing key concerns, such as detection accuracy and seamless integration into workflows (**F-05**).

PQ-02: Who are the primary users and focus areas for performance anti-pattern detection? This PQ seeks to explore primary users of automated performance anti-pattern detection tools and types of anti-patterns they should prioritize.

According to the survey results, the majority of respondents (86.4%) believe that both junior and experienced software engineers would benefit from these tools. 9.1% believe junior engineers benefit more from these tools, compared to 4.5% who feel experienced engineers do (**F-08**). This may be because junior engineers tend to rely more on tools to address performance issues, while experienced engineers prefer manual methods, though tools can still improve their efficiency. In terms of roles, respondents perceive that developers benefit the most from automated detection tools (95.5%), while a smaller proportion (54.5%) believe these tools are also useful for testers (**F-09**). These findings highlight previously unexplored aspects of the perceived benefits of automated detection tools across practitioners' different experience levels and roles.

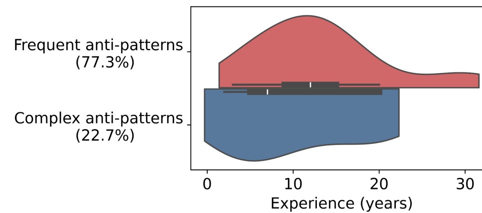

Fig. 2. Relationship between participants' experience and preferred performance anti-pattern types for detection

Regarding the focus areas of performance anti-pattern detection tools, the majority of respondents (95.5%) believe that code-level performance issues are particularly critical. In contrast, system architecture and external data dependencies (e.g., database operations) are the areas of relatively less focus, accounting for 40.9% and 36.4%, respectively (**F-10**). Figure 2 presents the relationship between the types of performance anti-patterns expected to be detected and participants' experience. We notice that practitioners tend to focus more on detecting frequently occurring performance anti-patterns (77.3%), especially those with more experience, while less experienced practitioners are more inclined to address complex performance anti-patterns, accounting for 22.7% (**F-11**). Furthermore, regarding the severity of false positives and false negatives, prior studies in bug detection [35], [40] have reported that practitioners are more concerned with false positives, as they can lead to alert fatigue, wasted effort, and reduced trust in tools. However, we observe that, in the context of performance anti-pattern, participants consider false negatives to be more serious than false positives, with 54.5% versus 45.5% (**F-12**). This is primarily due to false negatives allowing buggy code to go undetected, potentially leading to performance degradation or even system failures, whereas false positives primarily waste investigation effort without directly compromising system performance.

> Implication 2: Performance anti-pattern detection is considered most beneficial for developers (**F-09**), especially junior ones (**F-08**), due to their limited experience. The detection should prioritize identifying code-level (**F-10**) and frequent performance anti-patterns (**F-11**) as comprehensively as possible (**F-12**).

PQ-03: What are practitioners' expectations for performance anti-pattern detection? Our last PQ aims to identify key expectations for anti-pattern detection from practitioners.

As indicated by the survey results, respondents expect the tool to identify potential performance anti-patterns within 1 to 60 minutes, with a median time of 5 minutes (**F-13**). We also examine the relationship between the time practitioners currently spend addressing performance anti-patterns and the time they anticipate automated detection tools to produce results. In particular, we calculate the Pearson correlation between these two times, and our results indicate a correlation coefficient of 0.6 ($p < 0.05$). This strong positive relationship suggests that practitioners who spend more time chasing performance issues

tend to have more lenient expectations of automated detection tools (**F-14**). Regarding workflow integration, all participants (100.0%) prefer integrating detection tools directly into the developers' IDE to detect performance anti-patterns early in development. Additionally, 27.3% of participants also express a desire for integration into the CI/CD pipeline to enable continuous detection (**F-15**). Our finding is consistent with a prior study from Johnson et al. [35], which indicates that developers tend to prefer tools embedded in their primary development environment for real-time feedback and efficiency.

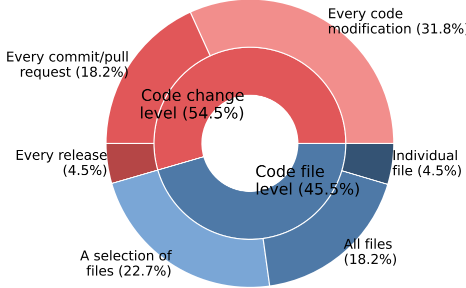

Fig. 3. Distribution of preferred detection granularities

In practice, performance anti-pattern detection can be utilized at various granularities of the code during development, including the code change level, which focuses on the changed code only, and the code level, which focuses on the overall code file. As shown in Figure 3, 54.5% of the respondents believe that, within the actual development process, detecting at the code change level is a more effective approach. Among them, most respondents prefer the tool to run either after every code modification (31.8%) or after every commit/pull request (18.2%) to ensure that issues are detected and fixed promptly. Only a small portion of respondents (4.5%) feel that detection should occur only at every release, which may be late (**F-16**). For those respondents who prefer detecting at the overall code level (45.5%), they present varying detection preferences: 22.7% favor examining a selection of code files, 18.2% prefer analyzing all files together, and the remaining 4.5% opt for reviewing an individual file (**F-17**). This preference may be influenced by the specific project's scale and complexity.

In terms of the expected additional features, we find that providing detailed explanations is considered the most beneficial, with 36.4% of participants ranking it first. Following that, 36.4% and 27.3% of participants suggest that providing actionable fix recommendations would be both the second and third most important feature. Meanwhile, most participants rank integration with developers' IDE (31.8%) and user-friendly interface (36.4%) as the fourth and fifth most beneficial features, respectively (**F-18**).

> Implication 3: Most practitioners expect performance anti-pattern detection to operate at the code change level after each code modification or commit (**F-16, F-17**) and produce results within 5 minutes (**F-13, F-14**). They also prefer detailed explanations, actionable fix recommendations, and seamless IDE integration as additional features (**F-15, F-18**).

III. Industrial Case Study

Our preliminary study has uncovered practitioners' perceptions and expectations regarding performance anti-pattern detection. Building upon these insights, in this section, we present an industrial case study in which we design and develop a performance anti-pattern detector and explore its practical adoption in a real-world software system from our industrial collaborator. We first explain the industrial background, then describe our detector designed to effectively address performance anti-patterns in the target system.

A. Industrial background

Our industrial system under study. The target industrial system, hereafter referred to as *System X*, is a large-scale commercial software system from our industrial collaborator. *System X* provides world-leading software solutions to address challenges in quantifying environmental impact and streamlining regulatory reporting for manufacturing facilities. *System X* is the market leader in its field and has been widely leveraged by several companies globally. The system is developed based on the Microsoft .NET framework and has over two million lines of code. It has over ten years of history and continues to evolve actively with a development and testing team consisting of approximately 30 software engineers. Due to a Non-Disclosure Agreement (NDA), we cannot reveal additional details about the hardware environment and usage scenarios.

The existing performance assurance practice and limitations. Due to the success of *System X*, its performance has become an important aspect in order to provide quality services and ensure future success. To this end, the first author works onsite with the developers of *System X* to support performance assurance on a daily basis. In particular, the system performance is analyzed using a system (A/B) testing approach, where the new version runs alongside the old one, initially serving a small client subset. During execution, the performance metrics of both the old and new versions are collected and compared. If any performance regressions are detected, the new version is rolled back for further investigation; otherwise, it is gradually rolled out to all users.

Despite our prior efforts and advances toward automated and efficient performance assurance activities [2], [8], [14], [16], [53], we still face significant challenges. In particular, our existing practice in ensuring performance remains a reactive task that is conducted after the fact, i.e., after the system is built, integrated, and deployed in the field. Based on our practical experience, large amounts of resources are required to detect, locate, understand, and fix performance issues at such late stages in the development cycle, which often leads to a high operational cost and extended release cycle.

Our proposed solution. To address the limitations, we propose a proactive solution that alerts developers early in development if a code change introduces performance issues. In particular, our solution focuses on automatically identifying performance anti-patterns in the source code. We further incorporate the expected call frequency of the impacted methods

TABLE II
THE SUMMARY OF THE PERFORMANCE ANTI-PATTERNS STUDIED WITHIN OUR INDUSTRIAL CONTEXT (I.E., *System X*)

ID	Anti-pattern	Description	Potential impact	Mitigation suggestion	Origin
AP-01	Inefficient String Concatenation	Concatenating strings with "+" or "+=" operators inside loops, which results in multiple object allocations due to string immutability	Increased memory usage and garbage collection (GC) overhead	Use "StringBuilder" for more efficient string concatenation in loops	[41], [42]
AP-02	Initiating List Without Proper Capacity	Creating a list without specifying an initial capacity, when the expected size is known, causing frequent resizing and copying of elements	Increased memory and CPU usage due to frequent reallocations	Initialize lists with a suitable capacity based on the expected size	[43]
AP-03	Unnecessary Boxing/Unboxing	Implicit conversion between value and reference types in "ArrayList" causes unnecessary overhead	Significant memory and CPU overhead, especially in loops	Use strongly-typed collections such as "List<T>" instead of "ArrayList"	[42], [44]
AP-04	Expensive String Comparison	Comparing strings using "ToLower()" or "ToUpper()" repeatedly, which creates temporary strings	Increased CPU and memory usage, especially in loops	Use "String.Equals()" with "StringComparison.OrdinalIgnoreCase" for efficient and culture-agnostic comparisons	[45]
AP-05	Redundant Execution in LINQ	Using "ToList()" to trigger execution in LINQ queries, which results in unnecessary list creation and iteration	Wasted memory and CPU resources by creating and iterating over a list twice	Avoid calling "ToList()" and directly iterate over the query	[46]
AP-06	Inefficient Emptiness Checking in LINQ	Using "Count()" to check for emptiness in LINQ, which causes full enumeration	CPU overhead due to enumerating the entire sequence to get the count	Use "Any()" to avoid the overhead of full enumeration for emptiness checking in LINQ	[46]
AP-07	Suboptimal Conditional Logic Evaluation	Using "IIf()" will evaluate all arguments regardless of the condition, leading to unnecessary execution	Side effects and unnecessary resource consumption	Use "If" operator instead of "IIf()" to enable short-circuit evaluation	[47]
AP-08	One-by-one Processing	Performing individual database queries in a loop instead of batching operations	Performance overhead from multiple database operations	Use batch processing to combine multiple operations into a single request	[29], [18]
AP-09	Unnecessary Finalizer	Implementing empty finalizers or ones that only call the base class finalizer causes a needless loss of performance	Increased GC pressure and performance degradation	Remove unnecessary finalizers when possible	[48]
AP-10	Redundant Finalization	Failing to call "GC.SuppressFinalize" when implementing "Dispose" leads to redundant finalization with no benefits	Additional GC work and performance overhead	Always call "GC.SuppressFinalize" in "Dispose" to prevent redundant finalization	[49]
AP-11	Inefficient Boolean Evaluation	Using non-short-circuit logical operators, like "Or" and "And", will always evaluate both sides of the operators	Wasted CPU resources, especially in loops	Use the short-circuit equivalents, like "OrElse" and "AndAlso", to avoid unnecessary evaluation	[50]
AP-12	Expensive Assembly Access	Using "Assembly.GetExecutingAssembly()" to get the executing assembly involves walking up the call stack, which is expensive	Performance overhead from call stack walking	Use "Type.Assembly" instead to avoid call stack walking overhead	[51]
AP-13	Suboptimal Searching in Lists	Using "Any()" for searching in Lists is less efficient due to additional enumerator creation and processing, especially for large lists	Slower performance and higher resource usage	Use "List.Exists()" instead of "Any()" for searching in lists with better performance	[52]

Note: In the "ID" column, "AP" stands for "Anti-Pattern".

to assess potential impact on end users, thereby helping developers prioritize anti-patterns that most affect user experience. Our solution aims to reduce the time and effort required for detecting, investigating, and fixing performance issues, and it has been integrated into our collaborator's IDE and internal platform to help improve system performance on a daily basis.

B. Detecting performance anti-patterns

1) Performance anti-patterns studied within our industrial context: In our study, we first analyze performance issues in the development history and issue tracking of *System X*. We then collect relevant issue reports, logs, metrics, and code changes to better understand their manifestation and impact. In the meantime, we conduct a thorough review of literature on software performance anti-patterns, including various sources such as technical blogs [44], [50], [51], [54] and research papers [18], [29], [41], to broaden our understanding of more types of anti-patterns. By integrating insights from the industrial context with academic literature, we identify a collection of performance anti-patterns that are widespread and may cause significant impacts. We have compiled a comprehensive list of studied performance anti-patterns in Table II, which presents each anti-pattern's name, description, potential impact, mitigation suggestion, and origin.

2) Performance anti-patterns detection: To address the aforementioned performance anti-patterns, we design and develop a performance anti-pattern detector based on the findings of our preliminary study (cf. Section II). The overall mapping between our identified PQ findings and corresponding implementations is summarized in Table III. In particular, unlike existing practice that mainly relies on manual effort, our detector aims to automatically identify performance anti-patterns in the source code of the system (cf. **F-01, F-03, F-04, F-06, F-07, F-10**). As illustrated in Figure 4, our detector comprises two main parts: 1) a performance anti-pattern detection frontend, which is developed as a Visual Studio IDE extension; 2) a performance anti-pattern detection backend, which is developed as an ASP.NET web application. The communication between the frontend and the backend is handled by the HTTP protocol. We adopt this architecture to align with practitioners' preference for seamless integration into existing workflows, specifically integrated within developers' IDEs (cf. **F-05, F-**

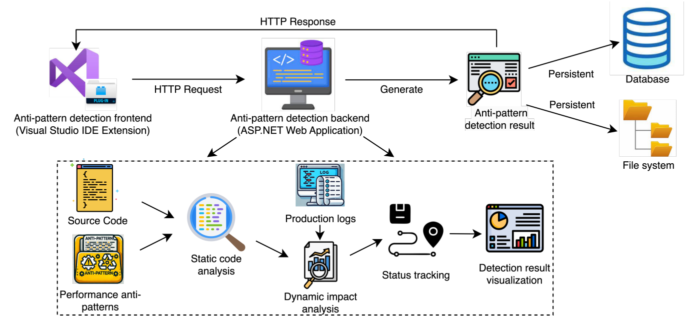

Fig. 4. The overall process of our performance anti-pattern detector

TABLE III
THE MAPPING OF OUR IDENTIFIED PQ FINDINGS, IMPLEMENTATIONS, REMAINING GAPS, EMERGING SUGGESTIONS, AND ACTION PLANS

PQ finding(s)	Implementation	Remaining gap	Suggestion	Action plan
F-01, F-03, F-04, F-06, F-07, F-10	Automated performance anti-pattern detector (overall design)	Insufficient promotion and communication, leading to limited user awareness	ES-04	Follow guidelines to improve user experience; develop comprehensive guides, tutorial videos, and case studies; enhance community outreach
F-05, F-08, F-09, F-15	Visual Studio IDE extension frontend and ASP.NET detection backend	Limited support for multiple programming languages	ES-02	Add JavaScript support; explore SQL and other languages as the longer-term goal
F-02, F-13, F-14	Static code analysis module	Limited customization prevents users from flexibly defining anti-patterns	ES-03	Develop anti-pattern templates and engine with dynamic configuration; advanced capabilities planned later
F-11, F-12	Dynamic impact analysis module	N/A	N/A	N/A
F-16, F-17	Anti-pattern status tracking module	N/A	N/A	N/A
F-18	Detection result visualization module	Lack auto-fix functionality, requiring manual intervention	ES-01	Explore rule-based auto-fix as an initial step; conduct further research on advanced automated fix techniques

Note 1: "F" stands for "Finding", and "ES" stands for "Emerging Suggestion".
Note 2: "N/A" indicates that no specific gaps, suggestions, and action plans are reported for that implementation in the practitioner feedback.

08, F-09, F-15). It also ensures a responsive user interface (frontend) by offloading computationally intensive tasks to the backend, which enhances performance. Additionally, this architecture improves scalability, maintainability, and flexibility by enabling independent updates and efficient communication between the two components. The whole workflow starts with developers' requests to detect performance anti-patterns on one or multiple source code files from the frontend plugin (i.e., Visual Studio IDE extension). Then, the plugin will send these target source code files to the backend service via HTTP request. After receiving the request, the backend service will apply our implemented analysis to identify performance anti-patterns in the target files. When the analysis finishes, the detection results will be sent back to the plugin via HTTP response for visualization to developers. In the meantime, these results will also be persisted in both the SQL database and the file system for further analysis. In the remaining part, we provide a more detailed explanation of each main step involved in the performance anti-pattern detection backend.

Static code analysis. As revealed in our previous study, practitioners prefer to receive detection feedback promptly, i.e., within 5 minutes (cf. **F-02, F-13, F-14**). To achieve it, we adopt static code analysis over dynamic analysis, allowing for effective and efficient scanning of the codebase and detection of anti-patterns without execution. In particular, our detector first leverages the static code analysis framework, i.e., .NET Compiler Platform SDK (Roslyn), to parse source code into the Abstract Syntax Tree (AST). The AST is a tree structure where each node represents a language element, like classes, methods, variables, and operators. We then traverse the extracted AST and apply a set of predefined detection rules, derived from our studied anti-patterns (cf. Table II), to analyze code and identify anti-patterns. Taking AP-08 as an example, our detector aims to identify code where every element in a collection issues an individual database query within a loop, resulting in unnecessary database overhead. During the static analysis process, we collect all code segments matching the anti-patterns and save the results for later analysis.

Dynamic impact analysis. Our preliminary study results indicate that practitioners prioritize detecting performance anti-patterns that occur frequently in practice, which highlights the importance of not only identifying anti-patterns statically in the code but also understanding their runtime impact (cf. **F-11, F12**). To achieve this, we further incorporate dynamic impact analysis using production logs. First, for each detected anti-pattern, we extract the method signature of its corresponding impacted method. We then collect production logs over a specific period (e.g., the past week or month) and count the execution frequency of each operation. As production logs typically record information at a coarse operational level (e.g., web request) due to performance considerations, we use source code analysis to parse the source code and build a call graph for each operation to analyze the execution flow and identify the methods invoked. Next, we match the impacted method signature of each anti-pattern with the call graph of each operation and determine its execution frequency during the specified period, using this as an indicator of potential impact on end users. In the meantime, we also categorize the performance impact into high, medium, and low levels based on the distribution of all methods' execution frequencies during that period to help developers better understand the severity of the anti-pattern. By combining static code analysis with dynamic user data, we can better prioritize performance anti-patterns based on their approximate real-world impact on end users, helping developers focus on the most critical ones.

Anti-pattern status tracking. Practitioners also express interest in detecting performance anti-patterns at the code change level to understand whether they are newly introduced or have existed previously (cf. **F-16, F-17**). To meet this expectation, we implement an analysis to track the status of the detected anti-patterns, including when they were introduced, who was responsible, and how long they have existed in the codebase. In particular, we analyze version history by comparing detected anti-patterns with code changes to determine if they are newly introduced or not. If so, we further identify the developer responsible for its introduction and the commit timestamp. Otherwise, we trace back to the earliest commit with the anti-pattern and calculate its lifespan. By following such practice, we are able to clearly track the status and lifecycle of each anti-pattern, offering valuable insights for further optimization.

Detection result visualization. A detailed presentation and explanation of detection results is among the most beneficial features for practitioners (cf. **F-18**), as it facilitates quick comprehension and resolution of detected anti-patterns. To support practitioners effectively, we present detailed detection results directly within the development environment, specifically in

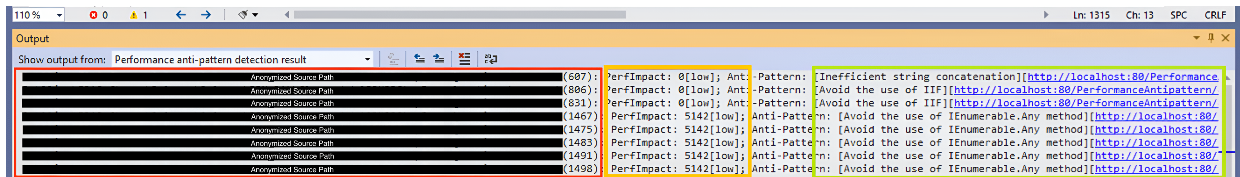

Fig. 5. The visualization of our anti-pattern detector's results (partial screenshot of the Visual Studio output window). The red, orange, and green boxes highlight the location of the performance anti-pattern, its performance impact, and the corresponding detailed information, respectively.

Visual Studio's output window. As an illustration, Figure 5 shows a screenshot of our detection results visualization. Each line in the output corresponds to a detected performance anti-pattern in the source code. The result consists of three main parts: 1) Code location (in the red box) shows where the anti-pattern occurs, including file path and line number. Developers can click to navigate directly to the line of code, allowing them to quickly locate and review the issue. 2) Performance impact (in the orange box) shows how often the affected code is called and its severity. This can help developers assess the impact and priority of the issue. 3) Anti-pattern information (in the green box) includes the name of the anti-pattern, along with a link to the detailed documentation and suggested remediation. It aims to help developers gain a deeper understanding of the detected anti-pattern and provide recommended improvements. Our design goes beyond simply displaying detected anti-patterns by providing additional insights that facilitate understanding, support decision-making, and streamline workflow.

IV. Industrial Adoption Feedback

In this section, we first present adoption results, highlighting how our detector is utilized and performed in the target industrial setting. We then discuss feedback from practitioners who have used our detector, focusing on their experiences and insights for further improvements to facilitate adoption.

A. *Adoption results*

Our performance anti-pattern detector has been adopted and integrated into our industrial collaborator's internal development platform to assist developers in ensuring and optimizing system performance. Based on the adoption results, we observe that after the first official release of our detector on the internal platform, all developers at our industrial collaborator have adopted it, with an average usage of >40 times in total per week. During its use, our detector has successfully detected >70 occurrences of performance anti-patterns. Among these detected occurrences, the most commonly occurring anti-pattern in *System X* is AP-01, with >20 occurrences. We further investigate the potential causes for this phenomenon and observe that, due to the database-centric nature of *System X*, the source code involves extensive string manipulations to generate SQL queries. As a result, developers may unintentionally introduce suboptimal practices when concatenating these SQL query strings. This is followed by AP-05 and AP-08, with >10 occurrences for each. In contrast, less frequent anti-patterns, such as AP-07, AP-10, and AP-12, appear only occasionally (<3 occurrences for each). For confidentiality concerns, we cannot disclose the exact number of anti-pattern occurrences. Compared to the traditional manual inspection process, where developers usually spend several hours or even days identifying performance issues, our detector demonstrates remarkable efficiency. The detection process is highly responsive, typically completing within one minute, with an average response time of 20 seconds.

As of the time of writing this paper, we observe that >80% of detected anti-patterns have been acknowledged, with <5% flagged as false positives, mainly due to context-dependent code behavior (e.g., small loops or startup-only invocations), while false negatives remain difficult to estimate without complete ground truth. Moreover, >60% have been fixed by developers, enabling them to address many common performance issues early in development. For instance, one notable improvement comes from AP-01, which reduces execution time in some cases from 10-15 seconds to merely 1 second, yielding a performance gain of >90%. The adoption results demonstrate that our detector has been highly effective in the target industrial setting, helping developers uncover and address many previously unnoticed performance anti-patterns.

B. *Feedback from practitioners*

To further assess the adoption of our detector, we have gathered feedback from practitioners through biweekly meetings and discussions, focusing on its usability, effectiveness, and areas for further improvement. According to the feedback, practitioners have highly praised our effort, with around 80% of them having already integrated it into their daily development workflows, using it, for instance, every time writing code or before submitting code changes, to detect performance anti-patterns (addressing **F-03, F-04, F-06, F-07, F-10, F-16, F-17**). The remaining 20% of practitioners also report using our detector 2 to 3 times a week. In particular, they highlight that our detector's responsive detection capability has significantly reduced the time and simplified the process required to identify performance anti-patterns, thus enhancing work efficiency (addressing **F-01, F-02, F-13, F-14**). In addition, the integration with their IDE has also been well-recognized, as developers can use it directly within their familiar workspace, further streamlining their workflow (addressing **F-05, F-15**). Developers also highly value our detector's accuracy and relevance, as it efficiently identifies performance-affecting anti-patterns and provides prioritized recommendations based on dynamic production information, helping them focus resources on the most urgent performance issues (addressing **F-11, F-12**). For instance, developers have pointed out that some occurrences of AP-05 are frequently invoked during certain periods, which could lead to significant performance impacts and thus require special attention from developers. Meanwhile, some detected issues, such as AP-11, are identified in legacy

code and have not been executed in recent times, posing no immediate impact on system performance. Furthermore, some developers, particularly junior ones, mention that the clear and understandable explanations in the result visualization have helped them quickly grasp the issues and take appropriate corrective actions (addressing **F-08, F-09, F-18**).

Despite the successful adoption of our performance anti-pattern detector in improving developers' workflows and ensuring system performance, practitioners, after using our detector, have also raised some emerging suggestions (**ES**) where the performance anti-pattern detection can be further enhanced for practical adoption. Table III summarizes the mapping of our identified PQ findings, implementations, remaining gaps, emerging suggestions, and action plans.

ES-01: Auto-fix performance anti-patterns with one click. One key aspect raised by practitioners in their feedback is the expectation that the detection tool should support automatic fixing of detected performance anti-patterns. While our current visualization module effectively highlights performance issues and helps developers understand them (aligning **F-18**), many participants express a desire for the tool to not only identify anti-patterns but also offer direct optimization suggestions and automated fixes. For instance, they wish the detector could help recognize inefficient algorithms or improper method calls, such as those described in AP-04, AP-11, and AP-13, and then automatically suggest or apply corrections. As one developer commented, *"Having the tool fix these problems automatically for me would save so much time and help avoid mistakes, especially for junior engineers."* This feedback underscores a remaining gap in our current implementation.

Action plan: Developing a reliable auto-fix capability requires advanced code analysis and contextual understanding, which goes beyond the current scope of our implementation. Therefore, we consider auto-fix a longer-term research goal. In the near term, we plan to explore simpler, rule-based automatic fixes to partially address this gap while continuing research on more sophisticated automated fix techniques.

ES-02: Support more programming languages. Another important direction for improving the adoption in practice is expanding the support to a broader range of programming languages. Although the current detection frontend and backend provide effective integration for C# projects and help developers seamlessly identify anti-patterns within their workflows (aligning **F-05, F-08, F-09, F-15**), practitioners point out that its language coverage remains somewhat limited. Specifically, several participants emphasize that performance issues can manifest differently across various languages. As one practitioner noted, *"Our main codebase is in JavaScript, so having support beyond C# would make this tool much more valuable to us."* Another developer highlights that, *"In SQL development, poorly written queries can be a huge performance bottleneck, but the detector does not handle that."* This feedback indicates that the lack of multi-language support is a remaining gap that limits broader adoption.

Action plan: We consider incremental expansion of language support to be feasible in the short term. Our initial focus will be on adding JavaScript coverage based on existing detection rules. Support for more specialized languages, such as SQL, will be treated as a longer-term goal requiring additional research and domain-specific rule development.

ES-03: Extend to more customizable anti-patterns. Extending the detector to support more customizable anti-patterns has also been identified by practitioners as an important area for improvement. The current static code analysis module effectively and efficiently detects a predefined set of common performance anti-patterns (aligning **F-02, F-13, F-14**). However, practitioners note that it could be more flexible if it supported the definition of complex or domain-specific anti-patterns. For example, some participants working on microservices projects report that performance issues such as inefficient inter-service communication, excessive synchronous calls, or suboptimal caching strategies are often critical but cannot be fully captured by the current detector. As one practitioner commented, *"Every project has its own performance pitfalls, and being able to define our own anti-patterns would significantly increase the tool's usefulness."* Practitioners believe that offering flexibility in customizing the detection of anti-patterns would improve the detector's utility across diverse development scenarios.

Action plan: We plan to provide a library of anti-pattern templates and a customizable engine that allows users to define anti-patterns, dynamically configure thresholds, and enable or disable specific rules. In the longer term, we will explore advanced capabilities, including detecting architectural- or system-level anti-patterns and leveraging large language models (LLMs) to mine bug reports for learning project-specific anti-patterns and generating static analysis templates.

ES-04: Better promotion and communication. One consensus among practitioners is that the promotion and communication of the performance anti-pattern detector needs to be further improved to support broader adoption. While the automated detector's overall design demonstrates strong technical capabilities (aligning **F-01, F-03, F-04, F-06, F-07, F-10**), many practitioners feel that its benefits and value are not sufficiently communicated or well understood. Practitioners emphasize that better documentation, tutorial videos, and real-world case studies are essential for helping users understand the detector's functionalities and practical applications. As one participant noted, *"Without clear guides and examples, it is difficult to see how this tool fits into our existing workflows."* Others also point out the importance of active outreach and training to bridge the knowledge gap between technical capability and practical use. This feedback highlights a remaining gap in user awareness and understanding, which, to some extent, limits the detector's industrial adoption.

Action plan: In the near term, we plan to follow Weninger et al.'s guidelines [55] to better guide and support novice users while improving the overall user experience. We will also focus on developing comprehensive documentation, creating tutorial videos, and producing real-world case studies to facilitate learning and adoption. Furthermore, we will increase community engagement and outreach efforts to raise awareness and demonstrate the detector's practical benefits.

V. Related work

Software performance anti-patterns. Software performance anti-patterns [28], [56]–[58] can manifest at various levels when designing and implementing a software system. At the highest level of software design, architectural-level performance anti-patterns often arise from inefficient component layouts or poorly designed dependencies [19]–[21], [31], [32]. At the class level, performance anti-patterns often manifest as suboptimally designed classes that introduce unnecessary complexity and inefficient resource usage [7], [28], [30], [41], [42], [59], [60]. In addition, since many systems rely heavily on databases to accomplish users' tasks, several prior studies have focused on object-relational mapping (ORM)-level anti-patterns, which pertain to how applications interact with databases through query execution [18], [29], [61]–[63].
Detecting software performance anti-patterns. Depending on the type of analysis used, performance anti-pattern detection can be broadly classified into two categories. The first category employs static code analysis techniques and involves examining source code [59] or compiled binaries [64] without actually executing the software. It identifies structural flaws, inefficient algorithms, or resource-heavy operations by applying predefined rules or patterns [25], [29]. In addition, the second category relies on dynamic analysis techniques, where software is executed and its performance is monitored during runtime [30]. This type allows for detecting anti-patterns that only manifest during runtime and are difficult to uncover through static analysis alone [18]. For instance, dynamic analysis can highlight inefficiencies in resource utilization [7], [18], unexpected latency under varying workloads [65], and the impact of complex interactions between system components [21], [66], [67]. On one hand, static analysis is less resource-intensive since it does not require software to be executed, making it a fast and efficient way to catch potential problems early in the development cycle. On the other hand, dynamic analysis typically provides greater accuracy in identifying runtime-specific performance issues and offers detailed insights into their impact under actual usage conditions.

Unlike existing studies, our detector combines static code analysis with dynamic impact analysis, enabling both efficient detection and reliable evaluation of the impact. Furthermore, we have applied our detector to a real-world industrial system and gathered valuable feedback from practitioners to evaluate its usability and effectiveness, as well as to identify areas for further improving performance anti-pattern detection.

VI. Threats to validity

This section discusses the threats to the validity of our study.
External validity. Our preliminary study on practitioner perspectives is based on responses from 22 participants. Although the sample may not be broad enough, we made a deliberate effort to include participants from diverse backgrounds to capture a wide spectrum of perspectives. In addition, we investigated the practical adoption of our detector in our industrial collaborator's software system. The system has many years of history and is actively evolving, which offers a certain degree of representativeness. However, the generalizability of our findings to other systems may be limited. Future research could expand to include a broader range of software systems.
Construct validity. We designed a questionnaire to capture practitioner perspectives on performance anti-pattern detection; however, the way questions were framed and the specific aspects we focused on may have influenced the responses. To mitigate this, we carefully designed and rigorously validated the questionnaire, further refining it through a pilot study to ensure clarity and effectiveness. Another potential threat is that, due to operational restrictions in our target industrial system, the effectiveness of our detector was assessed through adoption metrics and qualitative feedback rather than extensive quantitative analysis or benchmarking, which may not fully capture its impact on system performance. While developers report high satisfaction with our detector, future work could complement our findings with broader controlled experiments.
Internal validity. One potential threat to internal validity is self-report bias, where participants may provide inaccurate information due to social desirability or memory biases. To mitigate this, we made survey responses anonymous and used precise, neutral, and non-leading question wording to encourage participants to provide more accurate answers, thereby reducing such bias. Furthermore, we incorporated the expected call frequencies of impacted methods derived from dynamic production logs as a lightweight yet reasonably reliable approach to assess the impact of each detected performance anti-pattern, helping developers prioritize investigations and fixes. Nevertheless, this approach provides only an approximate estimation rather than the actual runtime performance impact. A more advanced dynamic impact analysis, while remaining low-overhead, will be pursued in our future research.

VII. Conclusion

In this paper, we conduct an industrial case study aiming to address the gap in the practical adoption of performance anti-pattern detection in real-world contexts. In particular, we first explore practitioners' perceptions and expectations of performance anti-pattern detection. Based on these valuable insights, we then design and develop a static performance anti-pattern detector and adopt it in a large-scale software system from our industrial collaborator, helping to proactively optimize its performance. Furthermore, we collect and analyze the feedback from developers who have used our detector to gain insights for further improvements to better meet practitioners' demands in performance anti-pattern detection.

Acknowledgment

We are grateful to ERA Environmental Management Solutions for providing access to the industrial system used in our study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of ERA Environmental Management Solutions and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of ERA Environmental Management Solutions' products.

References

[1] D. A. Al Shoaibi and M. W. Mkaouer, "Understanding software performance challenges an empirical study on stack overflow," in *2023 International Conference on Code Quality (ICCQ)*, 2023, pp. 1–15.

[2] L. Liao, H. Li, W. Shang, C. Sporea, A. Toma, and S. Sajedi, "Adapting performance analytic techniques in a real-world database-centric system: An industrial experience report," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. ACM, 2023, pp. 1855–1866.

[3] J. Chen, Z. Ding, Y. Tang, M. Sayagh, H. Li, B. Adams, and W. Shang, "Iopv: On inconsistent option performance variations," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. ACM, 2023, pp. 845–857.

[4] T. Chen, W. Shang, A. E. Hassan, M. N. Nasser, and P. Flora, "Detecting problems in the database access code of large scale systems: an industrial experience report," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. ACM, 2016, pp. 71–80.

[5] W. Shang, A. E. Hassan, M. N. Nasser, and P. Flora, "Automated detection of performance regressions using regression models on clustered performance counters," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, January 31 - February 4, 2015*. ACM, 2015, pp. 15–26.

[6] C. M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*. IEEE Computer Society, 2007, pp. 171–187.

[7] C. Trubiani, R. Pinciroli, A. Biaggi, and F. A. Fontana, "Automated detection of software performance antipatterns in java-based applications," *IEEE Trans. Software Eng.*, vol. 49, no. 4, pp. 2873–2891, 2023.

[8] L. Liao, "Addressing performance regressions in devops: Can we escape from system performance testing?" in *45th IEEE/ACM International Conference on Software Engineering: ICSE 2023 Companion Proceedings, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 203–207.

[9] C. Bezemer, S. Eismann, V. Ferme, J. Grohmann, R. Heinrich, P. Jamshidi, W. Shang, A. van Hoorn, M. Villavicencio, J. Walter, and F. Willnecker, "How is performance addressed in devops? A survey on industrial practices," *CoRR*, vol. abs/1808.06915, 2018.

[10] L. Traini, "Exploring performance assurance practices and challenges in agile software development: An ethnographic study," *Empir. Softw. Eng.*, vol. 27, no. 3, p. 74, 2022.

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

[12] T. Chen, M. D. Syer, W. Shang, Z. M. Jiang, A. E. Hassan, M. N. Nasser, and P. Flora, "Analytics-driven load testing: An industrial experience report on load testing of large-scale systems," in *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE Computer Society, 2017, pp. 243–252.

[13] Y. Xia, L. Liao, J. Chen, H. Li, and W. Shang, "Reducing the length of field-replay based load testing," *IEEE Trans. Software Eng.*, vol. 50, no. 8, pp. 1967–1983, 2024.

[14] L. Liao, J. Chen, H. Li, Y. Zeng, W. Shang, J. Guo, C. Sporea, A. Toma, and S. Sajedi, "Using black-box performance models to detect performance regressions under varying workloads: an empirical study," *Empir. Softw. Eng.*, vol. 25, no. 5, pp. 4130–4160, 2020.

[15] J. Chen, W. Shang, and E. Shihab, "Perfjit: Test-level just-in-time prediction for performance regression introducing commits," *IEEE Trans. Software Eng.*, vol. 48, no. 5, pp. 1529–1544, 2022.

[16] L. Liao, J. Chen, H. Li, Y. Zeng, W. Shang, C. Sporea, A. Toma, and S. Sajedi, "Locating performance regression root causes in the field operations of web-based systems: An experience report," *IEEE Trans. Software Eng.*, vol. 48, no. 12, pp. 4986–5006, 2022.

[17] E. J. Weyuker and F. I. Vokolos, "Experience with performance testing of software systems: Issues, an approach, and case study," *IEEE Trans. Software Eng.*, vol. 26, no. 12, pp. 1147–1156, 2000.

[18] B. Chen, Z. M. Jiang, P. Matos, and M. Lacaria, "An industrial experience report on performance-aware refactoring on a database-centric web application," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 653–664.

[19] R. Calinescu, V. Cortellessa, I. Stefanakos, and C. Trubiani, "Analysis and refactoring of software systems using performance antipattern profiles," in *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, ser. Lecture Notes in Computer Science, vol. 12076. Springer, 2020, pp. 357–377.

[20] R. Pinciroli, C. U. Smith, and C. Trubiani, "Qn-based modeling and analysis of software performance antipatterns for cyber-physical systems," in *ICPE '21: ACM/SPEC International Conference on Performance Engineering, Virtual Event, France, April 19-21, 2021*. ACM, 2021, pp. 93–104.

[21] V. Cortellessa, D. D. Pompeo, R. Eramo, and M. Tucci, "A model-driven approach for continuous performance engineering in microservice-based systems," *J. Syst. Softw.*, vol. 183, p. 111084, 2022.

[22] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: detecting performance problems via similar memory-access patterns," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, 2013, pp. 562–571.

[23] B. Chen and Z. M. J. Jiang, "Characterizing and detecting anti-patterns in the logging code," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE / ACM, 2017, pp. 71–81.

[24] C. Vassallo, S. Proksch, H. C. Gall, and M. D. Penta, "Automated reporting of anti-patterns and decay in continuous integration," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 2019, pp. 105–115.

[25] S. S. Afjehei, T. P. Chen, and N. Tsantalis, "iperfdetector: Characterizing and detecting performance anti-patterns in ios applications," *Empir. Softw. Eng.*, vol. 24, no. 6, pp. 3484–3513, 2019.

[26] J. Pantiuchina, F. Zampetti, S. Scalabrino, V. Piantadosi, R. Oliveto, G. Bavota, and M. D. Penta, "Why developers refactor source code: A mining-based study," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, pp. 29:1–29:30, 2020.

[27] I. van Dinten, P. Derakhshanfar, A. Panichella, and A. Zaidman, "The slow and the furious? performance antipattern detection in cyber-physical systems," *J. Syst. Softw.*, vol. 210, p. 111904, 2024.

[28] C. U. Smith and L. G. Williams, "Software performance antipatterns for identifying and correcting performance problems," in *38. International Computer Measurement Group Conference, Las Vegas, NV, USA, December 3-7, 2012*. Computer Measurement Group, 2012.

[29] T. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. N. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. ACM, 2014, pp. 1001–1012.

[30] A. E. Chis, "Automatic detection of memory anti-patterns," in *Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-13, 2007, Nashville, TN, USA*, G. E. Harris, Ed. ACM, 2008, pp. 925–926.

[31] V. Cortellessa, A. D. Marco, and C. Trubiani, "Software performance antipatterns: Modeling and analysis," in *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*, ser. Lecture Notes in Computer Science, vol. 7320. Springer, 2012, pp. 290–335.

[32] C. Trubiani, A. Koziolek, V. Cortellessa, and R. H. Reussner, "Guilt-based handling of software performance antipatterns in palladio architectural models," *J. Syst. Softw.*, vol. 95, pp. 141–165, 2014.

[33] B. A. Kitchenham and S. L. Pfleeger, "Personal opinion surveys," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. Springer, 2008, pp. 63–92.

[34] "Microsoft forms," 2025. [Online]. Available: https://forms.microsoft.com

[35] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *35th International Conference on Software Engineering, ICSE '13, San*

Francisco, CA, USA, May 18-26, 2013. IEEE Computer Society, 2013, pp. 672–681.

[36] M. Weninger, P. Grünbacher, E. Gander, and A. Schörgenhumer, "Evaluating an interactive memory analysis tool: Findings from a cognitive walkthrough and a user study," *Proc. ACM Hum. Comput. Interact.*, vol. 4, no. EICS, pp. 75:1–75:37, 2020.

[37] "Replication package," 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17582209

[38] "Github copilot · your ai pair programmer," 2025. [Online]. Available: https://github.com/features/copilot

[39] M. Ghanavati, D. Costa, J. Seboek, D. Lo, and A. Andrzejak, "Memory and resource leak defects and their repairs in java projects," *Empir. Softw. Eng.*, vol. 25, no. 1, pp. 678–718.

[40] H. Shen, J. Fang, and J. Zhao, "Efindbugs: Effective error ranking for findbugs," in *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*. IEEE Computer Society, 2011, pp. 299–308.

[41] Y. H. Tian, "String concatenation optimization on java bytecode," in *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 2*. CSREA Press, 2006, pp. 945–951.

[42] N. Tileubergenov, S. N. Romanchikova, and A. K. Zhumadillayeva, "Optimization of processing power by improving the performance of application service," in *2022 IEEE 23rd International Conference of Young Professionals in Electron Devices and Materials (EDM)*, 2022, pp. 718–721.

[43] "Increasing performance via low memory allocation in c," 2019. [Online]. Available: https://endjin.com/blog/2019/09/increasing-performance-via-low-memory-allocation

[44] "Boxing performance in c – analysis and benchmark," 2019. [Online]. Available: https://mihai-albert.com/2019/12/15/boxing-performance-in-c-analysis-and-benchmark/

[45] "String.equals with ordinalignorecase compared to tolower / toupper for string insensitive comparisons in c," 2023. [Online]. Available: https://davecallan.com/string-equals-ordinalignorecase-compared-tolower-toupper-string-insensitive-comparisons-csharp/

[46] "C linq performance optimization: Tips and tricks," 2023. [Online]. Available: https://www.bytehide.com/blog/linq-performance-optimization-csharp

[47] "If operator (visual basic)," 2021. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/visual-basic/language-reference/operators/if-operator

[48] "Finalizers (c programming guide)," 2023. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/finalizers

[49] "Ca1816: Call gc.suppressfinalize correctly," 2023. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca1816

[50] "Logical and bitwise operators in visual basic," 2021. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/operators-and-expressions/logical-and-bitwise-operatorsshort-circuiting-logical-operations

[51] ""assembly.getexecutingassembly" should not be called," 2023. [Online]. Available: https://rules.sonarsource.com/csharp/tag/performance/RSPEC-3902/

[52] "Understanding exist and any method in c and their differences," 2023. [Online]. Available: https://www.codepedia.info/csharp-exists-vs-any-difference-between

[53] H. Zhang, L. Liao, Z. Ding, W. Shang, N. Narula, C. Sporea, A. Toma, and S. Sajedi, "Towards a robust waiting strategy for web GUI testing for an industrial software system," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*. ACM, 2024, pp. 2065–2076.

[54] "Memory anti-patterns in c," 2021. [Online]. Available: https://techblog.criteo.com/memory-anti-patterns-in-c-7bb613d55cf0

[55] M. Weninger, E. Gander, and H. Mössenböck, "Guided exploration: A method for guiding novice users in interactive memory monitoring tools," *Proc. ACM Hum. Comput. Interact.*, vol. 5, no. EICS, pp. 209:1–209:34, 2021.

[56] C. U. Smith and L. G. Williams, "Software performance antipatterns," in *Second International Workshop on Software and Performance, WOSP 2000, Ottawa, Canada, September 17-20, 2000*. ACM, 2000, pp. 127–136.

[57] ——, "New software performance antipatterns: More ways to shoot yourself in the foot," in *28th International Computer Measurement Group Conference, December 8-13, 2002, Reno, Nevada, USA, Proceedings*. Computer Measurement Group, 2002, pp. 667–674.

[58] ——, "More new software performance antipatterns: Even more ways to shoot yourself in the foot," in *Computer Measurement Group Conference*, 2003, pp. 717–725.

[59] A. Nistor, P. Chang, C. Radoi, and S. Lu, "CARAMEL: detecting and fixing performance problems that have non-intrusive fixes," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society, 2015, pp. 902–912.

[60] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empir. Softw. Eng.*, vol. 21, no. 3, pp. 1143–1191, 2016.

[61] J. Yang, P. Subramaniam, S. Lu, C. Yan, and A. Cheung, "How *not* to structure your database-backed web applications: a study of performance bugs in the wild," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 2018, pp. 800–810.

[62] C. Yan, A. Cheung, J. Yang, and S. Lu, "Understanding database performance inefficiencies in real-world web applications," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*. ACM, 2017, pp. 1299–1308.

[63] J. Yang, C. Yan, P. Subramaniam, S. Lu, and A. Cheung, "Powerstation: automatically detecting and fixing inefficiencies of database-backed web applications in IDE," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 2018, pp. 884–887.

[64] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association, 2003.

[65] C. Trubiani, A. Bran, A. van Hoorn, A. Avritzer, and H. Knoche, "Exploiting load testing and profiling for performance antipattern detection," *Inf. Softw. Technol.*, vol. 95, pp. 329–345, 2018.

[66] L. Liao, S. Eismann, H. Li, C. Bezemer, D. E. Costa, A. van Hoorn, and W. Shang, "Early detection of performance regressions by bridging local performance data and architectural models," in *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2025, pp. 2841–2853.

[67] T. Parsons and J. Murphy, "Detecting performance antipatterns in component based enterprise systems," *J. Object Technol.*, vol. 7, no. 3, pp. 55–91, 2008.