Automated Detection of Software Performance Antipatterns in Java-based Applications

Catia Trubiani, Riccardo Pinciroli, Andrea Biaggi, and Francesca Arcelli Fontana

Abstract—The detection of performance issues in Java-based applications is not trivial since many factors concur to poor performance, and software engineers are not sufficiently supported for this task. The goal of this manuscript is the automated detection of performance problems in running systems to guarantee that no quality-based hinders prevent their successful usage. Starting from software performance antipatterns, i.e., bad practices (e.g., extensive interaction between software methods) expressing both the problem and the solution with the purpose of identifying shortcomings and promptly fixing them, we develop a framework that automatically detects seven software antipatterns capturing a variety of performance issues in Java-based applications. Our approach is applied to real-world case studies from different domains, and it captures four real-life performance issues of Hadoop and Cassandra that were not predicted by state-of-the-art approaches. As empirical evidence, we calculate the accuracy of the proposed detection rules, we show that code commits inducing and fixing real-life performance issues present interesting variations in the number of detected antipattern instances, and solving one of the detected antipatterns improves the system performance up to 50%.

Index Terms—Software performance antipatterns, Java-based applications, Dynamic analysis.

1 Introduction

THE performance evaluation of Java-based applications is challenging due to many variabilities, such as software failures and workload fluctuation in requests [1], [2], [3], that may occur when the system is running and inevitably contribute to affect the overall service quality [4], [5], [6]. Understanding if an application can always meet the desired performance (e.g., the system response time must be shorter than 5 seconds or resource utilization must be lower than 80%) is of key relevance since it impacts the perception of end-users and their satisfaction while interacting with the system [7], [8].

In the literature, several approaches have been proposed for modeling, analyzing, and optimizing the performance of software applications [9], [10]. Two main directions have been pursued: (i) model-based performance analysis, i.e., performance models are built out of Java applications [11], [12] and used for predictions; (ii) application performance monitoring, i.e., tools that collect trace data for inspection [13], [14]. Motivated by the recent trend of integrating development (Dev) and operations (Ops) teams, processes, and tools [15], [16], [17], it is necessary that software engineers are aware of the performance evolution of their applications. If performance issues are detected, then engineers must also be able to promptly fix such issues. To this end, several approaches emerged, e.g., automated performance tests [18] to guarantee the prompt identification and fixing of performance degradation, or performance load testing [19] to evaluate software refactorings that most likely lead to performance improvement. However, most of the approaches in the literature, e.g., [2], [20], [21], [22], act statically on the implementation code. A recent study [23] pointed out that static code analysis may fail in capturing complex root causes of real-life performance issues, e.g., the interactions between procedures that occur when executing the source code only. Hence, in this paper we aim to exploit dynamic information, which is fundamental for the detection of some performance issues, at the cost of deploying a testing environment for profiling applications and introducing runtime efforts.

We focus our attention on identifying performance issues in Java-based applications, i.e., the target systems under analysis are already in production and subject to multiple variations, such as changes in the execution environment. As a motivating example, let us consider a real-life case study [24] where a performance overhead of 17% is experienced for the continuous integration of a software release. The diagnosis of performance problems is indeed non-trivial, a study in [25] indicates the Apache project, i.e., relevant to our research since Java is used, with the longest average diagnosis time (194 days). Our goal is to improve the system performance by identifying the bad practices of software components and fixing them before the system becomes unusable. Let us consider as an example of bad practice a software component that monopolizes the processing (namely the *Blob* [26]). This implies a single and complex *controller* component that orchestrates the computation by extensively interacting with other components. As a consequence, the system response time can suffer by such a behaviour. To fix this problem, it would be beneficial to involve other software components and delegate them part of the overall computation. This way, the system response time can improve since it benefits from some processing running in parallel. To achieve this objective of identifying bad practices, we use software performance antipatterns [26], [27] since they include the description of both (i) the problems leading to performance flaws, and (ii) the best practices aimed to get performance improvements.

- *C. Trubiani and R. Pinciroli are with the Gran Sasso Science Institute, Italy. E-mail: {catia.trubiani, riccardo.pinciroli}@gssi.it*
- *A. Biaggi and F. Arcelli Fontana are with University of Milano-Bicocca, Italy. E-mail: francesca.arcelli@unimib.it*

In the context of Java-based applications, we focus on the following seven software performance antipatterns: 1) Circuitous Treasure Hunt (CTH), 2) Extensive Processing (EP), 3) Wrong Cache Strategy (WCS), 4) Blob, 5) Tower of Babel (ToB), 6) Empty Semi Trucks (EST), and 7) Excessive Dynamic Allocation (EDA). Further details on these antipatterns and the motivation on the selection of these antipatterns are provided in Section 3.3. We develop a framework to automatically detect these seven performance antipatterns and we evaluate it on a variegate set of real-world applications. The conducted experimentation advocates the following main findings: (i) our framework is efficient, the detection of antipatterns is performed, on average, in less than a minute; (ii) our framework is accurate, the F1 score is larger than 85% in the considered cases; (iii) our framework advances state-of-the-art methodologies, it detects complex performance problems not recognized by other tools; (iv) antipattern-based refactoring can lead to system performance improvement up to 50%. The main contributions of our work are as follows:

- the specification of seven software performance antipatterns that are customized to verify a set of properties for Java-based applications;
- the development of JPAD, *Java Performance Antipattern Detector*, a framework that automatically detects the seven software performance antipatterns;
- the evaluation of JPAD efficiency and accuracy on five real-world applications with different complexity and representative of multiple domains;
- the comparison of JPAD with state-of-the-art approaches on the detection of real-life performance issues in nine code commits of two further systems;
- empirical evidence on the benefit of solving performance antipatterns.

In summary, our approach advocates the usage of software performance antipatterns as valuable support to automatically detect performance issues of Java-based applications. The benefit is that software engineers are promptly informed of software components showing specific bad practices and candidate of being refactored.

The rest of the manuscript is organized as follows. Section 2 reviews the related work. Section 3 describes our approach, and we discuss the key properties of software performance antipatterns, thus to motivate the choice of implementing some of them. Details on detection algorithms for the seven implemented software performance antipatterns are provided in Appendix A, the rationale for not implementing some of them is explained in Appendix B. Research questions, analyzed software systems, and the experimental evaluation are presented in Section 4. Threats to validity are argued in Section 5. A discussion on limitations of the approach is reported in Section 6. Concluding remarks and possible directions for future research are outlined in Section 7. Replication data are publicly available [28].

2 Related Work

Our work mainly relates to three streams of research, i.e., architectural antipatterns, code smells, and Java-specific performance issues that are briefly reviewed in the following. This manuscript moves a step forward in the attempt of establishing synergies between architectural antipatterns and code smells for the performance evaluation of Java applications.

Architectural antipatterns. In the broader context of (anti)-patterns and quality attributes (e.g., reliability, security), there are several works that aim to match their connections, e.g., [29], [30], [31], [32]. When focusing on performance-related concerns there is much less work. Software performance antipatterns are studied first by Smith [26], [27] who provides the preliminary definitions based on her experience. The specification is expressed in natural language and is technology-independent, meaning that antipatterns can be customized in many different contexts. Other researchers redefine these natural lagnuage definitions using first-order logical predicates later applied to architectural design models [33] and recently adapted to further architectural formalism such as probabilistic model checking [34]. A first attempt of adopting architectural antipatterns in running systems is provided in [35], where problem root causes are isolated and a graph of dependencies is built to match problems with the specification of antipatterns.

Static and dynamic approaches for code smells. In the literature, extensive work is devoted to investigate code smells [36], [37], [38], and several investigations are performed, e.g., the analysis of (i) inter-smell interactions to understand their effects [39] and (ii) sequences of different kinds of bad smells to improve detection and solution [40]. Static analysis techniques are adopted to locate bugs in software, e.g., performance bugs that waste processing time due to superfluous loop iterations are detected in [21], and tools like FindBugs [41] can find potential root causes for performance antipatterns. Object-Relational Mapping (i.e., non-trivial database access) is exploited in [20] where static analysis can detect a huge number of performance antipattern instances. Li et al. [22] keep using static analysis, but focus on problematic duplicate logging code smells to emphasize that logging code is highly associated with both the structure and the functionality of the surrounding code. An approach to identify code changes that may potentially be responsible for performance regressions is proposed in [42], but it does not analyze root causes behind such regressions. An exploratory study on performance regressions is presented in [43] where six code level root-causes are identified, but they mainly refer to inner changes, e.g., function calls or parameter values. An attempt of using static and dynamic metrics is proposed in [44] where evolutionary algorithms are adopted to detect performance regressions, however causes are not treated. Static and dynamic data is exploited also in [45] where a selection of benchmarks limits runtime overhead at the cost of missing the prediction of some performance issues. Preventing performance issues before the commit of code changes is pursued in [23] where random forest classifiers are trained on large datasets of performance regressions. More recently, an experience report on locating the root causes of performance regressions is presented in [46] where web-access logs are exploited to tune the expected workload. However, this report targets only web-based systems, and machine learning techniques may lack to capture the relationship between problematic runtime activ-

ities and their impact on system performance. Summarizing, the main difference between static and dynamic approaches is that the former can detect some performance issues during the development process and cost fewer efforts, whereas the latter can benefit from runtime information probably capturing a larger set of issues (as confirmed in [23]), with the drawback of introducing monitoring efforts.

Java-specific performance issues. Our work mainly relates to Java-specific performance antipattern definitions, and there are several approaches that identify different bad practices in Java platforms and APIs [47], [48], [49], [50]. The difficulty of evaluating the performance of Java applications is acknowledged also in the testing domain; Java open source projects are usually subject to a limited number of performance tests that are rarely updated and typically maintained by a small group of developers [51]. In [52] a rule-based approach is proposed to detect performance antipatterns from runtime traces while targeting Java EE antipatterns. In [53] load testing and profiling data are exploited to detect bad practices in Java applications. This approach leverages performance experts to identify problematic snapshot(s), and the antipattern detection is performed (similarly to approaches dealing with performance regressions [23], [45]) comparing snapshots with the recognized problematic one(s).

To the best of our knowledge, there are few attempts in the direction of bridging performance and code-related issues with performance antipatterns. Our approach relies on dynamic analysis and adopts software performance antipatterns to identify bad practices arising when the system is in execution. The main difference w.r.t. work more closely related to ours [35], [53] is that we consider a plethora of seven software performance antipatterns applied to a broad set of real-world case studies. Experimental results are promising to foster further research.

3 Our Approach

This section introduces the approach proposed to detect software performance antipatterns in Java applications. The design of the approach is driven by the following key insight. Each performance antipattern describes a bad design practice that can be (partially) observed and checked by a combination of a particular set of system key properties including design characteristics (e.g., large number of calls) and performance metrics (e.g., long execution time). Our detection approach relies on monitoring these system key properties. A system component is recognized to be an antipattern when its design characteristics and/or performance metrics deviate most from the average values, as calculated considering all other components belonging to the same system.

Figure 1 depicts the workflow of our approach to automatically detect software performance antipatterns in Java applications. First, a load test suite is defined for each system, and it is used to simulate the interaction of users with the system under analysis. Then, the system under analysis is launched and the profiler is attached. We are aware that runtime performance monitoring of Java applications is expensive and the profiling process can generate overhead, however this is also assessed as a necessary task to collect the performance characteristics of interest [54]. After attaching the profiler, test suites are executed. During their execution, we periodically capture snapshots that contain the readings related to CPU, threads, and memory, later exported in CSV or XML files, according to the represented data. These files are provided as input to JPAD, i.e., the tool we developed for the automatic detection of software performance antipatterns.

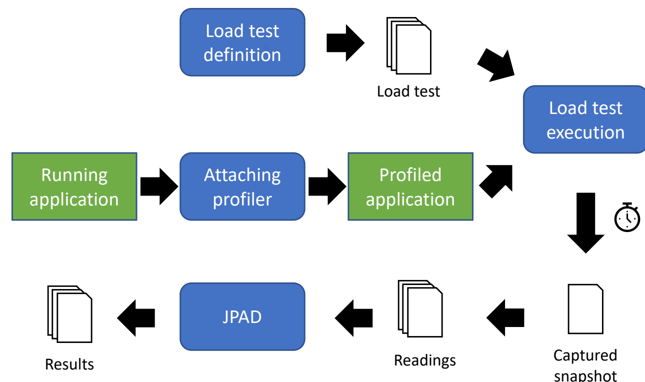


Fig. 1: High-level workflow of our approach.

In the sequel of the section, we detail the operational tasks of the proposed framework, i.e., how the load testing is performed, the technology used, and the criteria adopted for defining the test suite. Then, we present the application profiler used to monitor the systems under analysis during the execution of the load tests. We also describe the seven software performance antipatterns that we implement into JPAD to enable their automatic detection. Starting from the natural language definition of antipatterns, the detection rules are expressed in terms of the data acquired through the profiler. JPAD takes as input the readings from the profiler and makes use of such data to automatically perform the detection of software performance antipatterns.

JPAD is a JavaFX application that allows the user to load the profiling data exported from YourKit (i.e., the input files) and contains an embedded console where the results are reported. The tool works with thresholds and offsets that establish when performance issues arise. A threshold value t represents a boundary on a specific performance metric value v, and JPAD verifies if $v > t$. An offset value o is instead a percentage addendum that allows to deviate from systems' properties extracted as average values, i.e., a property on a specific system component (p) is compared with the average across all components ($\overline{P}$), and JPAD verifies if $p > \overline{P} + \overline{P} \cdot o\%$. Note that both thresholds and offsets are used to evaluate a specific test input. The goal is to detect those system characteristics that deviate, under the same input assumptions, with a certain margin from requirements and/or average values. For instance, 10% can be set as a threshold for the CPU usage, and JPAD considers suspicious all CPU resources showing larger values. As a further example, let us consider 5% as the offset on method number of calls. JPAD calculates the average value (considering the total number of calls across all the system methods) and augments such average by 5%. This way, JPAD detects as suspicious all the methods whose number of calls deviates by 5% from the average within a specific test input. Note that these threshold and offset settings can be

modified by users in case further knowledge is available and other values are considered more appropriate. Sensitivity analysis on threshold values is discussed in Appendix C.

3.1 Load testing

To define the required tests for each system under analysis, we explore the target application to understand key functionalities in execution. We are aware that designing load tests represents a threat to the internal validity of the approach. Indeed, we may miss some functional test cases, and there might be functionalities that are not monitored. Hence, there is a risk of not detecting some antipatterns and consequently not capturing performance issues. This threat is smoothed by delegating the design of load tests to software engineers that can decide which system functionalities require performance testing. To support software engineers in this task, as a rule of thumb, we foresee the selection of the software component(s) showing the highest CPU utilization as candidate to load testing. To support this guideline, in our experimentation (see Section 4) we show that such a selection leads to detect real-life performance issues.

3.2 Application profiling

To profile the applications under analysis during their execution, we use the YourKit Java Profiler [55]. This decision is motivated by several reasons. First, YourKit is widely used for the performance evaluation of real-world applications both in industry [56] and in academia [53], [57]. Second, precise instructions about reducing or avoiding performance overhead (generated from the profiler) are provided [58]. Third, YourKit is a powerful application profiler, it monitors several aspects of the profiled system, and provides a view for multiple system features. Besides, the output of YourKit (i.e., the readings, see Figure 1) can be easily exported in different data formats. For the purpose of this work, we monitor usage statistics: CPU usage, memory usage, CPU hotspots, call tree, blocked threads, and garbage collected objects. CPU and memory usage measure the percentage of used resources during the execution of the application. CPU hotspots are those methods that spend the longest time on the CPU. Call tree is divided in two different views: (i) a merged call tree that shows a top-down call tree of all application threads merged together into a single tree, and (ii) a call tree by thread that shows an individual top-down call tree for each application thread. From these files we can extract further information, e.g., callers and callees of each method. Blocked threads represent threads that fail to immediately enter the synchronized method/block. Garbage collected objects allow estimating the load of the garbage collector. All these statistics are fed to JPAD that uses them to detect software performance antipatterns.

3.3 Antipattern detection

The detection of software performance antipatterns relies on our interpretation of the natural language specification of software performance antipatterns [26]. Specifically, after extracting key properties, we implement the antipattern detection rule if a match with profiling data exists. Our effort is mainly devoted to match high-level guidelines and make them concrete for the inspection of Java applications. In this context, the difficulty relies on combining different sources of information (e.g., a class calling a high number of methods, high CPU usage), since the intrinsic nature of software antipatterns is to look for various performance problems that may arise when applications are running. Since the specification of antipatterns cannot be completely precise [26], the conditions and key properties we check as our detection rules approximate the antipatterns. These conditions are neither sufficient nor necessary, we empirically validate their relevance on spotting performance issues.

Table 1 reports a subset of seven software performance antipatterns proposed in [26], and we motivate in the following the choice of implementing them. The reason why some antipatterns cannot be automatically detected is argued in Appendix B. The first column reports the *name* of the performance antipatterns, and the second column describes the *problem* expressed in natural language. The third column lists the extracted *key properties* (along with the implemented helper functions) and provides a match with profiling data. The fourth column lists thresholds and offsets that are included in the detection algorithms. Helper functions are briefly described in Table 2, thresholds/offsets are presented in Table 3 along with the heuristics adopted in case users do not define their own preferences on perceived performance issues. Offsets are all set to 5% since we are interested to capture small deviations from average values. Thresholds are set to 10%, we refer to [59] where the CPU load in the *idle* phase is estimated to be 7% on average, and we are interested to exclude outliers and fluctuations that might be due to system's internal routines.

The detection algorithms and implementation details of these antipatterns are given in Appendix A, along with the match between textual descriptions and detection rules. In the following, we briefly discuss our interpretation of antipatterns, and their key properties.

Circuitous Treasure Hunt (CTH) occurs when a Java application must perform a large number of (database) queries to manage a request. This problem can be generalized considering a method that performs a chain of queries where the result of one query is used to build the next one, instead of writing a single, and more complex request. As key properties, we check the average number of calls performed by each thread (*call-tree-by-thread* view of YourKit), along with the average processor utilization (*chart-cpu-usage* view of YourKit) indicating if the system performance suffers.

Extensive Processing (EP) occurs when a long running job monopolizes the processor and creates a queue of processes that cannot be executed until the computation of such job is completed. As key properties, we monitor the average number of blocked threads (*monitor-usage* view of YourKit), and the execution time of Java methods (*call-tree-all-threads* view of YourKit), selecting only those methods that are showing a large number of the identified blocked threads.

Wrong Cache Strategy (WCS) occurs when too many objects (or objects hardly ever used) are cached. This leads to generate performance overhead resulting in high memory usage. As key properties, we check the average memory usage of methods (*method-list-allocation* view of YourKit), and we are interested to verify whether the memory is more used than the processor (*method-list-cpu* view of YourKit).

TABLE 1: Software Performance Antipatterns - key properties extracted from natural language specification [26].

Name	*Problem specification*	*Key properties (and helper functions)*	*Thresholds and offsets*
Circuitous Treasure Hunt	Occurs when an object must look in several places to find the information it needs. If a large amount of processing is required for each look, performance will suffer.	Large number of calls (*getMethodCountMap*, *getAvgMethodCount*), high execution time (*getAvgTime*), and high resources utilization (*getAvgCpuUsage*).	countOffset, cpuTh
Extensive Processing	Occurs when extensive processing monopolizes the processors and impedes the overall response time.	Large number of blocked threads (*countThreads*, *countBlockedThreads*), high execution time (*getTotExecTime*, *getAvgExecTime*, *getHsExecTime*).	execTimeOffset
Wrong Cache Strategy	Caching too many objects (or objects that are rarely used) quickly changes the advantage of caching into a disadvantage due to higher memory usage.	High memory usage from methods (*getAvgMemUsage*, *getHsMemUsage*), i.e., larger than the average processor usage (*getHsCpuUsage*).	memUsageOffset
Blob	Occurs when a single class either (i) performs all the work of an application or (ii) holds all the application data. Either manifestation results in excessive message traffic that degrades the performance.	Large number of calls performed and received by the methods (*getCallersMap*, *getCalleesMap*, *getAvg*) reflecting on high processor and/or memory utilization (*getAvgCpuUsage*, *getAvgMemUsage*).	msgOffset, cpuTh, memTh
Tower of Babel	Occurs when processes excessively convert, parse, and translate internal data into a common exchange format.	Large execution time of methods (*getMethodCallsMap*, *getTotExecTime*, *getAvgExecTime*, *getHsExecTime*) due to specific activities (*cpu-hotspots*).	execTimeOffset
Empty Semi Trucks	Occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both.	Large number of method calls (*getCalleesMap*, *getAvg*, *countCalls*) whose execution time (*getCvTime*) follows a deterministic distribution.	msgOffset
Excessive Dynamic Allocation	Occurs when an application unnecessarily creates and destroys large numbers of objects during its execution. The overhead has a negative impact on performance.	Large number of objects stored by the Java garbage collector (*getGcObjs*, *getHsGcObjs*) , high memory utilization (*getAvgMemUsage*).	memTh, gcObjectsOffset

TABLE 2: Software Performance Antipatterns - helper functions.

Name	*Description*
getMethodCountMap	It calculates the minimum, maximum, or average (depending on the *option* input parameter) number of calls in a method
getAvgMethodCount	It calculates the average number of calls in a specific hotspot method, to be compared with the average of all hotspots
getAvgTime	It calculates the average execution time of all hotspots, to be compared with the executime time of a specific hotspot method
getAvg[Cpu,Mem]Usage	It calculates the average utilization values showed by all system hardware (CPU or memory) resources
count[Blocked]Threads	It counts the number of (blocked) threads, to be compared with active threads to understand delays in the execution of methods
get[Tot,Avg,Hs]ExecTime	It retrieves the (total, average, or related to a specific hotspot method) execution time, to be compared with other methods
get[Callers,Callees]Map	It retrieves all methods that are callers or calles, and it is used to quantify the number of calls performed/received by a method
getAvg	It takes as input parameter a map of methods (e.g., through *getMethodCallsMap*, see next helper function) and calculates the average
getMethodCallsMap	It retrieves all methods that are invoked by other methods, to be compared with average values for spotting most invoked methods
countCalls	It takes as input a callee method and it calculates the average number of times that hotspots invoke their callees
getCvTime	It takes as input a callee method and it retrieves the coefficient of variation of the callee's service time
get[Hs]GcObjs	It retrieves the average number of garbage collected objects in the (hotspot) methods, thus to spot if there are many unused objects

TABLE 3: Software Performance Antipatterns - thresholds and offsets.

Name	*Description*	*Heuristic*
countOffset	Addendum for methods' number of calls	Set to 5%, it augments the average value for the number of methods' calls
cpuTh	Upper bound for CPU utilization	Set to 10%, lower values are considered part of the system's internal routines
execTimeOffset	Addendum for methods' execution time	Set to 5%, it augments the average value for the execution time of methods
memUsageOffset	Addendum for memory utilization	Set to 5%, it augments the average value for the memory resources utilization
msgOffset	Addendum for number of exchanged messages	Set to 5%, it augments the average value for the methods' exchanged messages
memTh	Upper bound for memory utilization	Set to 10%, lower values are considered part of the system's internal routines
gcObjectsOffset	Addendum for number of garbage collected objects	Set to 5%, it augments the average value for the number of garbage collected objects

Blob occurs in two different scenarios that contribute to distinguish two different types of antipattern instances in Java applications, i.e., *Blob-Controller* and *Blob-DataContainer*. The former is observed when a class centralizes many responsibilities, delegating minor roles to other classes. Classes affected by this problem usually are complex controllers which depend on simpler classes (with little to no computation). The latter case is observed when a class

includes most of the application data and other functions need to access that class to retrieve/update the data. As key properties, we monitor the average number of calls performed and received by the methods (*call-tree-all-threads* view of YourKit) to determine if these methods can be classified as potential controllers or data containers, respectively. Besides, we also check if such large number of calls impacts on the average usage of processor and memory (*chart-cpu-usage* and *chart-heap-memory-usage* views of YourKit).

Tower of Babel (ToB) occurs when some data is translated into an exchange format, such as XML, by the sending process. This data is later parsed and translated into an internal format by the receiving process. This means that the system may spend most of its time processing the text when the translation and parsing of data formats are excessive. As key properties, we check all the methods showing a large execution time (*call-tree-all-threads* view of YourKit), and then we inspect the names of such methods (*cpu-hotspots* view of YourKit). We check if there is a match with some specific keywords (i.e., "converse", "parse", and "translate"), thus to associate the performance overhead to the processing of exchange format data.

Empty Semi Trucks (EST) occurs when an excessive number of requests is required to perform a task. As key properties, similarly to the Blob-controller, we consider the number of calls performed by the methods (*call-tree-all-threads* view of YourKit), and we verify if the execution time of such calls shows a small coefficient of variation, i.e., the execution time follows a deterministic distribution (*call-tree-all-threads* view of YourKit). This way, we aim to capture the peculiarity of this antipattern when inefficiently using the bandwidth and/or interfaces.

Excessive Dynamic Allocation (EDA) occurs when an application unnecessarily creates and destroys objects. As key properties, we monitor the objects collected by the Java garbage collector (*method-list-garbage* view of YourKit), and we want to recognize those situations leading to a performance overhead, i.e., the average memory utilization (*chart-heap-memory-usage* view of YourKit) is high.

It is worth remarking that our detection algorithms may be *orthogonal*, i.e., they may return the simultaneous occurrence of multiple antipatterns since they share the verification of some performance indices, e.g., the CPU utilization. This is not a drawback of the approach since it may happen that a performance issue may be caused by the simultaneous presence of multiple bad practices [60].

4 EXPERIMENTATION

This section is organized as follows. We first present the research questions (Section 4.1), followed by the description of the analyzed real-world applications (Section 4.2). We describe the experiment design (Section 4.3), and we discuss the obtained experimental results (Section 4.4).

4.1 Research questions

The purpose of our experimental evaluation is threefold: (i) it shows that JPAD is efficient and accurate when applied to real medium/large-sized systems; (ii) it compares JPAD with state-of-the-art approaches on the detection of real-life performance issues; (iii) it provides empirical evidence on the usefulness of detecting antipatterns. In particular, we aim to answer three research questions:

- RQ_1 *Efficiency and accuracy of the antipatterns' detection rules:* Are the proposed detection rules efficient and accurate? Does JPAD efficiently detect antipattern instances in real-world case studies? **Motivation.** We want to provide developers with an efficient and accurate framework that points out performance issues, if any. By evaluating the efficiency of JPAD, we can understand whether developers are motivated to apply our approach in practice.
- RQ_2 *Comparison with state-of-the-art approaches:* Are the proposed detection rules comparable to other techniques in the literature? Does JPAD capture complex performance problems that are not recognized by available tools? **Motivation.** We want to compare our framework with state-of-the-art approaches that deal with the detection of performance issues. By comparing JPAD to other tools, we can study whether developers are motivated to use our framework.
- RQ_3 *Implication of applying antipattern-based refactorings:* What happens when an antipattern is solved? How does the number of detected antipattern instances for the refactored system change? What is the effect on the overall system performance? **Motivation.** The goal of our approach is to spot performance problems, so that developers are aware of possible shortcomings in some portions of the code. In this research question, we aim to answer at which extent our approach benefits developers interested to know if code fixings improve the system performance.

To answer the first two research questions, we analyze real-world Java applications under different loads, i.e., varying the number of clients and the execution time. For each combination of (i) number of clients and (ii) duration (in minutes), the application is profiled, data is collected, and JPAD is used to detect the software performance antipatterns. Moreover, to further stress the benefit of the proposed framework and to answer the third research question, we provide empirical evidence of its impact by solving one instance of detected antipatterns (for one of the analyzed systems), and showing variations in the number of detected instances and the system performance.

4.2 Analyzed systems

We select five Java applications that are highly concerned with their performance and have been studied in prior research [46], [61], [62]. These applications show a different complexity (in terms of number of classes) and belong to different domains, see Table 4 for their main characteristics. The rationale for selecting these five subject systems is that they provide evidence of all the seven performance antipatterns, as later described in the experimental results (see Section 4.4). Specifically, systems from [46] do not include ToB and EST antipattern instances. We select one system from [61] to provide evidence on the ToB antipattern. To strengthen the analysis of the EST antipattern, we borrow the subject system used in [62]. Hereinafter, a brief description of the analyzed systems is provided.

TABLE 4: Analyzed systems.

Name	*Version*	*Domain*	*# of Classes*
CloudStore	2	E-commerce	68
TeaStore	1.4.1	Microservices	138
WebGoat	8.0.0.M26	E-learning	301
TrainTicket	1.0	Microservices	584
OpenMRS	2.9.0	Medical Record System	1093

TABLE 5: Analyzed issues extracted from Chen et al. [23].

Name	*Issue ID*	*Issue fixing commit*	*Issue inducing commit*
Hadoop	YARN-4307	308d63f	e914220 7af5d6b
	YARN-7102	ff8378e	528b809
	HDFS-12754	738d1a2	decf8a6
Cassandra	CASSANDRA-13794	f93e6e3	88d2ac4

- CloudStore [63] is a free software synchronizing files between multiple devices. Its primary focus is on preventing data loss and unauthorized access.
- TeaStore [64] emulates a basic web store for automatically generated tea and tea supplies. It has been published in [65], and later largely used as a microservice reference system and test application.
- WebGoat [66] is a deliberately insecure web application maintained by OWASP and designed to teach web application security lessons.
- TrainTicket [67] is the largest benchmark for microservice architectures in the literature. It provides train ticket booking functionalities and is used for fault analysis and error prediction [62], [68].
- OpenMRS [69] is a free medical record system for health care providers. It is a modular open-source web application used by over 40 countries to improve health care delivery in resource-constrained environments.

Moreover, to compare JPAD with state-of-the-art approaches [23], [45], [53], we consider four real-life performance issues (see details in Table 5) of two other systems:

- Hadoop [70] is a framework which performs data processing in a reliable, efficient, high fault tolerance, low cost, and scalable manner.
- Cassandra [71] is a distributed NoSQL database management system; fault-tolerance on commodity hardware makes it suitable for mission-critical data.

Table 5 reports the code commits (inducing and fixing the four real-life performance issues) that we analyze, according to the study on performance regressions presented in [23]. The motivation of selecting these specific four performance issues is the following. Hadoop commits are evaluated in [23] by means of multiple performance metrics (i.e., response time, CPU and memory utilization, I/O operations) and we focus on those issues that have been highlighted as particularly complex, since such issues are not predicted by any of the considered metrics. About Cassandra, there is only one issue that is not predicted by *PerfJIT* in [23], and this is why we concentrate our effort on investigating that specific issue. Summarizing, our investigation includes those specific four issues (triggering the analysis of nine code commits) since they are more relevant to conduct a comparison.

TABLE 6: Threshold values used for analyzing systems with #clients $= 25$ and duration $= 3$ minutes.

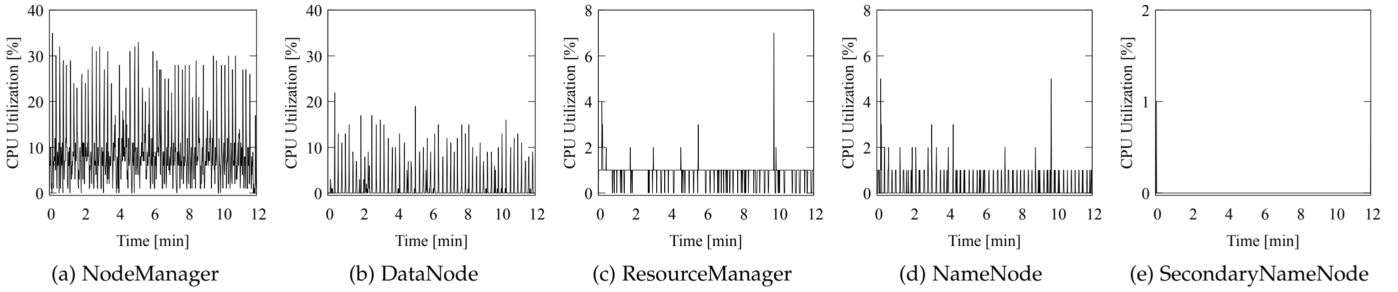
Antipatterns	*Thresholds*	*Systems* CloudStore	TeaStore	WebGoat	TrainTicket	OpenMRS
CTH	*countTh*	309	148	170	1198	108
	cpuTh	10%	10%	10%	10%	10%
	option	avg	avg	avg	avg	avg
EP	*execTimeTh*	5.29%	5.40%	5.88%	5.00%	5.10%
WCS	*memUsageTh*	16%	16%	13%	27%	76%
Blob	*callersTh*	19.36	24.00	4.76	9.28	8.58
	calleesTh	426.75	427.12	596.05	2637.25	1562.40
	cpuTh	10%	10%	10%	10%	10%
	memTh	10%	10%	10%	10%	10%
ToB	*execTimeTh*	5.29%	5.40%	5.88%	5.00%	5.10%
EST	*msgTh*	1.05	1.05	1.40	1.47	1.58
EDA	*gcObjectsTh*	8711349.8	4387604.0	1562411.2	3652923.5	3538209.5
	memTh	10%	10%	10%	10%	10%

4.3 Experimental setup

In the following, we discuss the design choices taken to run experiments and avoid biases in results.

System workload specification. To avoid biases when profiling applications, we test several workloads acting in the considered systems for a different duration. The choice on the number of clients and the duration of load testing is not trivial, however our assumption is that this is decided by software developers that are aware of load conditions and system's dynamics. Subject systems in Table 4 run for 3, 6, and 12 minutes with 25, 50, 75, and 100 clients. A larger variation in the number of clients is considered for the issues listed in Table 5 and related to Hadoop and Cassandra; these systems are tested with 1, 10, 100, 500, and 1k clients running for 3, 6, and 12 minutes. Each combination of number of clients (C) and duration of the testing (D) represents an input we use in our detection. Such a combination leads to a system configuration that is labeled in the following as $C-D$, for instance the system configuration "1000-6" means to consider 1000 clients with 6-minute duration of testing. Overall, 300 different system configurations are analyzed to explore a variegate set of systems' characteristics.

Load test definition. To avoid biases in the load testing, we explore benchmarks stressing different aspects (e.g., input/output operations, end-users services) of considered applications. For instance, we make use of available benchmarks for Hadoop and Cassandra (i.e., TestDFSIO [72] and Cassandra Stress [73]) stressing write and read operations. Locust [74] (i.e., a Python-based load testing tool) is adopted for systems reported in Table 4 since ready-to-use benchmarks are not available. We stress end-users services identified as crucial for the application, e.g., in TrainTicket we generate requests for monitoring the security service invoked when users make a reservation. As anticipated in Section 3.1, to support software engineers in the selection of system functionalities to be monitored, as a rule of thumb, we consider the CPU utilization of system



(a) NodeManager (b) DataNode (c) ResourceManager (d) NameNode (e) SecondaryNameNode

Fig. 2: CPU usage over 12 minutes for the five Hadoop components. The profiled Hadoop commit is *decf8a6*.

TABLE 7: Threshold values used for analyzing Hadoop (two components) and Cassandra with #clients = 1k and duration = 12 minutes. The symbol "–" indicates that a threshold is not calculated due to lack of data.

Antipatterns	*Thresholds*	*Hadoop – NodeManager*							*Hadoop – DataNode*							*Cassandra*	
		decf8a6	e914220	7af5d6b	528b809	308d63f	738d1a2	ff8378e	decf8a6	e914220	7af5d6b	528b809	308d63f	738d1a2	ff8378e	88d2ac4	f93e6e3
CTH	*countTh*	–	–	–	–	–	–	–	–	–	–	–	–	–	–	44	26
	cpuTh	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%
	option	avg	avg	avg	avg	avg	avg	avg	avg	avg	avg	avg	avg	avg	avg	avg	avg
EP	*execTimeTh*	5.06%	5.04%	5.06%	5.06%	5.04%	5.02%	5.02%	5.29%	5.21%	5.26%	5.25%	5.39%	5.32%	5.28%	5.24%	5.25%
WCS	*memUsageTh*	5.00%	5.00%	5.00%	5.00%	5.00%	5.00%	5.00%	5.00%	5.00%	5.00%	5.00%	5.00%	5.00%	5.00%	13.00%	26.00%
Blob	*callersTh*	11.12	11.95	11.89	11.33	10.15	10.26	9.56	2.42	2.91	2.51	2.59	2.04	2.28	2.31	4.61	4.51
	calleesTh	1.05	1.05	0.70	1.05	0.70	–	–	–	–	–	–	–	–	–	82.35	44.28
	cpuTh	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%
	memTh	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%
ToB	*execTimeTh*	5.06%	5.04%	5.06%	5.06%	5.04%	5.02%	5.02%	5.29%	5.21%	5.26%	5.25%	5.39%	5.32%	5.28%	5.24%	5.25%
EST	*msgTh*	–	–	–	–	–	–	–	–	–	–	–	–	–	–	1.05	1.05
EDA	*gcObjectsTh*	–	–	–	–	–	–	–	–	–	–	–	–	–	–	15980736.0	16730745.2
	memTh	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%

software components. This analysis provides support to the component choosing stage of the approach that decides which component(s) to test, and they are given as input to our detection. As an example, let us consider Figure 2 that shows the CPU utilization (observed when executing I/O operations) of all Hadoop components. Results show that two components have a non negligible utilization, see Figures 2(a)–(b) with CPU utilization varying up to 40%: (i) the *NodeManager* which is in charge of monitoring the resource usage of a node and reporting such information to the *ResourceManager* and (ii) the *DataNode* that stores data. Hence, these two components are selected to be profiled. Figures 2(c)–(e) instead indicate that there are underutilized components: *ResourceManager* and *NameNode* show a CPU utilization lower than 8%, whereas the CPU used by *SecondaryNameNode* is even less than 2%. This denotes that it is not relevant to profile such components.

Load test execution. To avoid biases in the obtained results, all experiments are run on a single node cluster deployed on a dedicated machine (with a 2.80 GHz quad-core CPU and 16 GB memory) to easily monitor used resources and avoid misleading performance results due to background activities. Readings extracted from CPU, memory, and garbage collector snapshots taken for each configuration are fed to JPAD that automatically detects performance antipatterns.

Antipattern detection and thresholds setup. We recall that most of thresholds are calculated through heuristics that consider average values and offsets, see Table 3. This means that each system configuration includes thresholds that vary when changing number of clients and duration. In the following, as illustrative examples, we report the thresholds values used with the lowest (25 clients, 3 minutes) and the largest (1k clients, 12 minutes) boundary values adopted in the proposed experimentation. Table 6 reports the threshold numerical values used to analyze the systems when setting the number of clients to 25 and the duration of the run to 3 minutes. In the first row of Table 6 we can notice that *countTh* threshold is calculated on average values, and it shows a large variation among the analyzed systems (e.g., 108 for OpenMRS, and 1198 for TrainTicket). There are other thresholds that vary less even if they are calculated with average values, e.g., *msgTh* varies between 1.40 and 1.58. Some other thresholds are instead fixed to values that usually are extracted from system requirements, e.g., *memTh* is fixed to 10% for all the systems. Table 7 reports thresholds used for all commits of Hadoop (*NodeManager* and *DataNode*) and Cassandra, when systems are loaded with 1k clients for 12 minutes. Note that some commits do not provide data needed to calculate the thresholds, e.g., *738d1a2* and *ff8378e* lack information for deriving *callesTh* related to the Blob antipattern disabling its detection. Besides, the same threshold may vary differently based on the considered applications. For example, *callersTh* related to the Blob antipattern changes significantly when considering the two Cassandra commits (i.e., 82.35 for *88d2ac4* and 44.28 for *f93e6e3*) and shows slight variations when observing each Hadoop component separately. Instead, relevant variations of this threshold are observed also for Hadoop when the

two components are compared, i.e., it varies in the range 9.56–11.95 for the *NodeManager* and in the range 2.04–2.91 for the *DataNode*. Threshold and offset values are given as input to JPAD, and these values can be easily modified in case system stakeholders express their own performance requirements.

Summarizing, our experimental setup includes the following inputs: (i) system workload specification, i.e., number of clients and duration of the testing, (ii) load test definition, i.e., which component(s) to test, and (iii) threshold and offset setup, i.e., numerical values for antipatterns' thresholds and offsets. Our experimentation investigates the variation of these inputs and how they affect the detection accuracy and results, see more details in Tables 8–12, and Appendix C.

4.4 Experimental results

This section presents experimental results answering our three research questions (see Section 4.1).

4.4.1 Effectiveness of the antipatterns' detection rules

To answer RQ_1, Table 8 shows the software performance antipattern instances detected by JPAD for each system, when varying the number of clients and the monitoring duration. The last column of Table 8 shows the detection time (in seconds), i.e., the time required by JPAD to complete the analysis. This way, we aim to analyze the scalability of the tool when exposed to medium- and large-sized systems.

We observe that the number of detected antipatterns may increase with the number of clients. For example, for the WebGoat case study, the number of detected antipatterns with 25 clients (i.e., 3, 6, 0, 5, 4, 0, 3) is smaller than (or equal to) the case with 100 clients (i.e., 4, 7, 2, 5, 4, 0, 4). The number of detected instances can also decrease for a larger number of clients, e.g., OpenMRS shows 2 instances of the CTH antipattern with 25 clients and 0 with 100 clients. However, we notice that this may lead to generate further instances of different antipatterns types, in fact with 100 clients we get 3 Blob instances instead of 2 (observed with 25 clients). Similarly, CloudStore shows 7 instances of the EP antipattern and 8 instances of the Blob antipattern with 25 clients. With 100 clients instead we get 6 instances of the EP antipattern and 7 instances of the Blob antipattern, at the cost of 2 instances of the EDA antipattern (not observed with 25 clients). This may be due to performance issues showing up to (previously not critical) parts of the system, i.e., affecting different system elements only when the number of clients increases. As a result, a larger workload may produce more instances of other antipattern types.

A longer experiment duration might reduce the number of detected instances due to the performance problems flattening/elevating in different parts of the system depending on the application peculiarities. In our experiments, OpenMRS benefits from a longer run time, whereas all other systems (especially TeaStore) generally show a constant or higher number of antipattern instances. The number of antipattern instances detected when OpenMRS is run for 3 minutes with 25 clients (i.e., 11) decreases when the same application is observed for 12 minutes (i.e., 3). For all other systems, it is unlikely that the number of detected instances decreases when the experiment duration increases. In a few cases, the opposite trend is observed, e.g., there are 21 antipattern instances when WebGoat is run with 25 clients for 3 minutes and 25 instances when the same application is run for 12 minutes. A further example is represented by the CloudStore system that shows 17 instances with 100 clients for 3 minutes and 19 instances when running for 12 minutes.

From the collected results, no straightforward relationship between the number of clients and the number of detected antipatterns is observed. Similar observations are drawn when considering the experiment duration since

TABLE 8: Analysis of how *system workload specification* inputs affect the detection. Number of detected antipattern instances and scalability of JPAD across the analyzed systems.

	Config.	*CTH*	*EP*	*WCS*	*Blob*	*ToB*	*EST*	*EDA*	*time (sec)*
CloudStore	25-3	2	7	0	8	0	0	0	1.9
	25-6	2	7	0	8	0	0	2	0.9
	25-12	2	6	0	7	0	0	2	1.0
	50-3	2	8	0	8	0	0	0	1.1
	50-6	2	7	0	7	0	0	0	1.1
	50-12	2	8	0	9	0	0	0	1.5
	75-3	2	6	0	7	0	0	0	1.0
	75-6	2	7	2	8	0	0	0	1.2
	75-12	2	6	0	7	0	0	0	1.6
	100-3	2	6	0	7	0	0	2	1.1
	100-6	2	7	2	8	0	0	2	1.4
	100-12	2	6	2	7	0	0	2	1.8
TeaStore	25-3	6	9	0	6	0	0	2	144.1
	25-6	6	9	0	6	0	0	2	166.8
	25-12	6	9	0	6	0	0	2	183.6
	50-3	6	9	0	6	0	0	2	103.8
	50-6	6	9	0	6	0	0	2	158.2
	50-12	6	9	0	6	0	0	2	178.8
	75-3	5	9	0	5	0	0	2	172.8
	75-6	6	9	0	6	0	0	2	245.4
	75-12	6	9	0	5	0	0	2	218.3
	100-3	5	9	0	6	0	0	2	282.5
	100-6	6	9	0	5	0	0	2	257.6
	100-12	6	9	0	5	0	0	2	220.3
WebGoat	25-3	3	6	0	5	4	0	3	29.8
	25-6	3	6	0	5	4	0	3	30.6
	25-12	4	7	0	6	4	0	4	29.9
	50-3	4	7	2	5	4	1	4	33.6
	50-6	4	7	1	5	4	1	4	37.4
	50-12	4	7	2	5	4	0	4	34.3
	75-3	3	6	1	5	4	0	3	30.1
	75-6	4	7	2	5	4	0	4	34.4
	75-12	4	7	2	5	4	0	4	33.7
	100-3	4	7	2	5	4	0	4	32.6
	100-6	4	7	2	5	4	0	4	30.6
	100-12	4	7	2	5	4	1	4	30.2
TrainTicket	25-3	1	6	1	1	0	1	0	1.0
	25-6	1	4	2	1	0	1	0	1.1
	25-12	1	5	2	1	0	1	0	1.1
	50-3	1	4	0	1	0	1	0	1.1
	50-6	1	4	0	1	0	1	0	1.2
	50-12	1	5	1	1	0	2	0	1.4
	75-3	1	6	1	1	0	1	0	1.3
	75-6	1	5	1	1	0	1	0	1.2
	75-12	1	5	1	1	0	1	0	1.4
	100-3	1	5	0	1	0	1	0	1.2
	100-6	1	5	0	1	0	1	0	1.3
	100-12	1	5	0	1	0	1	0	1.7
OpenMRS	25-3	2	3	2	2	0	0	2	0.6
	25-6	0	0	0	1	0	0	1	0.5
	25-12	0	0	1	1	0	0	1	0.7
	50-3	0	1	0	1	0	0	0	0.5
	50-6	0	1	0	1	0	0	0	0.5
	50-12	0	1	0	1	0	0	0	0.6
	75-3	1	2	1	3	0	0	1	0.6
	75-6	1	2	1	3	0	0	1	0.7
	75-12	0	1	0	1	0	0	0	0.6
	100-3	0	2	0	3	0	0	1	0.4
	100-6	0	1	0	1	0	0	0	0.6
	100-12	0	1	0	1	0	0	0	0.6

such duration is not linearly related with the number of detected antipattern instances, thus to confirm the complexity in the performance analysis of Java applications.

Table 8 also reports the time required by JPAD to detect antipatterns, and we can notice that it varies between 0.4 and 282.5 seconds (i.e., 4.7 minutes). The average detection time observed over all systems and loads is 45.9 seconds (i.e., less than a minute) with OpenMRS and TeaStore taking the shortest and longest average detection time (i.e., 0.6 and 194.4 seconds corresponding to 3.2 minutes), respectively.

Since TeaStore and WebGoat show the longest time to complete the detection, we further inspect these systems. For TeaStore we notice that Blob, ToB, and EST take longer than other antipatterns to be analyzed. This is due to the large size of the call tree (i.e., a key property analyzed to detect these antipatterns, see Table 1) of the TeaStore system. When Blob, ToB, and EST are excluded from the analysis, JPAD takes only 1.2 seconds on average to analyze TeaStore. For WebGoat, we observe that the detection of EST antipattern takes longer than others, i.e., the detection takes 3.5 seconds on average when EST is excluded. We think this is due to the large number of methods that are invoked (namely the *callees*) in this application. In fact, these methods must be analyzed by the EST detection algorithm to check their coefficient of variation, i.e., if the execution time follows a deterministic distribution, thus to spot inefficiency on the usage of resources.

Inspired by [46], we build a *ground truth* to investigate the accuracy (along with precision, recall, and F1 score) of our detection rules. The ground truth is constituted of all the hotspot methods (i.e., over approximation of the detection outcome) since JPAD analyzes their design and performance characteristics to detect antipatterns. We exploit the variation in the system workload specification inputs to decide if a hotspot method is correctly detected as source of performance problems. We determine that a hotspot method has performance issues if it is reported as a violation of antipatterns while analyzing different system configurations. If a hotspot method is detected as an antipattern instance in at least half (i.e., 6 over a total of 12) analyzed configurations, then such a method is considered as a positive instance of that antipattern, otherwise it is classified as a negative instance. Let us consider as example `executeQuery()` that is a hotspot method in OpenMRS. It is detected as EP antipattern in 10 (out of 12) analyzed configurations, hence such method is considered as a positive instance of EP in OpenMRS. Another hotspot method of OpenMRS is `includeFragment(String, String)` but it is detected as EP antipattern in only 2 (out of 12) analyzed configurations. Consequently, this method is considered as a negative instance.

The *confusion matrix* is built for each system configuration as follows. True positives (TP) are the methods for which an antipattern instance is detected by both the ground truth and JPAD. True negatives (TN) are those hotspot methods that are detected as specific antipattern instances by neither the ground truth nor JPAD. False positives (FP) are defined by calculating hotspot methods that are detected as antipatterns by JPAD, but are not in the ground truth. False negatives (FN) are calculated by counting methods that are not detected by JPAD but are in the ground truth.

TABLE 9: Confusion matrix of all considered configurations and applications when detecting the Blob antipattern.

	CloudStore				TeaStore				WebGoat				TrainTicket				OpenMRS			
	TP	TN	FP	FN	TP	TN	FP	FN	TP	TN	FP	FN	TP	TN	FP	FN	TP	TN	FP	FN
25-3	8	21	0	0	6	23	0	0	4	25	1	1	1	16	0	0	1	11	1	0
25-6	8	21	0	0	6	23	0	0	4	25	1	1	1	16	0	0	1	12	0	0
25-12	7	21	0	1	6	23	0	0	5	25	1	0	1	16	0	0	1	12	0	0
50-3	8	21	0	0	6	23	0	0	5	26	0	0	1	16	0	0	1	12	0	0
50-6	7	21	0	1	6	23	0	0	5	26	0	0	1	16	0	0	1	12	0	0
50-12	8	20	1	0	6	23	0	0	5	26	0	0	1	16	0	0	1	12	0	0
75-3	7	21	0	1	5	23	0	1	4	25	1	1	1	16	0	0	1	10	2	0
75-6	8	21	0	0	6	23	0	0	5	26	0	0	1	16	0	0	1	10	2	0
75-12	7	21	0	1	5	23	0	1	5	26	0	0	1	16	0	0	1	12	0	0
100-3	7	21	0	1	6	23	0	0	5	26	0	0	1	16	0	0	1	10	2	0
100-6	8	21	0	0	5	23	0	1	5	26	0	0	1	16	0	0	1	12	0	0
100-12	7	21	0	1	5	23	0	1	5	26	0	0	1	16	0	0	1	12	0	0

The confusion matrix for the analyzed system configurations in all considered applications (when detecting the Blob antipattern) is reported in Table 9. Confusion matrices derived for other antipatterns are omitted for the sake of space, but they are reported as part of replication data [28].

Accuracy, precision, and recall are defined for the detection of each antipattern and calculated for the analyzed system configurations as follows: $A_{conf} = \frac{TP+TN}{TP+TN+FP+FN}$, $P_{conf} = \frac{TP}{TP+FP}$, and $R_{conf} = \frac{TP}{TP+FN}$, respectively. P_{conf} and R_{conf} are *undefined* when their denominators are equal to zero. Accuracy, precision, and recall (i.e., A, P, and R, respectively) are also derived for the considered applications by averaging the results calculated for each configuration (M_{conf}), i.e.,

$$M = \begin{cases} \frac{\sum_{conf} M_{conf}}{\#conf} & \textit{if } M_{conf} \neq \textit{undefined } \forall \textit{ conf} \\ \textit{undefined} & \textit{otherwise} \end{cases},$$

where $M = \{A, P, R\}$ and $\#conf = 12$, i.e., the number of analyzed system configurations in our experimentation. F1 score is defined as the *harmonic mean* of precision and recall, i.e., $F1 = 2 * \frac{P*R}{P+R}$, and it is not computed in case P or R are undefined.

Table 10 reports accuracy (A), precision (P), recall (R), and F1 score (F1) of the proposed detection rules. It is worth remarking that our calculations leverage the variation in *system workload specification* inputs since the ground truth (by construction) is an over approximation and TP, TN, FP, and FN values keep into account how these inputs affect the detection. The average accuracy (across the five systems) is well above 90% for all antipatterns. The average precision is also above 90%, the lowest value (i.e., 79%) is observed in OpenMRS for the Blob antipattern. The average recall is mostly larger than 90% except for WCS that shows a lower value (62%); the lowest recall (58%) is observed in TrainTicket for WCS. The F1 score shows the lowest value (i.e., 88%) for Blob in OpenMRS as a reflection of the previous result on precision, even if average values are larger than 90% for all antipatterns.

RQ$_1$: efficiency and accuracy

The proposed detection rules efficiently and accurately capture performance issues of medium- and large-sized systems. CTH, EP, and Blob are the antipattern types that occur in all the analyzed systems. EP shows the largest number of instances across all the system configurations. WCS is the antipattern type with the smallest number of instances. JPAD efficiently detects the instances of the presented software performance antipatterns, in fact the system configurations are analyzed, on average, in less than a minute. TeaStore shows a longer detection time, but in the worst case it is less than 5 minutes. The F1 score, derived from precision and recall metrics, is larger than 85% in all the considered cases, denoting accurate detection rules.

4.4.2 *Comparison with state-of-the-art approaches*

The goal of this section is to investigate if JPAD is able to detect a variation on the number of antipattern instances when comparing the code commits that are known from [23] to induce and fix real-life performance issue, respectively.

TABLE 10: JPAD detection performance. The symbol "–" means that the metric is undefined.

		CTH	*EP*	*WCS*	*Blob*	*ToB*	*EST*	*EDA*
CloudStore	A	1.00	0.98	0.98	0.98	1.00	1.00	0.97
	P	1.00	0.98	–	0.99	–	–	–
	R	1.00	0.93	–	0.94	–	–	–
	F1	1.00	0.95	–	0.96	–	–	–
TeaStore	A	0.99	1.00	1.00	0.99	1.00	1.00	1.00
	P	1.00	1.00	–	1.00	–	–	1.00
	R	0.97	1.00	–	0.94	–	–	1.00
	F1	0.99	1.00	–	0.97	–	–	1.00
WebGoat	A	0.99	0.99	0.98	0.98	1.00	0.99	0.99
	P	1.00	1.00	–	0.94	1.00	–	1.00
	R	0.94	0.96	0.67	0.95	1.00	–	0.94
	F1	0.97	0.98	–	0.94	1.00	–	0.97
TrainTicket	A	1.00	0.98	0.97	1.00	1.00	1.00	1.00
	P	1.00	0.97	–	1.00	–	0.96	–
	R	1.00	0.95	0.58	1.00	–	1.00	–
	F1	1.00	0.96	–	1.00	–	0.98	–
OpenMRS	A	0.97	0.96	0.97	0.96	1.00	1.00	0.96
	P	–	–	–	0.79	–	–	–
	R	–	0.83	–	1.00	–	–	–
	F1	–	–	–	0.88	–	–	–
Avg.	A	0.99	0.97	0.97	0.98	1.00	1.00	0.98
	P	1.00	0.98	–	0.93	1.00	0.96	1.00
	R	0.98	0.92	0.62	0.97	1.00	1.00	0.94
	F1	0.99	0.97	–	0.95	1.00	0.98	0.97

Table 11 reports the performance antipattern instances that have been found in Hadoop across 7 different code commits. JPAD takes 0.31 seconds on average to analyze these configurations, and it detects EP and Blob antipatterns only, other antipatterns are not captured.

Table 11(a) focuses on the issue identified by *YARN-4307* that is not predicted by state-of-the-art approaches [23], [45]. Interestingly, we can notice that the selected software component(s) show a different number of antipattern instances. The column reporting the fixing of the issue (i.e., *308d63f*) always shows an equal or lower number of antipattern instances in all the cases for both the considered *NodeManager* and *DataNode* components. For the *NodeManager* we can notice that there are some configurations (e.g., 100-3 and 100-6) where EP instances are not detected in the commit fixing the issue, at the cost of emerging Blob antipattern instances. There are some further configurations (e.g., 500-3 and 500-6) for the *NodeManager* where we can notice no variation for the EP, one instance is detected in all commits. Overall, commits inducing the issue (*e914220* and *7af5d6b*) show a total of 13 and 12 EP instances across all the analyzed configurations, respectively. The commit fixing the issue manifests less antipattern instances, i.e., 4 EP and 2 Blob instances. The *DataNode* component instead has a different number of EP and Blob instances. Specifically, JPAD detects a total of 14 EP instances summing up all configurations independently of the commit. 9 and 8 Blob instances are detected for commits inducing the issue (i.e., *e914220* and *7af5d6b*, respectively), and 4 Blob instances are found in the commit solving the issue.

Table 11(b) reports the results for the YARN-7102 issue. The commit fixing the issue (i.e., *ff8378e*) shows a behavior similar to the previous case, i.e., there are some configurations (i.e., 10-12, 100-6, 1000-3) of *NodeManager* for which EP is solved at the cost of a new Blob instance. EP

TABLE 11: Analysis of how *load test definition* inputs affect the detection. Number of EP and Blob antipatterns detected using JPAD in two components (i.e., *NodeManager* and *DataNode*) and seven commits of Hadoop. EP and Blob instances are reported only, JPAD does not detect other antipatterns. JPAD takes 0.31 seconds on average for the detection of antipatterns in these configurations.

(a) YARN-4307. Commits *e914220* and *7af5d6b* introduce the issue, commit *308d63f* solves it [23].

Config.	*NodeManager*						*DataNode*					
	e914220		*7af5d6b*		*308d63f*		*e914220*		*7af5d6b*		*308d63f*	
	EP	Blob	EP	Blob	EP	Blob	EP	Blob	EP	Blob	EP	Blob
1-3	1	0	1	0	0	0	1	0	1	0	1	0
1-6	0	0	0	0	0	0	0	0	0	0	0	0
1-12	0	0	0	0	0	0	1	0	1	0	1	0
10-3	1	0	1	0	0	0	1	1	1	0	1	0
10-6	1	0	1	0	0	0	1	1	1	1	1	1
10-12	1	0	1	0	0	0	1	1	1	1	1	1
100-3	1	0	1	0	0	1	1	1	1	1	1	1
100-6	1	0	1	0	0	1	1	1	1	1	1	1
100-12	1	0	1	0	0	0	1	1	1	1	1	0
500-3	1	0	1	0	1	0	1	1	1	1	1	0
500-6	1	0	1	0	1	0	1	1	1	1	1	0
500-12	1	0	0	0	0	0	1	1	1	1	1	0
1000-3	1	0	1	0	1	0	1	0	1	0	1	0
1000-6	1	0	1	0	1	0	1	0	1	0	1	0
1000-12	1	0	1	0	0	0	1	0	1	0	1	0

(b) YARN-7102. Commit *528b809* introduces the issue, commit *ff8378e* solves it [23].

Config.	*NodeManager*				*DataNode*			
	528b809		ff8378e		528b809		ff8378e	
	EP	*Blob*	*EP*	*Blob*	*EP*	*Blob*	*EP*	*Blob*
1-3	1	0	0	0	1	0	0	0
1-6	0	0	0	0	0	0	0	0
1-12	0	0	0	0	0	0	0	0
10-3	1	0	0	0	1	1	0	0
10-6	1	0	0	0	1	1	1	0
10-12	1	0	0	1	1	1	1	0
100-3	1	0	0	0	1	1	1	0
100-6	1	0	0	1	1	1	1	0
100-12	0	0	0	0	1	1	1	0
500-3	1	0	1	0	1	1	1	1
500-6	1	0	0	0	1	1	1	0
500-12	0	0	0	0	1	1	1	0
1000-3	1	0	0	1	1	0	1	0
1000-6	1	0	1	0	1	0	1	0
1000-12	0	0	0	0	1	0	1	0

(c) HDFS-12754. Commit *decf8a6* introduces the issue, commit *738d1a2* solves it [23].

Config.	*NodeManager*				*DataNode*			
	decf8a6		738d1a2		decf8a6		738d1a2	
	EP	*Blob*	*EP*	*Blob*	*EP*	*Blob*	*EP*	*Blob*
1-3	0	0	0	0	0	0	0	0
1-6	0	0	0	0	0	0	0	0
1-12	0	0	0	0	0	0	0	0
10-3	1	0	0	0	1	1	0	0
10-6	1	0	1	0	1	1	1	0
10-12	1	0	0	0	1	1	1	0
100-3	1	0	0	0	1	1	1	0
100-6	1	0	0	0	1	1	1	0
100-12	0	0	0	0	1	1	1	0
500-3	1	0	1	0	1	1	1	1
500-6	1	0	0	0	1	1	1	0
500-12	1	0	0	0	1	1	1	0
1000-3	1	0	1	0	1	0	1	0
1000-6	1	0	1	0	1	0	1	0
1000-12	1	0	0	0	1	0	1	0

instances in the *DataNode* rarely change (i.e., only for 1-3 and 10-3), whereas Blob instances are reduced in eight configurations (e.g., 10-3 and 10-6). Similarly to the previous issue, the *DataNode* component shows more variations for Blob instances than for EP instances. Summing up all the analyzed configurations there are 13 EP instances and 9 Blob instances for the code commit inducing the issue (*528b809*), against 11 EP instances and 1 Blob instance for the code commit fixing the issue (*ff8378e*).

Table 11(c) presents the results for the HDFS-12754 issue, and also here the number of antipattern instances is equal or lower when considering the commit fixing the issue, i.e., *738d1a2*. Differently from previous cases, if EP instances are not detected then Blob instances do not arise in the *NodeManager* component. Overall, 11 and 4 EP instances across all configurations are observed for *decf8a6* and *738d1a2*, respectively. For the *DataNode* component, we get 1 Blob instance and 11 EP instances in the commit solving the issue, whereas in the commit inducing the issue we found 12 EP instances and 9 Blob instances. Summarizing, JPAD detects the variation across different commits when calculating the total number of detected antipattern instances. Overall, we can notice that detected instances significantly decreases when comparing code commits which induce and fix issues. Besides, software components impact on such influence, the *NodeManager* shows, on average, less EP and Blob instances when issues are fixed. Instead, for the *DataNode*, only Blob instances are observed to reduce after fixing the issue, the number of EP instances slightly varies. This is due to the nature of the analyzed issues, in fact both YARN-7102 and HDFS-12754 are indicated in [23] as complicated performance issues (like deadlock), and EP captures that there is a large number of blocked threads (i.e., a symptom of a deadlock) leading to long execution time (see Table 1).

Table 12 reports the performance antipattern instances that are found in Cassandra across two different code commits. JPAD takes 1 second on average to analyze these configurations, and it detects CTH, WCS, Blob, and EDA antipatterns, other antipatterns are not reported since no instances are detected. Between the two commits we can notice that all detected antipatterns show some decrease in their numbers when considering the commit fixing the

TABLE 12: Number of detected antipatterns using JPAD in two commits of Cassandra. Commit *88d2ac4* introduces the CASSANDRA-13794 issue, commit *f93e6e3* solves it [23]. JPAD takes 1 second on average for the detection of antipatterns in these configurations.

Config.	88d2ac4				f93e6e3			
	CTH	WCS	Blob	EDA	CTH	WCS	Blob	EDA
1-3	0	0	0	0	0	0	0	0
1-6	0	0	0	0	0	0	0	0
1-12	0	0	0	0	0	0	0	0
10-3	1	0	2	0	1	0	2	0
10-6	1	0	2	0	1	0	2	0
10-12	1	0	2	0	1	0	2	0
100-3	2	1	3	2	2	1	3	1
100-6	2	1	4	2	2	1	3	1
100-12	2	1	5	2	2	1	4	2
500-3	2	1	3	1	2	1	3	1
500-6	3	1	5	2	2	1	3	1
500-12	3	1	4	2	2	1	3	1
1000-3	3	2	4	2	2	1	3	1
1000-6	3	1	4	2	2	1	4	1
1000-12	3	1	4	2	2	1	4	2

TABLE 13: Detection capability of JPAD and state-of-the-art tools w.r.t. Hadoop and Cassandra real-life performance issues. Selected issues, systems commits, and detection results of PerfJIT and Perphecy are extracted from [23].

Issue fixing commit	*Issue inducing commit*	*PADprof* [53]	*PerfJIT* [23]	*Perphecy* [45]	***JPAD***
308d63f	e914220	NO	NO	NO	**YES**
	7af5d6b	NO	NO	NO	**YES**
ff8378e	528b809	NO	NO	YES	**YES**
738d1a2	decf8a6	NO	NO	YES	**YES**
f93e6e3	88d2ac4	NO	NO	YES	**YES**

issue, i.e., *f93e6e3*. Blob is the antipattern showing a larger number of instances, in fact commit *88d2ac4* shows 42 instances summing up all configurations, whereas commit *f93e6e3* includes 36 instances. About WCS, we can notice that there is one configuration only (i.e., 1000-3) showing a decrease of antipattern instances, no major variation is observed for this specific antipattern. Both CTH and EDA show a considerable variation; looking at the total number of detected instances across all analyzed configurations of *88d2ac4* and *f93e6e3* commits, we get 29 and 21 (17 and 11) CTH (EDA) instances, respectively. Hence, JPAD effectively detects a remarkable difference across the analyzed code commits (inducing and fixing real-life performance issues).

To answer RQ_2, Table 13 summarizes results of comparing JPAD with state-of-the-art approaches. The last four columns of this table indicate if the specified tool can detect the considered performance issue. The column named *PADprof* refers to the framework presented in [53] which we test providing problematic snapshots (i.e., commits inducing the issue) and comparison snapshots (i.e., commits fixing the issue). All the analyzed snapshots show that no antipatterns are detected. Results for PerfJIT [23] and Perphecy [45] are instead extracted from [23] when investigating the detection of real-life performance issues.

RQ_2: comparison with state-of-the-art

JPAD overcomes state-of-the-art approaches [23], [45], [53] in the detection of some real-life performance issues. The proposed detection rules effectively capture complex performance problems that are not recognized by available tools. This consolidates the adoption of our framework as an alternative approach to support software engineers in understanding performance issues in Java applications.

4.4.3 *Implication of applying antipattern-based refactorings*

To answer RQ_3, we refactor OpenMRS (i.e., the largest application among those considered in Table 4) to understand if the detection information provided by JPAD can support software engineers in solving performance issues. The selection of OpenMRS as target system for investigating the refactoring is also motivated by a recent paper [46] that analyzes the same system to locate performance regression root causes. Specifically, we focus on the `OwaFilter` method that JPAD detects as a Blob-Controller instance. Such a method is responsible for filtering the requests directed to protected endpoints, i.e., access is granted for authenticated requests only. Due to the modular nature of the application, requests come from different modules and the filter must check the URL of all incoming requests before granting access to authenticated users and forward their requests.

Listing 1 reports a code excerpt of the `OwaFilter`. We can notice that there are several requests to be managed, for example:

- `getRequestURL()` (see line 9),
- `getServletPath()` (see line 12),
- `getAdmnistrationService()` (see line 14)

to mention a few. There is indeed a match with the textual description of the antipattern (see Table 1) indicating that the Blob-controller occurs in case of a single class performing all the work of an application.

As specified in the literature [26], when solving a Blob-Controller antipattern, the refactoring consists of moving computation from the affected instance to a different one. We delegate the verification of URLs to a centralized authentication system that forwards requests to the correct endpoint after the authentication process is completed. This way, the `OwaFilter` method must only check that users are authenticated. After refactoring OpenMRS, we evaluate its performance under all loads and compare the obtained results with those observed from the original OpenMRS version. To quantify performance improvements, we consider these metrics of interest: (i) the number of detected software performance antipatterns, (ii) the CPU utilization, and (iii) the system response time. It is worth remarking that our focus is on showing empirical evidence on the benefit of solving antipatterns, i.e., possible performance improvements that can be derived by detecting and removing antipatterns, and this is why we do not investigate further refactoring types or solutions.

Table 14 reports the antipattern instances detected for the refactored system. Overall, compared to the original system, the number of instances mostly decreases, see the

```
1  public void doFilter(final ServletRequest req, final ServletResponse res, final FilterChain chain) throws IOException,
       ServletException {
2          final HttpServletRequest request = (HttpServletRequest)req;
3          String owaBasePath = Context.getAdministrationService().getGlobalProperty("owa.appBaseUrl", "/owa");
4          if (StringUtils.isBlank((CharSequence)owaBasePath)) {
5              owaBasePath = "/owa";
6          }
7          String requestURL = null;
8          if (isFullBasePath(owaBasePath)) {
9              requestURL = request.getRequestURL().toString();
10         }
11         else {
12             requestURL = request.getServletPath();
13         }
14         final String loginUrl = Context.getAdministrationService().getGlobalProperty("login.url", "login.htm");
15
16         ...
17         }
18     }
```

Listing 1: Code excerpt of the `OwaFilter` method detected by JPAD as Blob-Controller antipattern instance.

TABLE 14: Number of detected instances in OpenMRS after applying antipattern-based refactoring. ToB and EST are omitted since no instances have been detected.

Config.	*CTH*		*EP*		*WCS*		*Blob*		*EDA*	
	Refact.	Diff.	Refact.	Diff.	Refact.	Diff.	Refact.	Diff.	Refact.	Diff.
25-3	0	-2	2	-1	0	-2	3	+1	1	-1
25-6	0	0	1	+1	0	0	1	0	0	-1
25-12	0	0	1	+1	0	-1	1	0	1	0
50-3	0	0	1	0	0	0	1	0	0	0
50-6	0	0	1	0	0	0	1	0	0	0
50-12	0	0	1	0	0	0	1	0	0	0
75-3	0	-1	1	-1	0	-1	1	-2	0	-1
75-6	0	-1	2	0	0	-1	3	0	1	0
75-12	0	0	1	0	0	0	1	0	0	0
100-3	1	+1	2	0	1	+1	2	-1	1	0
100-6	0	0	1	0	0	0	1	0	0	0
100-12	0	0	1	0	0	0	1	0	0	0

TABLE 15: Performance variation (%) obtained by applying the antipattern-based refactoring to OpenMRS.

Config.	Utilization			Response Time		
	Original [%]	*Refactored [%]*	*Variation [%]*	*Original [s]*	*Refactored [s]*	*Variation [%]*
25-3	27.89	15.00	**46.22**	3.913	4.874	-24.56
25-6	23.03	12.38	**46.24**	8.706	8.075	7.25
25-12	20.38	10.75	**47.25**	18.295	17.299	5.45
50-3	55.37	53.54	3.31	5.919	4.111	**30.55**
50-6	57.22	52.77	7.78	13.743	9.042	**34.20**
50-12	56.23	51.26	8.84	30.180	21.446	**28.94**
75-3	63.29	62.36	1.47	5.545	3.138	**43.42**
75-6	66.84	60.30	9.78	14.839	7.350	**50.47**
75-12	66.96	56.80	**15.17**	32.977	17.902	**45.71**
100-3	55.92	47.21	**15.58**	3.262	3.038	6.88
100-6	57.37	49.23	**14.19**	11.546	6.747	**41.57**
100-12	57.49	51.20	**10.94**	28.883	24.292	**15.89**

Diff. column where negative numbers indicates that the number of instances is decreased after the refactoring. For example, with 75 clients and 3 minutes of load tests running, JPAD detects 1 CTH, 2 EP, 1 WCS, 3 Blob, and 1 EDA instances in the OpenMRS original system. When applying the antipattern-based refactoring, we remove 1 instance of CTH, EP, WCS, and EDA, and 2 Blob instances. A similar improvement (see -2 entries in Table 14) is observed for CTH and WCS when there are 25 clients and the load test runs for 3 minutes. However, it is worth noting that some configurations (i.e., 25-6, 25-12, and 100-3) show more antipattern instances in the refactored system. For instance, for the configuration 25-3, the number of Blob instances increases in the refactored case. After further investigation, we find that interestingly this is due to the introduction of a *Blob-DataContainer* instance while solving the *Blob-Controller* antipattern. However, this is the only case for which Blob instances increase. Generally, the number of Blob instances is constant and decreases with 75 or 100 clients and 3 minutes of load tests running. The increment of antipattern instances (see +1 entries in Table 14) is observed for EP in two configurations and for CTH, WCS, and Blob in only one configuration. The number of EDA instances does not increase in any of the considered configurations.

The impact of the antipattern-based refactoring is also observed on performance indices of interest, i.e., CPU utilization and system response time of the OpenMRS Java application, results are shown in Table 15. The two perfor-

mance indices are reported for both the original and the refactored OpenMRS system. The *Variation* column shows the observed performance change and is computed as: *Variation* = [(*Original* - *Refactored*) / *Original*] · 100. When *Variation* > 0, the considered index is smaller for the refactored system than for the original one, meaning that the system performance has improved. Table 15 highlights with bold entries all those for which *Variation* is larger than 10%. The relevance of antipattern-based refactoring is shown by the general enhancement of the OpenMRS application performance (up to 50.47%, observed for the system response time). The configuration with 25 clients and 3 minutes of load tests running is the only exception to this observation. In this case, the response time of the refactored system is 25% longer than the one of the original system even if the CPU utilization is lower for the refactored system. This may be due to performance issues that are generated in different (and previously not critical) parts of the system. We already observed that the 25-3 configuration introduces a *Blob-DataContainer* instance, when solving the *Blob-Controller* antipattern. This might be the reason for the longer response time. As future work, we plan to further investigate the solution of antipatterns and possible implications in generating new instances.

RQ$_3$: antipattern-based refactoring

Antipattern-based refactoring does not guarantee in advance neither a reduction of the total number of detected instances nor an improvement in the system performance. However, our experimentation shows that usually less antipattern instances are detected, and most performance indicators of interest improve. By refactoring OpenMRS, we find empirical evidence on the benefit of solving one antipattern: 12 less antipattern instances are detected; on average (across all the analyzed configurations), the CPU utilization is 18.90% lower and the system response time is 23.81% shorter. Maximum improvements for CPU utilization and system response time are 47.25% and 50.47%, respectively.

5 Threats to validity

Besides inheriting all the limitations related to the performance evaluation of Java-based applications [2], our approach exhibits the following main (construct, conclusion, internal, and external) threats to validity [75].

Construct threats. This type of threat is observed when metrics deviate from the focus of the investigation. To smooth it, we provide a quantitative evaluation of the approach motivated by the research questions. We show that (i) detection rules work on real-world case studies, (ii) real-life performance issues are captured, and (iii) solving one antipattern instance improves the system performance.

Conclusion threats. A threat of this type is related to the reliability of collected measures. To smooth these threats, we run all experiments on the same machine. Moreover, the profiling of the Java applications under analysis is delegated to the YourKit Java Profiler, a well-assessed and widely-used tool for this scope [76].

Internal threats. We thoroughly test JPAD to spot errors in its implementation. For each experiment (whose setup can be easily changed by users), when an antipattern instance is detected, we verify if thresholds are violated. We recall that JPAD is publicly available [28] for inspection and to replicate experiments of this paper.

External threats. We are aware that findings from our experiments may not transfer to different Java applications. To increase the external validity, we select software systems from different domains whose class number ranges from 68 to more than 1k. We also inspect code commits related to four real-life performance issues that are considered rather complex to be predicted by state-of-the-art approaches [23]. This way, we evaluate our approach against diverse applications so that our results may generalize to other case studies.

6 Discussion

In this section, we discuss limitations of our approach that we consider as open issues paving the way for future research investigations.

Soundness and completeness. Our approach currently detects seven software performance antipatterns experimented on five Java applications belonging to different domains, and nine specific commits of two further subject systems used in prior research to extract performance data [77]. Even if we demonstrate that our approach is able to recognize some performance issues that are not detected by other approaches in the literature (see Table 13), soundness and completeness are not guaranteed. To partially address this issue, a preliminary investigation is conducted experimenting (i) a set of four commits known to fix performance issues (see Tables 11–12) and (ii) an antipattern-based refactoring along with the consequent performance variation on utilization and response time indices (see Table 15). As future work we plan to strengthen this investigation involving practitioners in the evaluation of JPAD.

Antipattern specification. Detection algorithms reflect our interpretation of the textual description of software performance antipatterns provided in [26]. We are aware that further interpretations can be provided by different stakeholders (e.g., practitioners), and we leave as part of our future work the possibility of customizing detection rules and to provide a flexible framework that reflects multiple interpretations. More in general, we plan to introduce a domain-specific language for software performance antipatterns as support for users that may define their own detection rules. This way, we aim to strengthen the specification of antipatterns and to collect the experience of different stakeholders, possibly even discovering new antipatterns.

Profiling overhead. The performance monitoring of Java applications is known to generate overhead [54], a comparison of different profiling tools and their overhead is presented in [78], [79]. In this paper, we use YourKit since both academia [53], [57] and industrial partners, such as Apple and Google, employ it as support for evaluating the performance of industrial and real-world applications. To partially cope with the overhead introduced by YourKit, all our detection algorithms include at least one threshold derived from offsets and average values. Offsets are independent of the absolute value of considered metrics,

and they allow specifying thresholds based on values that already include the profiling overhead. This way, JPAD compares performance metrics and thresholds that are both affected by the profiling overhead. We leave as future work the investigation on the usage of other monitoring tools to compare (and possibly smooth) the profiling overhead.

Antipattern thresholds. As argued in [80], thresholds must be set in software performance antipatterns to express performance requirements (when available), or to establish boundaries which represent the perception of different system stakeholders. In fact, users can differently judge the importance of performance requirements, e.g., the hardware utilization may be associated to monetary costs and more relevant for system administrators, whereas the execution time of a service is taken into account mainly by software developers. Therefore, JPAD provides the possibility to specify such thresholds, and this task is intentionally transferred to users that can decide which numerical values are more suitable for their purposes. We leave as future work the possibility of exploring further strategies, possibly synthesizing the need of different stakeholders.

Software performance testing. Test cases are often very important for an effective dynamic analysis [81]. Our approach delegates the test design to software engineers that may focus on general requirements and miss the relevant ones (from a performance-based perspective). Our experimentation highlights the importance of designing test suites (see Table 11), and demonstrates that such a selection can be guided by a preliminary analysis of the CPU utilization of software components. However, as future work we plan to investigate if approaches in the literature dealing with an efficient design of performance tests [82], [83] can be integrated in JPAD.

Antipattern-based refactorings. This is a very complex activity, and it is not guaranteed that the number of detected antipatterns decreases or the system performance improves. Our experimentation shows that solving one antipattern may generate other antipattern instances; more in general, antipattern instances can increase and the system response time can worsen (see Tables 14–15). Besides, the complexity is exacerbated by the possibly large number of detected antipattern instances, each matching with multiple code refactorings, and it is very difficult to understand which changes should be prioritized. In our previous work [33] we proposed a ranking methodology for the evaluation of architectural alternatives. We leave as part of future work to experiment ranking strategies on code refactorings thus to better investigate this aspect. Large systems may show the additional difficulty of being more sensitive to the impact of code changes, probably due to dependencies (among components) that need to be propagated when implementing refactorings. Antipattern-based solutions might be enriched with information about their effect (e.g., the involvement of dependent components) to identify which subsystems are involved in the refactoring process and may trigger new antipatterns. To automatically fix the detected antipatterns, it is necessary that code refactorings undergo a verification process that guarantees their functional correctness.

Guidelines for developers. When adopting JPAD in practice, we encourage developers to consider two different dimensions on the results they get as output. First, one can determine that a hotspot method shows performance issues if it is reported as a violation of some antipatterns. Second, when considering the different system configurations, the presence of the very same hotspot method (across many system configurations) contributes to the decision that such a method is indeed relevant for the performance issues under analysis. Both these two cases may indicate that such a hotspot method includes several design flaws and it indeed contributes to poor system performance.

7 Conclusion and Future Work

In this paper we present JPAD, a tool-based approach to automatically detect software performance antipatterns in Java applications. The experimentation is performed on real-world Java applications from different domains, and JPAD captures four real-life performance issues that are not predicted by state-of-art approaches [23], [45], [53]. Results show the efficiency and accuracy of the proposed approach. The antipattern detection is executed on 300 configurations and we exploit such extensive experimentation to build a ground truth, thus to quantify JPAD accuracy. Overall, the accuracy is larger than 95% and the F1 score, derived from precision and recall metrics, is larger than 85% in the considered cases, leading to assess accurate detection rules. About efficiency, system configurations are analyzed, on average, in less than a minute, some configurations require more time and JPAD always takes less than 5 minutes to complete the detection of antipatterns. Besides, the number of detected antipattern instances substantially vary when experimenting software code commits known to induce and fix real-life performance issues. Antipattern-based refactoring turns out to be beneficial, the system performance improves up to 47% and 50% when measuring two specific metrics of interest, i.e., CPU utilization and system response time, respectively. JPAD points out system characteristics (e.g., number of times a method is invoked) that lead to performance issues, and its report includes quantitative information. This way, we aim to support software engineers in the task of taking decisions on which methods require more attention than others from a performance-based perspective.

Several research directions have been identified for future research. First, we want to extend the specification of antipatterns and make them *flexible*, i.e., users can add and modify detection rules to provide their own interpretation of antipatterns, possibly by introducing a domain-specific language. Second, we plan to extend the set of analyzed systems, possibly including case studies from the industrial domain to further assess both efficiency and accuracy. Third, we plan to extend JPAD to point out possible directions for antipattern-based refactorings, but the actual implementation of code fixings is delegated to software engineers who can assure the preservation of the business logic of applications. Moreover, the solution process is complex due to the number of detected antipatterns that may be large, as demonstrated in this paper, and it is difficult to select which antipattern to solve first. Hence, we want to investigate concurrent (or prioritized) resolution of multiple antipatterns, this may lead to inconsistencies due to conflicting solutions for which ad-hoc methodologies need to be defined.

REFERENCES

[1] C. Hunt and B. John, *Java performance*. Prentice Hall Press, 2011.

[2] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous Java performance evaluation," *ACM SIGPLAN Notices*, vol. 42, no. 10, pp. 57–76, 2007.

[3] M. Harkema, D. Quartel, B. Gijsen, and R. D. van der Mei, "Performance monitoring of Java applications," in *Proceedings of the International Workshop on Software and Performance (WOSP)*, 2002, pp. 114–127.

[4] R. Rabiser, S. Guinea, M. Vierhauser, L. Baresi, and P. Grünbacher, "A comparison framework for runtime monitoring approaches," *Journal of Systems and Software*, vol. 125, pp. 309–321, 2017.

[5] R. Calinescu, C. Ghezzi, M. Z. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," *Communications of the ACM*, vol. 55, no. 9, pp. 69–77, 2012.

[6] H. Zhang and S. Kim, "Monitoring software quality evolution for defects," *IEEE Software*, vol. 27, no. 4, pp. 58–64, 2010.

[7] R. Capilla, M. A. Babar, and O. Pastor, "Quality requirements engineering for systems and software architecting: methods, approaches, and tools," *Requirements Engineering*, vol. 17, no. 4, pp. 255–258, 2012.

[8] J. L. De La Vara, K. Wnuk, R. Berntsson-Svensson, J. Sánchez, and B. Regnell, "An Empirical Study on the Importance of Quality Requirements in Industry," in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2011, pp. 438–443.

[9] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software Architecture Optimization Methods: A Systematic Literature Review," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 658–683, 2013.

[10] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Performance Evaluation*, vol. 67, no. 8, pp. 634–658, 2010.

[11] F. Brosig, S. Kounev, and K. Krogmann, "Automated extraction of palladio component models from running enterprise java applications," in *Proceedings of the International Conference on Performance Evaluation Methodologies and Tools (ICST)*, 2009, pp. 1–10.

[12] S. Voneva, M. Mazkatli, J. Grohmann, and A. Koziolek, "Optimizing parametric dependencies for incremental performance model extraction," in *Proceedings of the European Conference on Software Architecture (ECSA)*, 2020, pp. 228–240.

[13] C. Heger, A. V. Hoorn, D. Okanovic, S. Siegl, and A. Wert, "Expert-Guided Automatic Diagnosis of Performance Problems in Enterprise Applications," in *Proceedings of the European Dependable Computing Conference (EDCC)*, 2016, pp. 185–188.

[14] A. Van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *Proceedings of the International Conference on Performance Engineering (ICPE)*, 2012, pp. 247–248.

[15] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, "A survey of devops concepts and challenges," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–35, 2019.

[16] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[17] M. Hüttermann, *DevOps for developers*. Apress, 2012.

[18] J. Waller, N. C. Ehmke, and W. Hasselbring, "Including Performance Benchmarks into Continuous Integration to Enable DevOps," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 2, pp. 1–4, 2015.

[19] H. Schulz, D. Okanovic, A. van Hoorn, and P. Tuma, "Context-tailored workload model generation for continuous representative load testing," in *Proceedings of the International Conference on Performance Engineering (ICPE)*, 2021, pp. 21–32.

[20] 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 *Proceedings of the International Conference on Software Engineering (ICSE)*, 2014, pp. 1001–1012.

[21] A. Nistor and et al., "CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015, pp. 902–912.

[22] Z. Li, T. P. Chen, J. Yang, and W. Shang, "Dlfinder: characterizing and detecting duplicate logging code smells," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2019, pp. 152–163.

[23] J. Chen, W. Shang, and E. Shihab, "PerfJIT: Test-Level Just-in-Time Prediction for Performance Regression Introducing Commits," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1529–1544, 2022.

[24] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2018, pp. 1–23.

[25] L. Song and S. Lu, "Statistical debugging for real-world performance problems," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 561–578, 2014.

[26] C. U. Smith and L. G. Williams, "More new software performance antipatterns: Even more ways to shoot yourself in the foot," in *Proceedings of the International Conference on Computer Measurement Group (CMG)*, 2003, pp. 717–725.

[27] C. U. Smith, "Software performance antipatterns in cyber-physical systems," in *Proceedings of the International Conference on Performance Engineering (ICPE)*, 2020, pp. 173–180.

[28] C. Trubiani, R. Pinciroli, A. Biaggi, and F. Arcelli Fontana, "Replication Package: Automated Detection of Software Performance Antipatterns in Java-based Applications," 2022. [Online]. Available: https://doi.org/10.5281/zenodo.5878953

[29] A. Shokri, J. C. S. Santos, and M. Mirakhorli, "Arcode: Facilitating the use of application frameworks to implement tactics and patterns," in *Proceedings of the International Conference on Software Architecture (ICSA)*, 2021, pp. 138–149.

[30] D. Feitosa, A. Ampatzoglou, P. Avgeriou, A. Chatzigeorgiou, and E. Y. Nakagawa, "What can violations of good practices tell about the relationship between GoF patterns and run-time quality attributes?" *Information and Software Technology*, vol. 105, pp. 1–16, 2019.

[31] G. Hecht, B. Jose-Scheidt, C. D. Figueiredo, N. Moha, and F. Khomh, "An Empirical Study of the Impact of Cloud Patterns on Quality of Service (QoS)," in *Proceedings of the International Conference on Cloud Computing Technology and Science (CloudCom)*, 2014, pp. 278–283.

[32] M. Galster and P. Avgeriou, "Qualitative Analysis of the Impact of SOA Patterns on Quality Attributes," in *Proceedings of the International Conference on Quality Software (QSIC)*, 2012, pp. 167–170.

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

[34] R. Calinescu, V. Cortellessa, I. Stefanakos, and C. Trubiani, "Analysis and Refactoring of Software Systems Using Performance Antipattern Profiles," in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2020, pp. 357–377.

[35] A. Wert, J. Happe, and L. Happe, "Supporting swift reaction: automatically uncovering performance problems by systematic experiments," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013, pp. 552–561.

[36] B. Walter, F. A. Fontana, and V. Ferme, "Code smells and their collocations: A large-scale experiment on open-source systems," *Journal of Systems and Software*, vol. 144, pp. 1–21, 2018.

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

[38] F. A. Fontana, V. Ferme, A. Marino, B. Walter, and P. Martenka, "Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains," in *Proceedings of the International Conference on Software Maintenance (ICSME)*, 2013, pp. 260–269.

[39] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013, pp. 682–691.

[40] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of bad smell detection and resolution: A new way to save effort," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 220–235, 2011.

[41] D. Hovemeyer and W. Pugh, "Finding more null pointer bugs, but not too many," in *Proceedings of the International Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2007, pp. 9–14.

[42] Q. Luo, D. Poshyvanyk, and M. Grechanik, "Mining performance regression inducing code changes in evolving software," in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 25–36.

[43] J. Chen and W. Shang, "An exploratory study of performance regression introducing code changes," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 341–352.

[44] D. Alshoaibi, K. Hannigan, H. Gupta, and M. W. Mkaouer, "Price: Detection of performance regression introducing code changes using static and dynamic metrics," in *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE)*, 2019, pp. 75–88.

[45] A. B. De Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Perphecy: performance regression test selection made simple but effective," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 103–113.

[46] 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 Transactions on Software Engineering*, 2021, (Early Access).

[47] B. A. Tate, *Bitter Java*. Manning Publications Co., 2002.

[48] B. Dudney, S. Asbury, J. K. Krozak, and K. Wittkopf, *J2EE antipatterns*. John Wiley & Sons, 2003.

[49] B. Tate, M. Clark, and P. Linskey, *Bitter EJB*. Manning Publications Co., 2003.

[50] H. Hallal, E. Alikacem, W. Tunney, S. Boroday, and A. Petrenko, "Antipattern-based detection of deficiencies in Java multithreaded software," in *Proceedings of the International Conference on Quality Software (QSIC)*, 2004, pp. 258–267.

[51] P. Leitner and C. Bezemer, "An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects," in *Proceedings of the International Conference on Performance Engineering (ICPE)*, 2017, pp. 373–384.

[52] T. Parsons and J. Murphy, "Detecting performance Antipatterns in Component Based Enterprise Systems," *Journal of Object Technology*, vol. 7, no. 3, pp. 55–90, 2008.

[53] C. Trubiani, A. Bran, A. van Hoorn, A. Avritzer, and H. Knoche, "Exploiting load testing and profiling for performance antipattern detection," *Information and Software Technology*, vol. 95, pp. 329–345, 2018.

[54] V. Horký, J. Kotrc, P. Libic, and P. Tuma, "Analysis of Overhead in Dynamic Java Performance Monitoring," in *Proceedings of the International Conference on Performance Engineering (ICPE)*, 2016, pp. 275–286.

[55] YourKit GmbH, "YourKit Java Profiler Features," https://www.yourkit.com/features/, [Online, accessed 2024-04-23].

[56] ——, "YourKit Customers," https://www.yourkit.com/customers/, [Online, accessed 2024-04-23].

[57] Y. Zhao, L. Xiao, X. Wang, Z. Chen, B. Chen, and Y. Liu, "Butterfly space: An architectural approach for investigating performance issues," in *Proceedings of the International Conference on Software Architecture (ICSA)*, 2020, pp. 202–213.

[58] YourKit GmbH, "Profiling overhead: how to reduce or avoid," https://www.yourkit.com/docs/java/help/overhead.jsp, [Online, accessed 2024-04-23].

[59] G. Canfora, F. Martinelli, F. Mercaldo, V. Nardone, A. Santone, and C. A. Visaggio, "Leila: formal tool for identifying mobile malicious behaviour," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1230–1252, 2018.

[60] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 77–88.

[61] J. Thome, L. K. Shar, D. Bianculli, and L. Briand, "An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving," *IEEE Transactions on Software Engineering*, vol. 46, no. 2, pp. 163–195, 2018.

[62] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2021.

[63] "CloudStore synchronizes your data," https://github.com/cloudstore/cloudstore, [Online, accessed 2024-04-23].

[64] "TeaStore: a micro-service reference and test application," https://github.com/DescartesResearch/TeaStore, [Online, accessed 2024-04-23].

[65] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research," in *Proceedings of the International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2018, pp. 223–236.

[66] "WebGoat 8: A deliberately insecure Web Application," https://github.com/WebGoat/WebGoat, [Online, accessed 2024-04-23].

[67] "Train Ticket: A Benchmark Microservice System," https://github.com/FudanSELab/train-ticket, [Online, accessed 2024-04-23].

[68] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 683–694.

[69] "OpenMRS: Medical Record System," https://github.com/openmrs/openmrs-core, [Online, accessed 2024-04-23].

[70] The Apache Software Foundation, "Apache Hadoop," https://hadoop.apache.org/, [Online, accessed 2024-04-23].

[71] ——, "Apache Cassandra: Open Source NoSQL Database," https://cassandra.apache.org/_/index.html, [Online, accessed 2024-04-23].

[72] "TestDFSIO," https://github.com/apache/hadoop/blob/master/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-jobclient/src/test/java/org/apache/hadoop/fs/TestDFSIO.java, [Online, accessed 2024-04-23].

[73] The Apache Software Foundation, "Cassandra Stress," https://cassandra.apache.org/doc/4.0/cassandra/tools/cassandra_stress.html, [Online, accessed 2024-04-23].

[74] J. Heyman, C. Byström, J. Hamrén, and H. Heyman, "Locust: An open source load testing tool," https://locust.io/, [Online, accessed 2024-04-23].

[75] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[76] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Evaluating the accuracy of Java profilers," *ACM SIGPLAN Notices*, vol. 45, no. 6, pp. 187–197, 2010.

[77] M. D. Syer, W. Shang, Z. M. Jiang, and A. E. Hassan, "Continuous validation of performance test workloads," *Automated Software Engineering*, vol. 24, no. 1, pp. 189–231, 2017.

[78] P. Su, Q. Wang, M. Chabbi, and X. Liu, "Pinpointing performance inefficiencies in java," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 818–829.

[79] F. David, G. Thomas, J. Lawall, and G. Muller, "Continuously measuring critical section pressure with the free-lunch profiler," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 291–307, 2014.

[80] V. Cortellessa, A. Di Marco, and C. Trubiani, "An approach for modeling and detecting software performance antipatterns based on first-order logics," *Software and Systems Modeling*, vol. 13, no. 1, pp. 391–432, 2014.

[81] L. Mariani, F. Pastore, and M. Pezze, "Dynamic analysis for diagnosing integration faults," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 486–508, 2010.

[82] S. Mostafa, X. Wang, and T. Xie, "Perfranker: prioritization of performance regression tests for collection-intensive software," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, T. Bultan and K. Sen, Eds., 2017, pp. 23–34.

[83] D. G. Reichelt and S. Kühne, "Better early than never: Performance test acceleration by regression test selection," in *Proceedings of the International Conference on Performance Engineering (ICPE), Companion volume*, 2018, pp. 127–130.

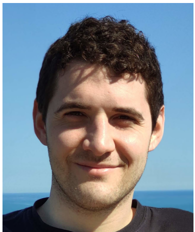
[84] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: machine learning inference serving on serverless platforms with adaptive batching," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–15.

[85] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu, "Yak: A high-performance big-data-friendly garbage collector," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 349–365.

Acknowledgments

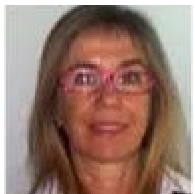
The authors would like to thank the Editor and the anonymous reviewers for their constructive comments and valuable feedback. We are also grateful to the Computing and Network Service for their support in our experiments on the U-LITE cluster at INFN, LNGS, L'Aquila, Italy. This work has been partially supported by the MUR PRIN project SEDUCE 2017TWRCNB.

Catia Trubiani is Associate Professor at the Gran Sasso Science Institute (GSSI), Italy. Previously she collaborated with the Karlsruhe Institute of Technology in Germany, and the Imperial College of London in UK. Her research interests include software performance modeling and analysis, software quality optimization, uncertainty propagation, model-based testing and software architectures. For more information, please visit https://cs.gssi.it/catia.trubiani



Riccardo Pinciroli received M.S. (2014) and Ph.D. (2018) degrees in computer engineering from Politecnico di Milano. He is currently a Postdoc Fellow in Computer Science at the Gran Sasso Science Institute. His research interests include stochastic modeling, performance evaluation, energy efficiency, and uncertainty propagation applied to cloud computing, data-centers, and cyber-physical systems.

Andrea Biaggi took his Master degree at the Evolution and Reverse Engineering Lab of University of Milano Bicocca, Italy, with the supervision of Prof. Francesca Arcelli Fontana. He had a fellowship from the same Lab and currently works as a software engineer in a company in Milano, Italy.



Francesca Arcelli Fontana is Full Professor at University of Milano Bicocca, Italy. Her research activity principally concerns software engineering, in particular software evolution and reverse engineering, architectural smell detection, program comprehension, software quality assessment and managing technical debt. She is the Head of the Software Evolution and Reverse Engineering Lab at University of Milano Bicocca.

APPENDIX A
ANTIPATTERN DETECTION ALGORITHMS

This appendix provides the detection algorithms and implementation details of the following seven performance antipatterns: 1) Circuitous Treasure Hunt (CTH), 2) Extensive Processing (EP), 3) Wrong Cache Strategy (WCS), 4) Blob, 5) Tower of Babel (ToB), 6) Empty Semi Trucks (EST), and 7) Excessive Dynamic Allocation (EDA).

In the following we argue on the matching between the textual description of antipatterns (see Table 1) and the proposed detection rules, thus to discuss the soundness of algorithms in checking sufficient information on the basis of our interpretation.

Algorithm 1 shows the detection algorithm for the Circuitous Treasure Hunt (CTH) antipattern. Three input parameters are provided to the algorithm, i.e., `countOffset`, `cpuTh`, and `option`. The algorithm first computes the `averageCpuUsage` and checks if it exceeds the threshold `cpuTh`. This condition matches with verifying if "performance will suffer" (see Table 1). In that case, depending on the `option` parameter, it computes the minimum, maximum, or average number of calls in each hotspot method (identified as suspicious in the previous condition). For each hotspot, the algorithm checks if the number of method calls is larger than the thresholds obtained by combining `countOffset` with the average number of calls previously computed. This rule is introduced to check if "an object must look in several places to find the information it needs" (see Table 1). If this condition is verified, then the hotspot is detected as a CTH instance. As output of the detection, we also report the average CPU usage and the number of calls produced by the identified hotspot method.

Algorithm 1 CTH detection.

```
 1: function DETECTCTH(countOffset, cpuTh, option)
 2:     avgCpuUsage ← GETAVGCPUUSAGE( )
 3:     if avgCpuUsage > cpuTh then
 4:         methodCountMap ← GETMETHODCOUNTMAP(option)
 5:         avgCount ← GETAVGMETHODCOUNT(methodCountMap)
 6:         avgTimeHotspot ← GETAVGTIME(hotspotMethods)
 7:         for all hotspotMethods do
 8:             if hotspot.time > avgTimeHotspots then
 9:                 hsMethodCount ← methodCountMap.get(hotspot)
10:                 countTh ← avgCount · (1 + countOffset)
11:                 if hsMethodCount > countTh then
12:                     REPORTCTH(avgCpuUsage, hsMethodCount)
13:                 end if
14:             end if
15:         end for
16:     end if
17: end function
```

Algorithm 2 shows the detection algorithm for the Extensive Processing (EP) antipattern. The execution time offset (i.e., `execTimeOffset`) is the only input parameter of the algorithm and it is a percentage value used with the average execution time of all methods (i.e., `avgExecTime`) to define the threshold `execTimeTh`. The algorithm first checks whether the blocked threads are more than a half of the total threads. This rule is introduced to check if "extensive processing monopolizes the processors" (see Table 1). If this condition is verified, it computes `avgExecTime` as the average method execution time normalized over the total execution time. The algorithm checks whether the execution time of hotspots is longer than `execTimeTh` and reports the EP antipattern when this condition holds. The condition on execution time matches with verifying whether "extensive processing impedes the overall response time" (see Table 1).

Algorithm 2 EP detection.

```
 1: function DETECTEP(execTimeOffset)
 2:     allThreadsCount ← COUNTTHREADS( )
 3:     blockedThreadsCount ← COUNTBLOCKEDTHREADS( )
 4:     if blockedThreadsCount > allThreadsCount/2 then
 5:         totExecTime ← GETTOTEXECTIME( )
 6:         avgExecTime ← (GETAVGEXECTIME( )/totExecTime) · 100
 7:         for all hotspotMethods do
 8:             hsExecTime ← (GETHSEXECTIME( )/totExecTime) · 100
 9:             execTimeTh ← avgExecTime + execTimeOffset
10:             if hsExecTime > execTimeTh then
11:                 REPORTEP(hsExecTime)
12:             end if
13:         end for
14:     end if
15: end function
```

Algorithm 3 reports the detection rules for the Wrong Cache Strategy (WCS) antipattern. It takes `memUsageOffset` as an input parameter, i.e., a percentage value used with the average memory usage of all methods to derive the threshold `memUsageTh`. This algorithm checks whether the memory usage of each hotspot exceeds both the CPU usage of the same method and the threshold `memUsageTh`. These two conditions verify when "caching too many objects (or objects that are rarely used) quickly changes the advantage of caching into a disadvantage due to higher memory usage" (see Table 1). If both conditions are verified, the hotspot is detected as a WCS instance.

Algorithm 3 WCS detection.

```
 1: function DETECTWCS(memUsageOffset)
 2:     avgMemUsage ← GETAVGMEMUSAGE(hotspotMethods)
 3:     memUsageTh ← avgMemUsage · (1 + memUsageOffset)
 4:     for all hotspotMethods do
 5:         hsMemUsage ← GETHSMEMUSAGE( )
 6:         hsCpuUsage ← GETHSCPUUSAGE( )
 7:         if hsMemUsage > MAX(hsCpuUsage, memUsageTh) then
 8:             REPORTWCS(hsMemUsage)
 9:         end if
10:     end for
11: end function
```

Algorithm 4 shows the detection rules for the Blob antipattern. Three input parameters are required: `msgOffset` (i.e., a percentage value used to define the caller and callee thresholds, namely `callersTh` and `calleesTh`), `cpuTh`, and `memTh` (i.e., two thresholds on the CPU and memory usage). First, the algorithm finds the resource (CPU or memory) with the largest average utilization and checks if such a usage exceeds the corresponding threshold (i.e., `cpuTh` for CPU usage and `memTh` for the memory). This rule is introduced to check if there is a situation that "degrades the performance" (see Table 1). If this condition holds, the algorithm computes `callersTh` and `calleesTh` using the defined offset (i.e., `msgOffset`) and the average number of callers (i.e., the number of methods that are invoked by each method) and callees (i.e., the number of times that each method is called). A hotspot method is reported as an instance of the Blob antipattern (more specifically as *Blob-Controller*) if its number of callees exceeds `calleesTh`. If

Algorithm 4 Blob detection.

```
1: function DETECTBLOB(msgOffset, cpuTh, memTh)
2:     avgCpuUsage ← GETAVGCPUUSAGE( )
3:     avgMemUsage ← GETAVGMEMUSAGE( )
4:     checkHwUtil ← false
5:     if avgCpuUsage ≥ avgMemUsage then
6:         checkHwUtil ← avgCpuUsage > cpuTh
7:     else
8:         checkHwUtil ← avgMemUsage > memTh
9:     end if
10:    if checkHwUtil then
11:        callersMap ← GETCALLERSMAP( )
12:        calleesMap ← GETCALLEESMAP( )
13:        avgCallers ← GETAVG(callersMap)
14:        avgCallees ← GETAVG(calleesMap)
15:        calleesTh ← avgCallees · (1 + msgOffset)
16:        callersTh ← avgCallers · (1 + msgOffset)
17:        for all hotspotMethods do
18:            hsCallers ← callersMap.get(hotspot)
19:            hsCallees ← calleesMap.get(hotspot)
20:            if hsCallees > calleesTh then
21:                REPORTBLOB("Controller")
22:            end if
23:            if hsCallers > callersTh then
24:                REPORTBLOB("Data Container")
25:            end if
26:        end for
27:    end if
28: end function
```

a hotspot method shows a number of callers that exceeds `callersTh`, then it is reported as *Blob-DataContainer* antipattern. This distinction is due to the verification of the following condition: "a single class either (i) performs all the work of an application or (ii) holds all the application data. Either manifestation results in excessive message traffic" (see Table 1).

Algorithm 5 ToB detection.

```
1: function DETECTTOB(execTimeOffset)
2:     methodCallsMap ← GETMETHODCALLSMAP( )
3:     avgCalls ← GETAVG( methodCallsMap )
4:     totExecTime ← GETTOTEXECTIME( )
5:     avgExecTime ← (GETAVGEXECTIME( )/totExecTime) · 100
6:     execTimeTh ← avgExecTime + execTimeOffset
7:     for all hotspotMethods do
8:         if  hotspot contains "parse"
9:           or hotspot contains "convert"
10:          or hotspot contains "translate"
11:        then
12:            hsCalls ← methodCallsMap.get(hotspot)
13:            if hsCalls > avgCalls then
14:                hsExecTime ← (GETHSEXECTIME( )/totExecTime)·100
15:                if hsExecTime > execTimeTh then
16:                    REPORTTOB( )
17:                end if
18:            end if
19:        end if
20:    end for
21: end function
```

Algorithm 5 shows the detection rules for the Tower of Babel (ToB) antipattern. It takes as input `execTimeOffset` (already defined for EP detection, see Algorithm 2). This algorithm iterates over each hotspot and checks whether (i) the method name contains one of the keywords reported in the definition of the antipattern (i.e., *parse*, *convert*, *translate*, as defined in [26]), (ii) if the hotspot method is invoked more than other methods, and (iii) if the hotspot execution time exceeds `execTimeTh`. These rules are introduced to check if "processes excessively convert, parse, and translate internal data into a common exchange format" (see Table 1). If all these conditions are verified, the hotspot method is reported as an instance of the ToB antipattern.

Algorithm 6 reports the detection rules for the Empty Semi Trucks (EST) antipattern. One input parameter is required, i.e., `msgOffset`. It is a percentage value used to define the callee threshold, namely `calleesTh`. To compute this threshold, we compute the average number of times that hotspots invoke their callees. For all callees, we count how many times each callee has been invoked, and we compare such count with the threshold value. This rule is aimed to check when "an excessive number of requests is required to perform a task" (see Table 1). If it is larger than the threshold, we check the coefficient of variation of the callee's service time. This condition is introduced to verify if "it may be due to inefficient use of available bandwidth, an inefficient interface, or both" (see Table 1). In case no large variation (i.e., lower than 0.1 since it can be approximated as a deterministic distribution [84]) is detected, then the callee is reported as an instance of the EST antipattern.

Algorithm 6 EST detection.

```
1: function DETECTEST(msgOffset)
2:     calleesMap ← GETCALLEESMAP( )
3:     for all hotspotMethods do
4:         hsCallees ← calleesMap.get(hotspot)
5:         avgCallees ← GETAVG(hsCallees)
6:         calleesTh ← avgCallees · (1 + msgOffset)
7:         for all hsCallees do
8:             numCalls ← COUNTCALLS(callee)
9:             if numCalls > calleesTh then
10:                coeffVariationTime ← GETCVTIME(callee)
11:                if coeffVariationTime < 0.1 then
12:                    REPORTEST(callee)
13:                end if
14:            end if
15:        end for
16:    end for
17: end function
```

Algorithm 7 EDA detection.

```
1: function DETECTEDA(memTh, gcObjectsOffset)
2:     avgMemUsage ← GETAVGMEMUSAGE( )
3:     if avgMemUsage > memTh then
4:         avgGcObjs ← GETGCOBJS(hotspotMethods)
5:         numObjsTh ← avgGcObjs · (1 + gcObjectsOffset)
6:         for all hotspotMethods do
7:             hsGcObjs ← GETHSGCOBJS( )
8:             if hsGcObjs > numObjsTh then
9:                 REPORTEDA(hsGcObjs)
10:            end if
11:        end for
12:    end if
13: end function
```

Algorithm 7 describes the detection rules of the Excessive Dynamic Allocation (EDA) antipattern. It takes as input **two** parameters, i.e., `memTh` and `gcObjectsOffset`, that refer to the thresholds on the memory usage and on the number of objects in the garbage collector. First, we verify the utilization of the memory to capture that "the overhead has a negative impact on performance" (see Table 1). The choice of checking memory instead of CPU utilization is motivated by the nature of the antipattern since the creation and destruction of objects is known to impact on the

TABLE 16: Software Performance Antipatterns - key properties extracted from natural language specification [26].

Name	*Problem specification*	*Key properties*
Concurrent Processing Systems	Occurs when processing cannot make use of available processors because of (i) a non-balanced assignment of tasks to processors or (ii) single-threaded code.	Unbalanced utilization of hardware platforms.
"Pipe and Filter" Architectures	Occurs when the slowest filter in a "pipe and filter" architecture causes the system to have unacceptable throughput.	Software bottleneck, low system throughput.
One-Lane Bridge	Occurs at a point in execution where only one, or a few, processes may continue to execute concurrently. Other processes are delayed while they wait for their turn.	Concurrency not guaranteed due to resources showing different levels of parallelism.
Falling Dominoes	Occurs when one failure causes performance failures in other components.	Large number of failures and their propagation, it refers more to reliability than performance.
Traffic Jam	Occurs when one problem causes a backlog of jobs that produces wide variability in response time which persists long after the problem has disappeared.	Evolution of system response time.
The Ramp	Occurs when processing time increases as the system is used.	Evolution of response time in system services.
More is Less	Occurs when a system spends more time thrashing than accomplishing real work because there are too many processes relative to available resources.	Evolution of frequent thrashing events.

memory allocation [85]. Then, we get the average number of garbage collected objects in the hotspot methods, and we check whether the number of unused objects of each hotspot is larger than the threshold. If this holds, then we report such hotspot and the value of objects in the garbage collector. This condition is introduced to check "when an application unnecessarily creates and destroys large numbers of objects during its execution" (see Table 1).

Appendix B
Rationale for not implementing some Antipatterns

Table 16 lists all the software performance antipatterns that cannot be implemented. The first column shows the *name* of the antipattern, the second column describes the *problem* expressed in natural language, the third column reports the extracted *key properties* that represent the required information for the antipattern detection. In the following we argue on the lack of data from profiling, hence the implementation of these antipatterns is unfeasible.

Concurrent Processing Systems (CPS) occurs when processing cannot make effective use of available processors either because of (i) a non-balanced assignment of tasks to processors or because of (ii) single-threaded code. We do not implement this antipattern since its specification includes causes that cannot be derived by profiling data, in fact both the deployment of software components and the number of threads lack as types of information. Note that profiling data includes the number of running threads, not the number of static threads defined in the code itself. This is why we cannot match the presence of a single running thread with the specification of the antipattern referring to single-threaded code. The impact on the system performance consists of checking the utilization of cpu/memory resources that instead is available, however we think this information is not sufficient to match its textual description.

"Pipe and Filter" Architectures (P&F) occurs when the system throughput is determined by the slowest filter. It means that there is a stage in a pipeline which is significantly slower than all other stages. We do not implement this antipattern because information on the system architecture is not available in profiling data, hence the recognition of a pipeline is not feasible.

One-Lane Bridge (OLB) occurs in concurrent systems when mechanisms of mutual access to a shared resource are badly designed. We do not implement this antipattern since synchronization and locks (i.e., mechanisms to regulate concurrency) lack in profiling data.

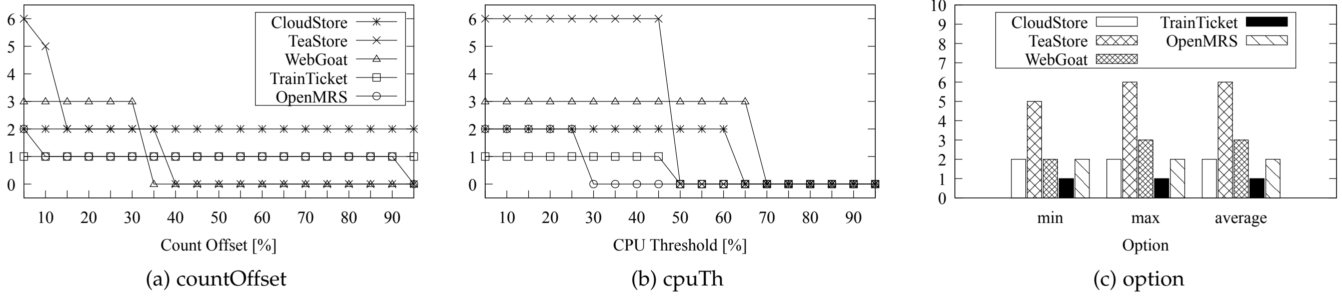
Falling Dominoes antipattern occurs when failures propagates through components. It is not implemented since it refers to reliability and fault tolerance issues that are out of the scope of this paper.

Traffic Jam (TJ), *The Ramp* (TR), and *More is Less* (MiL) are classified in the literature [80] as multiple-values antipatterns, since their detection relies on observing the trend (or evolution) of performance indices along the time. However, at the current stage, the detection of antipatterns is focused on the analysis of single snapshots. In fact, YourKit provides aggregated data, it does not report the continuous evolution of performance indices, but only their overall computation (e.g., method execution time). Moreover, the performance overhead caused by thrashing events (e.g., too many web/-database connections) cannot be recognized on the basis of aggregated data.

Appendix C
Sensitivity analysis on antipatterns' threshold and offset values

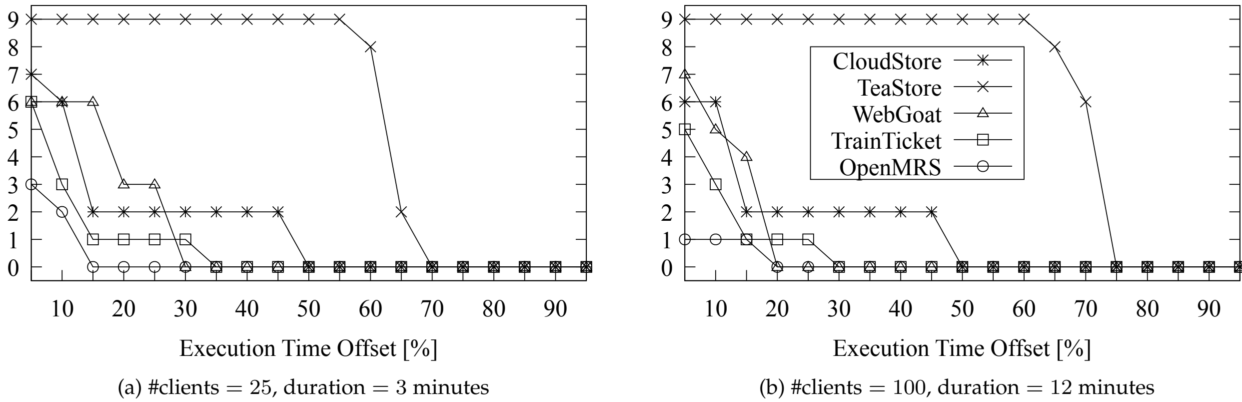
Here we report our investigation about how *thresholds and offsets setup* inputs affect the detection. Specifically, we vary threshold and offset numerical valuesf and study their effect on the number of detected instances.

Figure 3 depicts variations found on the number of detected CTH instances (fixing #clients $= 25$, and duration $= 3$ minutes) when experimenting different values of the input



(a) countOffset (b) cpuTh (c) option

Fig. 3: Number of detected CTH instances (with #clients = 25, duration = 3 minutes).



(a) #clients = 25, duration = 3 minutes (b) #clients = 100, duration = 12 minutes

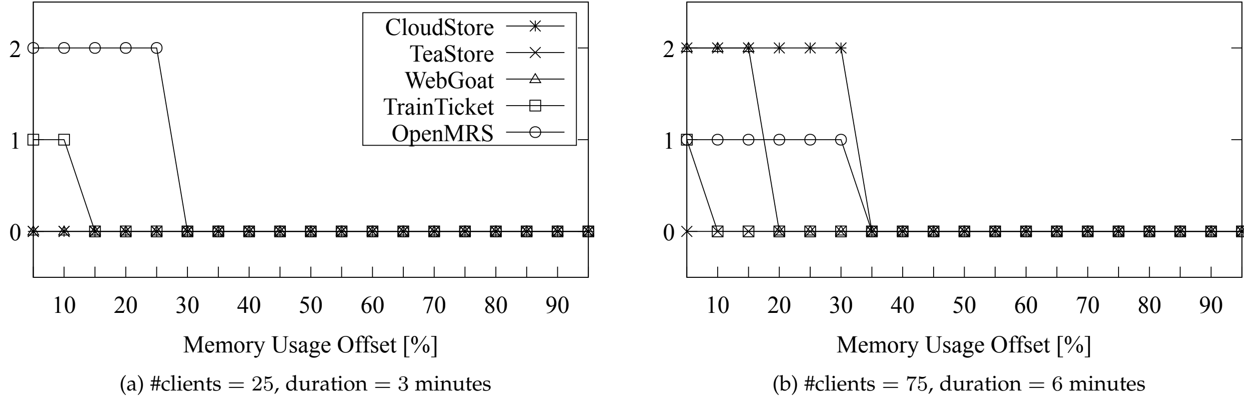
Fig. 4: Number of detected EP instances varying *execTimeOffset* threshold.

parameters (i.e., *countOffset*, *cpuTh*, and *option*, see Algorithm 1) included in the specification of the corresponding antipattern. Figure 3(a) shows that the largest variation is observed for the *TeaStore* system, whose number of CTH instances varies between 6 and 0, no instances are detected for *countOffset* $\geq$ 40%. The number of CTH instances of *OpenMRS* varies between 2 and 0 that are experienced in the extreme cases of setting *countOffset* to 5% and 95%, whereas all other values lead to detect 1 instance of the antipattern. *TeaStore* shows the largest variation also in Figure 3(b) where there are 6 CTH instances for *cpuTh* $\leq$ 45%, 0 otherwise. Figure 3(c) shows that setting the *option* parameter to *average* or *max* allows detecting the same number of CTH instances for all systems, whereas the *min* option reduces the number of detected CTH instances for *WebGoat* (i.e., 3 instances with *average* and *max*, 2 with *min*) and *TeaStore* (i.e., 6 instances with *average* and *max*, 5 with *min*).

Figure 4 reports the variation on the number of EP instances. This antipattern includes one offset value only, i.e., *execTimeOs*. We evaluate two configurations, specifically Figure 4(a) shows #clients = 25 and duration = 3 minutes, whereas Figure 4(b) considers #clients = 100 and duration = 12 minutes. In both configurations, at least 2 EP instances are detected in *CloudStore* when *execTimeOs* $\leq$ 45%. There are 9 instances in *Teastore* for *execTimeOs* $\leq$ 55% and configuration 25-3 ($\leq$ 60% with the 100-12 configuration) that rapidly go to 0 for *execTimeOs* $\geq$ 70% ($\geq$ 75% with the 100-12 configuration). *WebGoat* experiences a large number of EP instances that decreases to zero when *execTimeOs* is $\geq$ 30%. *OpenMRS* shows a different number of instances varying the settings. In Figure 4(a), 3 and 2 instances are detected for low values of the threshold, whereas 1 instance only is detected with a larger number of clients, see Figure 4(b). Similarly, the behavior of *TrainTicket* is affected by the analyzed configuration. With 25 clients, up to 6 instances are detected when *execTimeOs* $\leq$ 30%, whereas up to 5 instances are observed with 100 clients when *execTimeOs* $\leq$ 25%.

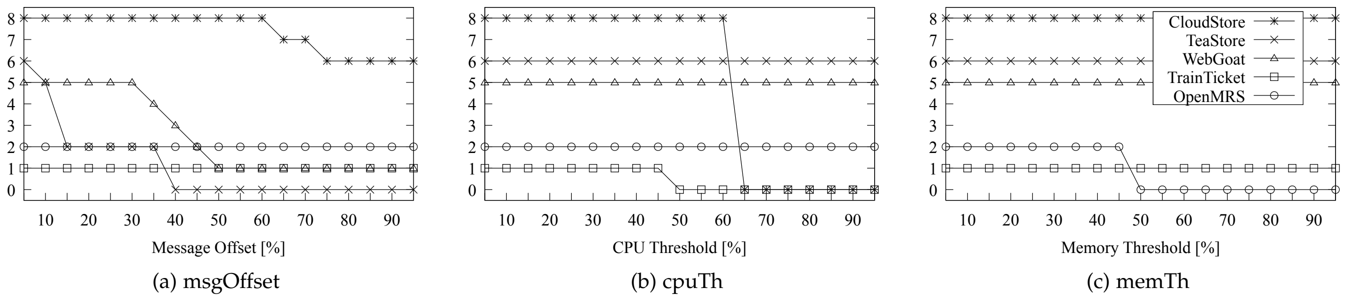
Figure 5 reports the variation on the number of WCS instances. This antipattern includes one offset value only, i.e., *memUsageOs*. We consider two settings, specifically Figure 5(a) shows #clients = 25 and duration = 3 minutes, whereas Figure 5(b) considers #clients = 75 and duration = 6 minutes. In *WebGoat* and *CloudStore*, the number of WCS instances increases with the number of clients. In *TrainTicket*, 1 WCS instance is detected for small *memUsageOs* values independently of the number of clients. In *OpenMRS*, the number of instances decreases when the number of clients is larger and it drops to zero when threshold is greater than 30%. No WCS instances are detected for *TeaStore* in the considered configurations. For TrainTicket and OpenMRS, we get a counter-intuitive result of obtaining less instances in case of a larger number of clients. This confirms that the workload is not directly matched with the number of detected instances, other software performance antipattern types may show up.

Figure 6 depicts the variations found on the number of detected Blob instances (fixing #clients = 25 and



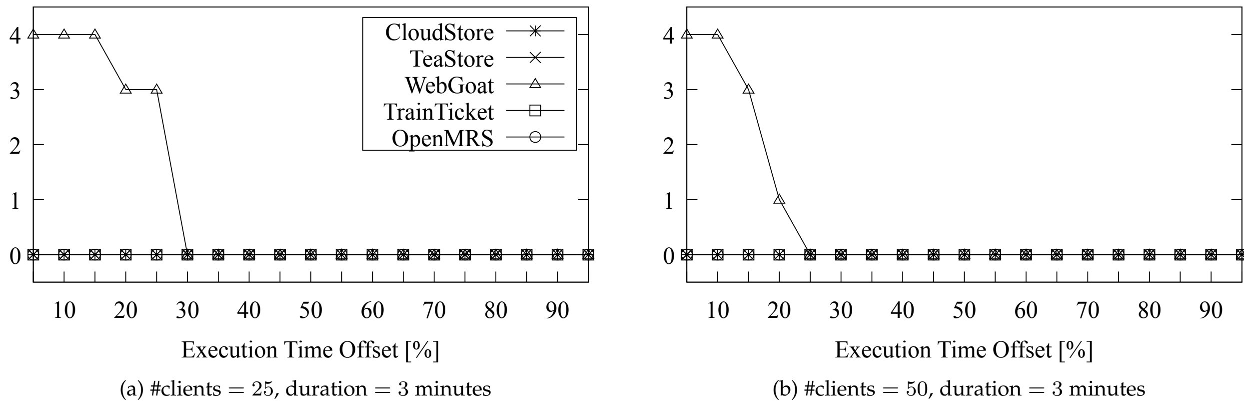
(a) #clients = 25, duration = 3 minutes (b) #clients = 75, duration = 6 minutes

Fig. 5: Number of detected WCS instances varying *memUsageOffset* threshold.



(a) msgOffset (b) cpuTh (c) memTh

Fig. 6: Number of detected Blob instances (with #clients = 25, duration = 3 minutes).

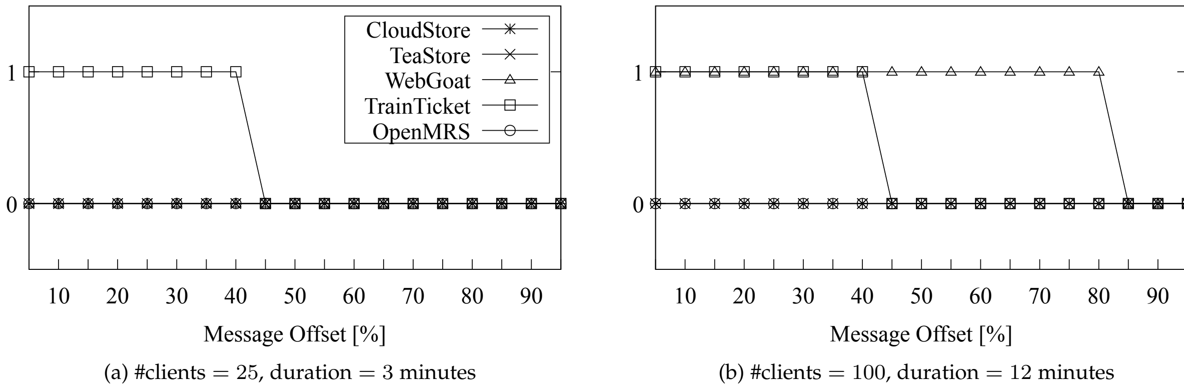


(a) #clients = 25, duration = 3 minutes (b) #clients = 50, duration = 3 minutes

Fig. 7: Number of detected ToB instances varying *execTimeOffset* threshold.

duration = 3 minutes) when experimenting different values of input parameters (i.e., *msgOs*, *cpuTh*, and *memTh*, see Algorithm 4). Figure 6(a) shows that *msgOs* affects the number of instances detected in the systems under analysis. *TrainTicket* and *OpenMRS* are the only systems for which the number of Blob instances is constant with any offset value, all other systems are exposed to variations. As visible from Figures 6(b) and 6(c), *cpuTh* and *memTh* mostly does not affect the number of instances in the considered systems. Few exceptions are: (i) *TrainTicket* since instances go from 1 to 0 when $cpuTh \geq 50\%$; (ii) *CloudStore* whose Blob instances go from 8 to 0 when $cpuTh \geq 65\%$; (iii) *OpenMRS* since instances go from 2 to 0 when $memTh \geq 50\%$.

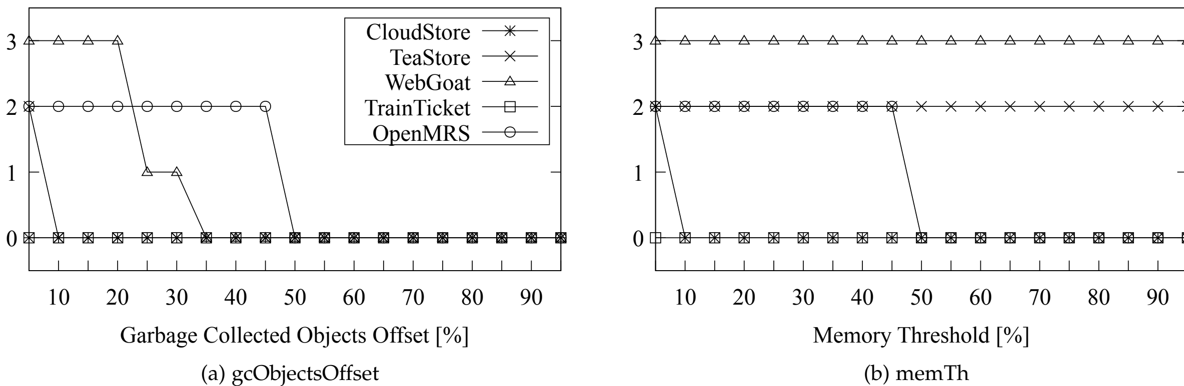
Figure 7 reports variations on the number of ToB instances. This antipattern includes one offset, i.e., *execTimeOs*. We evaluate two configurations, specifically Figure 7(a) shows #clients = 25 and duration = 3 minutes, whereas Figure 7(b) considers #clients = 50 and duration = 3 minutes. Only *WebGoat* is affected by this antipattern, 3 to 4 ToB instances are detected with 25 clients when $execTimeOs < 30\%$, whereas such value varies between 1 and 4 with 50 clients when $execTimeOs \leq 20\%$.



(a) #clients = 25, duration = 3 minutes

(b) #clients = 100, duration = 12 minutes

Fig. 8: Number of detected EST instances varying *msgOffset* threshold.



(a) gcObjectsOffset

(b) memTh

Fig. 9: Number of detected EDA instances (with #clients = 25, duration = 3 minutes).

Figure 8 depicts variations on the number of EST instances. This antipattern includes one offset, i.e., *msgOffset*. We evaluate two configurations, specifically Figure 8(a) shows #clients $= 25$ and duration $= 3$ minutes, whereas Figure 8(b) considers #clients $= 100$ and duration $= 12$ minutes. *TrainTicket* and *WebGoat* are the systems affected by this threshold. *TrainTicket* shows 1 instance up to 40% (in both cases of considering 25 and 100 clients), and later drops to zero. *WebGoat* is instead showing 1 instance when considering 100 clients only, and it drops to zero when the threshold becomes larger than 80%.

Figure 9 reports variations on the number of EDA instances. This antipattern includes two input parameters, i.e., *gcObjectsOffset* and *memTh*. Figure 9(a) shows that the largest variation is observed for the *WebGoat* system, whose number of EDA instances goes from 3 to 1 when *gcObjectsOffset* $\leq 30\%$ and drops to 0 for larger values. The number of EDA instances of *OpenMRS* varies between 2 (experienced in the case of setting *gcObjectsOffset* up to 45%) and 0 for larger values of the offset. *TeaStore* shows 2 EDA instances when *gcObjectsOffset* $= 5\%$. The effect of the *memTh* threshold is observed in Figure 9(b) where the *OpenMRS* system has 2 EDA when setting *memTh* $\leq 45\%$, and the *CloudStore* system shows 2 EDA instances for *memTh* $= 5\%$. *TeaStore* and *WebGoat* show 2 and 3 EDA instances, respectively, independently of the *memTh* value, i.e., this threshold does not affect the number of antipattern instances detected in these systems.

Summarizing, the detection process is indeed affected by the numerical values of thresholds and offsets. Although system characteristics affect the number of detected instances and some thresholds/offsets look marginal for some antipatterns (e.g., *cpuTh* and *memTh* for *Blob*), thoughtfully choosing the value of input parameters allows JPAD to detect a different amount of antipattern instances. This is helpful to reflect the different perception that system stakeholders might have on performance requirements.