Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Scalability testing automation using multivariate characterization and detection of software performance antipatterns[☆]

Alberto Avritzer[a], Ricardo Britto[b,c], Catia Trubiani[d], Matteo Camilli[e], Andrea Janes[e], Barbara Russo[e], André van Hoorn[f], Robert Heinrich[g,*], Martina Rapp[h], Jörg Henß[h], Ram Kishan Chalawadi[b]

[a] eSulab Solutions, Princeton, USA
[b] Ericsson AB, Karskrona, Sweden
[c] Blekinge Institute of Technology, Karskrona, Sweden
[d] Gran Sasso Science Institute, Italy
[e] Free University of Bozen-Bolzano, Italy
[f] University of Hamburg, Germany
[g] Karlsruhe Institute of Technology, Germany
[h] FZI Forschungszentrum Informatik, Karlsruhe, Germany

ARTICLE INFO

Article history:
Received 31 March 2021
Received in revised form 6 May 2022
Accepted 14 July 2022
Available online 23 July 2022

Keywords:
Software Performance Antipatterns
Characterization
Detection
Multivariate analysis

ABSTRACT

Context: Software Performance Antipatterns (*SPAs*) research has focused on algorithms for their characterization, detection, and solution. Existing algorithms are based on the analysis of runtime behavior to detect trends on several monitored variables, such as system response time and CPU utilization. However, the lack of computationally efficient methods currently limits their integration into modern agile practices to detect *SPAs* in large scale systems.
Objective: In this paper, we extended our previously proposed approach for the automated *SPA* characterization and detection designed to support continuous integration/delivery/deployment (CI/CDD) pipelines, with the goal of addressing the lack of computationally efficient algorithms.
Method: We introduce a machine learning-based approach to improve the detection of *SPA* and interpretation of approach's results. The approach is complemented with a simulation-based methodology to analyze different architectural alternatives and measure the precision and recall of our approach. Our approach includes *SPA* statistical characterization using a multivariate analysis of load testing experimental results to identify the services that have the largest impact on system scalability.
Results: To show the effectiveness of our approach, we have applied it to a large complex telecom system at Ericsson. We have built a simulation model of the Ericsson system and we have evaluated the introduced methodology by using simulation-based *SPA* injection. For this system, we are able to automatically identify the top five services that represent scalability choke points. We applied two machine learning algorithms for the automated detection of *SPA*.
Conclusion: We contributed to the state-of-the-art by introducing a novel approach to support computationally efficient *SPA* characterization and detection that has been applied to a large complex system using performance testing data. We have compared the computational efficiency of the proposed approach with state-of-the-art heuristics. We have found that the approach introduced in this paper grows linearly, which is a significant improvement over existing techniques.

1. Introduction

The performance assessment and improvement of large distributed systems is challenging because of the need to systematically assess a complex dynamic ecosystem (Microsoft, 2019). The identification of scalability choke points is often expensive, and it involves load testing and complex analysis by performance experts.

To facilitate the avoidance, identification, and correction of performance-related issues, researchers have identified and formalized a series of architecture anti-patterns. Architecture anti-patterns are defined as recurring design problems (Brown et al., 1998), which might have a significant impact on the software quality attributes. In the context of software performance, there

[☆] Editor: Raffaela Mirandola.
[*] Corresponding author.
E-mail address: robert.heinrich@kit.edu (R. Heinrich).

https://doi.org/10.1016/j.jss.2022.111446
0164-1212/© 2022 Elsevier Inc. All rights reserved.

are several Software Performance Antipatterns (*SPAs*) that have been defined in the literature (Smith and Williams, 2000; Wert, 2015).

In this work, we consider the following *SPAs* (Wert, 2015): Application Hiccups — repeated violations of the baseline response time requirement; Continuous Violated Requirements — continuous violations of the baseline response time requirement, for every evaluated load; Traffic Jam — high variability in the externally observed system response times; The Stifle — a software service or component that issues many short database calls; Expensive Database Call — few long database calls; Empty Semi Trucks — a transaction that issues many short messages; The Blob — a component or service that acts as a central message processor.

The characterization of each *SPA* depends on the specific performance issue. For example, a *Traffic Jam* is triggered by queuing for software resources that could be a result of contention for semaphores. In contrast, an *Application Hiccup* could occur as a consequence of garbage collection events, or database backups. Therefore, *SPAs* could be triggered by several factors, such as programming errors, environmental mis-configurations, or high-level design flaws.

The extensive body of knowledge has addressed several aspects of SPAs like *SPA* classification and solution (Smith and Williams, 2012), early detection/solution at the design phase (Cortellessa et al., 2014), methodologies to rank *SPAs* occurring in design models (Trubiani et al., 2014), detection/solution during the testing or operational phases (Wert et al., 2013), load testing and profiling to detect *SPAs* in Java applications (Trubiani et al., 2018), and an automated approach for detection in load testing and production (Wert, 2015).

The existing approaches use searching for several *SPAs* for each load test result. In addition, for every evaluated *SPA*, a heuristic is executed to evaluate the load test data. Therefore, if there are N load tests, M *SPAs* to be evaluated, and given a worst-case heuristic cost of H_M among the evaluated *SPAs*, the computational complexity of the state-of-the-art algorithms is $\mathcal{O}(H_M \cdot N)$. Even though polynomial, the computational effort required for each load test could be expensive. Therefore, the lack of computationally efficient methods for *SPA* characterization limits the adoption of *SPA* detection in continuous integration/delivery/deployment pipelines (CI/CDD) of large and complex systems (Avritzer et al., 2020).

By taking a holistic view on the literature about the application of performance testing approaches to CI/CDD, we learned that it mainly consists of theoretical contributions (Avritzer et al., 2020; Javed et al., 2020a; Laaber, 2019a) suggesting recommendations for performance testing implementation using existing tools.[1] The state-of-the-art approaches for detecting performance issues consist of the execution of automated load testing combined with Application Performance Monitoring (APM) tools (Heger et al., 2017).

To be useful for engineers, the automated characterization and detection of *SPAs* need to provide timely results and be integrated into shorter feedback loops. To do so, a logical solution is to integrate this type of approach into CI/CDD pipelines. However, the state-of-the-art approaches cannot be easily integrated into CI/CDD pipelines due to their computational complexity. The state-of-the-art algorithms (Cortellessa et al., 2014; Trubiani et al., 2018; Wert, 2015; Bran, 2017) that are being used to detect *SPAs*, employ detailed knowledge about the system internals and execute a search over the candidate *SPAs*, while checking large portions of the monitored performance data. Additional knowledge about system internals are not usually available in large system logs, and are computationally expensive to process in real-time for large monitored data, as required by CI/CDD pipelines. In contrast to measuring system-internal data, common monitoring tools routinely track response time measurements.

In our previous paper (Avritzer et al., 2021a), we started addressing the gap mentioned above by proposing a multivariate approach to characterize and detect *SPAs* in the context of scalability testing. We introduced a black box approach that only uses response time data, which makes it a good fit for CI/CDD pipelines. Our approach has been engineered into the PPTAM software toolchain (Avritzer et al., 2018) and empirically evaluated by means of an industrial case study conducted at Ericsson. The case was a large real-time telecommunication system that is expected to serve millions of users per second.

This paper extends our previous work (Avritzer et al., 2021a) by addressing the following limitations:

- The proposal in our previous paper lacked support to interpret the detection results, what could make it difficult for engineers. We have improved on this aspect by using machine learning (unsupervised learning) to support the interpretation of the detection results (mainly Section 3.4).
- Our previous work lacked an evaluation of precision and recall. We addressed this limitation by evaluating our approach using simulation (mainly Section 4.4).

In this paper, we answer the following research questions:

RQ1: What is the computational complexity of the proposed approach?

RQ2: What are the precision and recall of the proposed approach?

The remainder of the paper is organized as follows. Section 2 provides the relevant background. Section 3 introduces an high-level an overview of our approach, while Section 4 presents the research design, which includes the description of the telecommunication system case study, and the data collection approach. Section 5 illustrates the main steps of our approach and discusses their application to our case study. Section 6 describes the evaluation of the proposed approach and answers our research questions. Section 7 introduces a discussion of our approach including generalizability as well as threats to validity. Section 8 discusses related work. Finally, Section 9 presents our conclusions and challenges ahead.

2. Background

In this section, we present relevant concepts, including the motivation for detecting *SPAs*, along with a brief description of their main characteristics, followed by some examples illustrating their effect in application scenarios. We also present more details about multivariate classification approaches.

2.1. SPAs

Motivation. The reason for applying *SPA* detection is that it has been demonstrated to be beneficial, even in industrial contexts (Trubiani et al., 2018). More in general, the detection of performance problems is considered very relevant. Many studies are recently focusing on performance regressions (Liao et al., 2021; Chen et al., 2020) as well as modeling performance violations over time (Camilli and Russo, 2022). The benefit of detecting SPAs is that their specification includes both the problem and the solution, hence software developers are supported in the task of improving the system performance. *Description*. SPAs have

[1] https://www.loadview-testing.com/blog/ci-cd-load-testing-deploying-performance-optimized-applications/,https://www.neotys.com/insights/automation-testing.

been defined in the literature as bad practices leading to performance flaws (Smith and Williams, 2012; Parsons et al., 2008). Hereafter, we briefly present the bad practices considered in our study; their characterization is detailed in Section 3.2. Specifically, this manuscript focuses on the following seven performance antipatterns (Smith and Williams, 2012; Wert, 2015):

- *Application Hiccups* – Occurs in the case of temporarily increased response times (i.e., hiccups) caused by periodic tasks that either overload the system or block other requests (e.g., OS routines, garbage collection).
- *Continuous Violated Requirements* – Occurs in the case of a continuous violation of the response time requirements.
- *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.
- *The Stifle* – Occurs when data is retrieved by means of many similar (or equal) database queries. As each database request entails a considerable overhead, the high amount of database requests leads to a performance problem.
- *Expensive Database Call* – Occurs when there exist single database calls that show a long execution time. Differently from The Stifle antipattern, the performance overhead is not caused by a large number of database accesses, but by a single and long-running database request.
- *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 (e.g., in message-based systems with a huge load of messages, each containing a small amount of information), an inefficient interface (e.g., too fragmented access to data), or both.
- *The Blob* – Occurs when a single class or component either (1) performs all of the work of an application or (2) holds all of the application's data. Either manifestation results in excessive message traffic that can degrade performance.

There exist some further antipatterns defined in the literature (Smith and Williams, 2012; Wert, 2015) and used in related work, see more details in Section 8.3. Specifically: *Extensive Processing* (i.e., a long running process monopolizes a processor and prevents other jobs to be executed); *Circuitous Treasure Hunt* (i.e., a high amount of requests to retrieve the data), *Wrong Cache* (i.e., memory pollution through improper use of a cache); *The Ramp* (i.e., the processing required to satisfy a request increases over time); *One Lane Bridge* (i.e., mutual access to a shared resource is badly designed).

The detection and solution of performance antipatterns can be performed at different stages of the software development process, and approaches can be reviewed accordingly.

In the case of early detection/solution, i.e., during the design phase: (i) in Cortellessa et al. (2014) a first-order logic representation of performance antipatterns is provided, specifically a set of rules express the system properties under which antipatterns occur; (ii) in Trubiani et al. (2014) a methodology to rank performance antipatterns occurring in Palladio-based design models (Reussner et al., 2016) is proposed and applied to optimize (i.e., by reducing the number of design alternatives to be analyzed) the solution process.

In the case of late detection/solution, i.e., during the testing or operational phases, there exist two approaches (Wert et al., 2013; Trubiani et al., 2018) working in this direction. In Wert et al. (2013) a performance antipatterns detection approach is presented, more specifically a decision tree specifying the performance problems hierarchy is used to capture the root causes of the identified performance problems; The nature of the data required for a detailed analysis for each *SPA*, as introduced in Wert et al. (2013) are not usually collected in CI/CDD pipelines. For example, for Empty Semi-Trucks, messaging related information such as the number of connections and the number of messages would be required. In contrast, in the work introduced in this paper, we focus on average response time patterns as defined by the first and second moments of the response time. In Trubiani et al. (2018), load testing and profiling data is exploited to detect performance antipatterns in Java applications, and an industrial case study demonstrates the usefulness of detecting and solving antipatterns for system performance improvement.

Examples. In the following we describe two applications as well as possible scenarios leading to the visible manifestation of *SPAs*.

Smart Parking. Let us consider cars looking for an empty parking spot. Each request triggers the server to acquire updated information about a specific zone. Let us also assume that there exist scanning cars that periodically send images on found available spots. In case a large number of cars is configured to contact the server at a specific point in time (e.g., 6 pm), then the *Traffic Jam* antipattern is triggered, in fact a backlog of jobs is generated, and wide variability in response time is expected at the server side. To avoid such a scenario, it is better to design the system so that scanning cars randomly send images to the server. This way, all images can be parsed without generating a peak of incoming requests to be simultaneously processed by the server. We let the reader refer to Pinciroli et al. (2021) for further examples of antipatterns occurring in the Smart Parking scenario.

Medical Application. Let us consider one service displaying a list of all patients, and the doctor has to scroll through it to select the current patient before retrieving her/his history of diseases. It is quite evident that this system behavior matches with the *Expensive Database Call* antipattern, since a single database call is executed, and useless information is displayed after a while. This can be avoided by allowing to introduce the names of patients and accessing the database to get information more efficiently. We let the reader refer to Smith (2020) for further examples of antipatterns occurring in the Medical Application.

2.2. Multivariate classification approaches

A *partially ordered set* (or simply poset) consists of a set along with a binary relation indicating that, for specific pairs of elements, one of the elements is greater than the other one in the ordering. Namely, we say that two elements are numerically ordered, i.e., $e_1 > e_2$ if each variable value in e_1 is greater than the correspondent variable value in e_2.

A Hasse diagram (Skiena, 1990) is a graph structure used to represent a poset. A Hasse diagram represents a convenient visualization method when the multiple values associated with the nodes in the poset can be numerically ordered. In this paper, we use Hasse diagrams based on two variable values that are numerically ordered to prioritize software components according to their impact on the software scalability of the system under study.

The theory of scalogram algebra and associated heuristics known as Multiple Scaling by Partial Order Scalogram Analysis by Coordinates (POSAC) was introduced in Shye (1985) to generalize the concept of one dimension (Guttman scale (Maggino, 2014)) to several dimensions. When a one-dimension variable can be numerically ordered, it is called a Guttman Scale (Maggino, 2014). In this paper, we apply concepts derived from the POSAC (Shye, 1985) methodology to partition the performance antipattern domain into regions using the structure induced by the measurement variables.

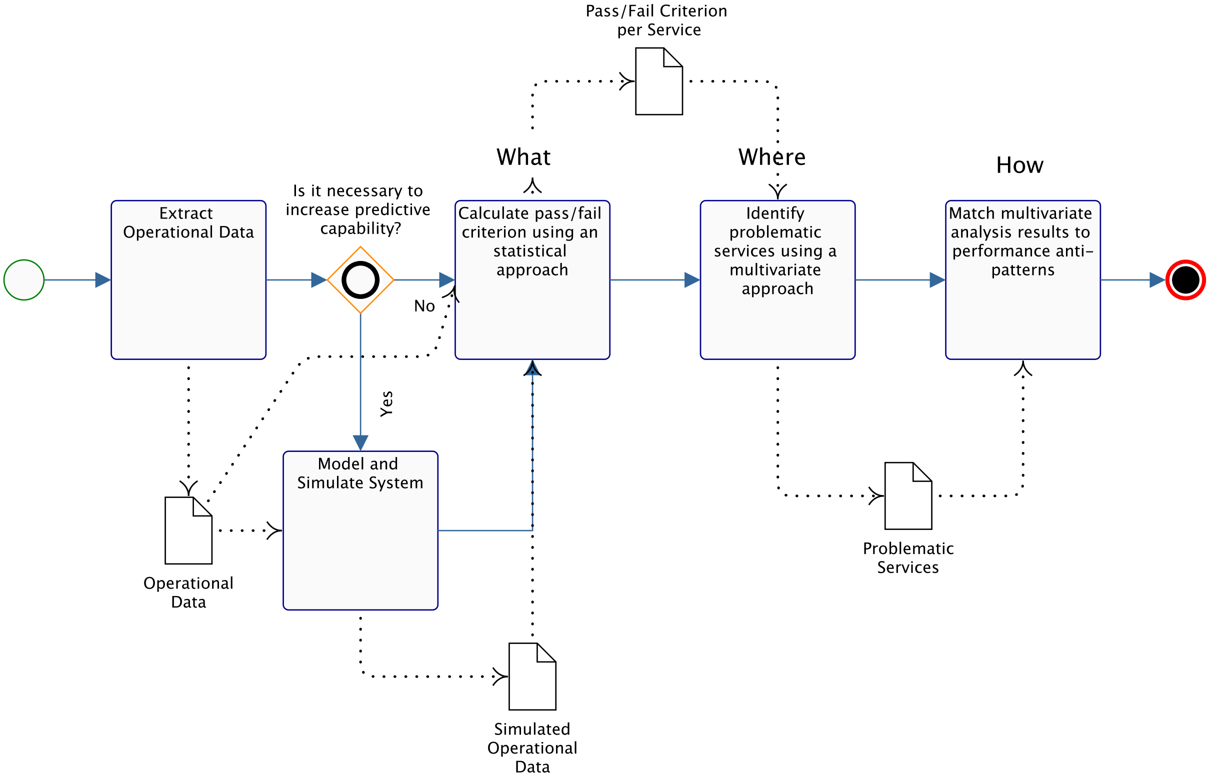

Pass/Fail Criterion per Service
What
Where
How
Extract Operational Data
Is it necessary to increase predictive capability?
Calculate pass/fail criterion using an statistical approach
Identify problematic services using a multivariate approach
Match multivariate analysis results to performance anti-patterns
No
Yes
Model and Simulate System
Operational Data
Problematic Services
Simulated Operational Data

Fig. 1. High-level schema of the proposed approach.

3. The proposed approach

In this section, we present our approach to characterize and detect *SPAs* that can be integrated into CI/CDD pipelines. The section is organized as follows. Section 3.1 introduces an high-level overview of our approach. Section 3.2 presents the performance modeling approach used to characterize *SPAs*. Section 3.3 describes the approach used for performance modeling parametrization using measurements from the telecommunication system case study. Finally, Section 3.4 presents the application of the POSAC procedure to *SPAs* detection.

3.1. Overview

The approach, illustrated in Fig. 1, consists of three main parts: (1) *What*, the automated calculation of the pass/fail criteria, (2) *Where*, the automated identification of the problematic services, (3) *How*, the automated analysis of the performance results to match the detected performance problems to their root causes, using the specification of software performance anti-patterns. The shaded boxes in Fig. 1 represent our contributions. The major boxes are briefly described below.

1. Extract operational data. This step was described in our previous research (Avritzer et al., 2020; Avritzer and Weyuker, 1995). It aims at creating a model of operational usage based on the most-likely usage scenarios.
2. *What*. Calculate pass/fail criteria based on automatically computed baseline requirements from load test response time measured under small workload. The automated scalability assessment approach used to define scalability requirements follows the approach introduced in Avritzer et al. (2020). This approach computes average response time and standard deviation at low loads to automatically compute the scalability requirement for each method.
3. *Where*. Define a multivariate approach that can be used to create a *poset* of the evaluated services. Identify a set of services, in the poset, that have the largest impact on system scalability.
4. *How*. Match multivariate analysis results to performance anti-patterns. Multivariate analysis is used to characterize and detect *SPAs*.
5. *Model and Simulation*. Conduct simulation activities to validate modeling assumptions.

Our approach uses a multi-layer decomposition approach based on performance modeling and multivariate characterization of *SPAs*. In doing so, it decouples the performance cost of *SPA* characterization, which can be done offline, from the cost of executing the (*SPA*) detection procedure, which needs to be efficient enough to be integrated into the CI/CDD pipelines. In particular, the cost of *SPA* detection in our approach consists of the execution of the multivariate characterization of load test results; it is not a function of the number of evaluated *SPAs*.

The *induced partition of the studied domain* creates a profile based on the two-coordinate values: (1) *slope* of the fitted linear regression line of the maximum response time measurement for each of the evaluated loads in the y-axis vs. load level in the x-axis; and (2) *normalized distance* between the maximum response time and the baseline performance/scalability requirement.

Intuitively, the slope of the maximum response time regression line is an indication of the performance degradation as a function of the offered load. The normalized distance is an indication of the system ability to meet the scalability requirements and is related to the customer perception of system performance.

The normalized distance, *nd*, is evaluated through Eq. (1), where M is the measurement and B is the baseline.

$$nd = 2 \cdot \frac{M}{M+B} \tag{1}$$

The normalized distance is evaluated to 1 when the measurement is equal to the baseline, <1 when the measurement is lower than the baseline, and >1 when the measurement is larger than the baseline.

3.2. SPA characterization

The approach introduced in this section relies on the assumption that the specification of *SPAs* includes the most common performance issues introduced by programming and configuration errors. This creates network queuing bottlenecks that are ultimately reflected in the perceived customer experience through the first and second moments of the response time. In addition, to be efficiently integrated into CI/CDD pipelines, an analytical solution to the queuing *SPA* induced queuing problem needs to be implemented. The $M/G/1$ is the most general model for which we have a simple closed form analytical solution for the average waiting time (W) (Bertsekas and Gallager, 1992). In this section, we abstract from each of the most relevant *SPAs* by using the $M/G/1$ queue with vacations model. We instantiate from the General distribution, for each *SPA*, by specifying its first and second moment as shown in Table 1. In summary, we propose to use a model with Poisson Arrivals (M) and general service distributions (G) for several reasons. First, it allows to model service variability by the parametrization of G. Second, the Poisson arrivals assumption has been shown to be a reasonable approximation for the arrivals process, in several domains (Kleinrock and Collection, 1974), where the independence assumption for the user arrival process can be modeled (Bertsekas and Gallager, 1992).

Let us describe how the single server $M/G/1$ model, and its analytical P–K (Pollaczek–Khinchine) (Bertsekas and Gallager, 1992) solution, can be used to characterize the impact of *SPA*s on the system under study performance, as defined by the average response time and variance.

In the single server $M/G/1$ queueing model, M represents the Poisson process arrivals assumption and G stands for general, i.e., any service time distribution of the single server (Bertsekas and Gallager, 1992). $M/G/1$ is the appropriate model to represent *SPAs*, because the most common impact of *SPAs* is to introduce single service bottlenecks (Avritzer and Weyuker, 2004).

The first moment of a random variable is the Mean. The second moment of a random variable X around a is defined as $E\left[(X-a)^2\right]$. Variance is the second moment around the mean (Bertsekas and Gallager, 1992). Variance is a measure of the distribution spread around the mean.

The $M/G/1$ P–K (Pollaczek–Khinchine) formula (Bertsekas and Gallager, 1992) for average waiting time in the system, W, and residence time, T, as a function of the first and second moments of the service time distribution, $\overline{X}$, and $\overline{X^2}$, and system arrival rate, λ is given by (Bertsekas and Gallager, 1992):

$$W=\frac{\lambda\overline{X^2}}{2(1-\lambda\overline{X})} \tag{2}$$

and

$$T=\overline{X}+W \tag{3}$$

In the following, we describe how the $M/G/1$ model can be used to characterize the performance of the *SPAs* considered in this paper.

3.2.1. Application Hiccups

The *Application Hiccups* (Wert, 2015) is characterized by repeated violations of the scalability requirement. There are several approaches for modeling the *application hiccups*, such as, $M/G/1$ queues with vacations that occur when the system transitions to the idle state, and using queues with preemptive or non-preemptive priorities. In this section, we model *Application Hiccups* by using the $M/G/1$ with vacations model, described below. Using the same notation as in Eq. (2), and using as the first and second moments of the vacation time distribution, $\overline{V}$, and $\overline{V^2}$, the $M/G/1$ with vacations average waiting time in the system is given by Bertsekas and Gallager (1992) and shown in Eq. (4):

$$W=\frac{\lambda\overline{X^2}}{2(1-\lambda\overline{X})}+\frac{\overline{V^2}}{2\overline{V}} \tag{4}$$

3.2.2. Continuous Violated Requirements

The Continuous Violated Requirements (Wert, 2015) is characterized by the continuous violations of the scalability requirement, for every evaluated load. The approach used in this section, for modeling the Continuous Violated Requirements is to use an $M/D/1$ queue, with an average service time larger than the computed baseline requirement. D stands for deterministic and represents a constant service time.

3.2.3. Traffic Jam

The Traffic Jam software antipattern represents high variability in the externally observed system response times that is caused by queuing at software resources. In this section, we model the Traffic Jam *SPA* by using the $M/G/1$ queuing system, and modeling the increase in the system variability in the departure process function G.

3.2.4. The Stifle

The Stifle software antipattern (Wert, 2015) represents a software component that issues many short database calls to implement a service. In this section, the Stifle *SPA* is modeled by using the $M/G/1$ queueing system, where G is modeled by an Erlangian distribution with $k-stages$ (Bertsekas and Gallager, 1992), $M/E_k/1$. Each of the k states in the Erlangian distribution is used to model the Stifle fan-out of one call to k serial database calls.

3.2.5. Expensive Database Call

The Expensive Database Call software antipattern (Wert, 2015) represents few long database calls and can be modeled by using an $M/G/1$ queuing system with a long tail distribution for service times.

3.2.6. Empty Semi Trucks

The Empty Semi Trucks software antipattern (Wert, 2015) represents a transaction that issues many short messages in series to implement the transaction. Therefore, this antipattern can be modeled similarly to the *Stifle* antipattern, by using a $M/G/1$ queueing system, where G is modeled by an Erlangian distribution with $k-stages$ (Bertsekas and Gallager, 1992), $M/E_k/1$.

3.2.7. The Blob

The Blob antipattern (Wert, 2015) represents a component that manages most of the overall messages in the system. As a consequence of being the focus of messages, message processing performance degradation is observed. Therefore, the Blob software antipattern can be modeled similarly to the Expensive Database Call software antipattern. In Section 3.3 we describe the approach that was used for the parametrization of the *SPA* models just described, by using measurements derived from the large telecom software, as described in Section 4.1.

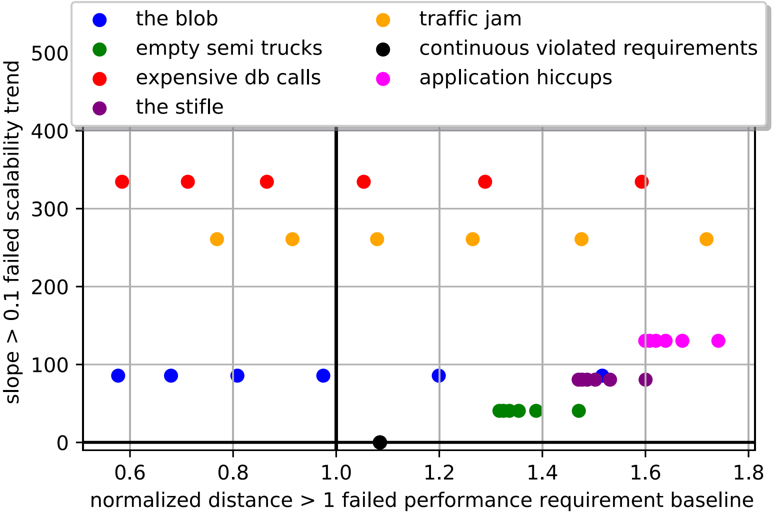


Fig. 2. Partition induced by slope and normalized distance by the *SPA*s analyzed, for the loads 40%, 50%, 60%, 70%, 80%, and 90%.

Table 1
SPA modeling parametrization approach.

SPA	Model	Service used	Calibration approach	$\overline{X}$	$\overline{X^2}$
Application hiccups	$M/G/1$ with vac.	Control	Add vacation variability per control	5.01	88.1
Continuous violated req.	$M/D/1$	Interrogation + c	Add c = 5 to baseline	26.95	0
Traffic jam	$M/G/1$	Control	Double variability	5.01	177.6
The stifle	$M/E_k/1$	Interrogation	Use 10 stages	99.5	1089.4
Expensive DB call	$M/G/1$	Enquiry	Double $\overline{X}$	7.2	327.6
Empty semi-trucks	$M/E_k/1$	Control	Use 10 stages	50.1	276.1
The blob	$M/G/1$	Database management	Double $\overline{X}$	6.6	71.98

Table 2
Partition induced by slope and normalized distance by the *SPA*s, for the load 90%.

	Normalized distance	Slope
Application hiccups	1.74	130.42
Continuous violated requirements	1.08	0.00
Traffic jam	1.72	260.84
The stifle	1.60	80.54
Expensive database call	1.59	334.51
Empty semi-trucks	1.47	40.55
The blob	1.51	79.56

3.3. Analytical model parametrization

In this section, we present an illustration of one possible *calibration* that was used for the parametrization of the *SPAs* performance models.

The *SPA* model parametrization approach shown in Table 1 presents the model used, the service from the case software selected for calibration, the calibration approach, and the first and second moments selected to represent the *SPA*. The results obtained for each *SPA* multivariate characterization are presented in Table 2 and Fig. 2. They show the two-dimensional (x, y) coordinates, i.e., the normalized distance and slope, for each of the evaluated *SPAs*. Table 2 shows the multivariate pair for the 90% load experiment, while Fig. 2 shows the multivariate pairs associated with the evaluated *SPAs*, for loads varying from 40% to 90%.

3.4. SPA detection

In Shye (2009), a step-by-step procedure is presented to illustrate the application of Multiple Scaling by Partial Order Scalogram Analysis by Coordinates (POSAC) in the behavioral sciences domain. POSAC induces a partition of the studied space, because profiles are distinguished by high/low values of the x, y coordinates. In the application of POSAC described in Shye (2014) the interpretation of the POSAC coordinates represents the subjective concern in the target research community. For example, in the assessment of quality of life, a two-dimensional space representing intelligence and well-being was used.

In this section, we present the application of POSAC to the *SPA* detection domain. The x coordinate, the normalized distance of the performance requirement, represents the impact of response time requirement violation to the user, or user well-being Shye (2014). In addition, the y coordinate, the slope of the linear regression of response time as load levels increase, represents the risk that the user will be impacted by the response time degradation, or system intelligence (Shye, 2014).

The notion of POSAC can be used to partition the *SPA* domain by means of the following steps:

1. Define a mapping sentence to create a framework for the system of observations.
2. Determine the empirical structure of the content universe.
3. Execute the POSAC algorithm using the (x, y) coordinates of the location of their profile in space.
4. Interpret the POSAC coordinates to understand the induced partition on the POSAC space.

4. Research design

To address our research questions, we conducted an industrial case study. In this section, we describe the case and the research design of our investigation. Section 4.1 introduces our case and unit of analysis, that is, a large real-world telecommunication system. Section 4.2 describes data collection carried out by using load testing and a simulation model. Section 4.3 describes the

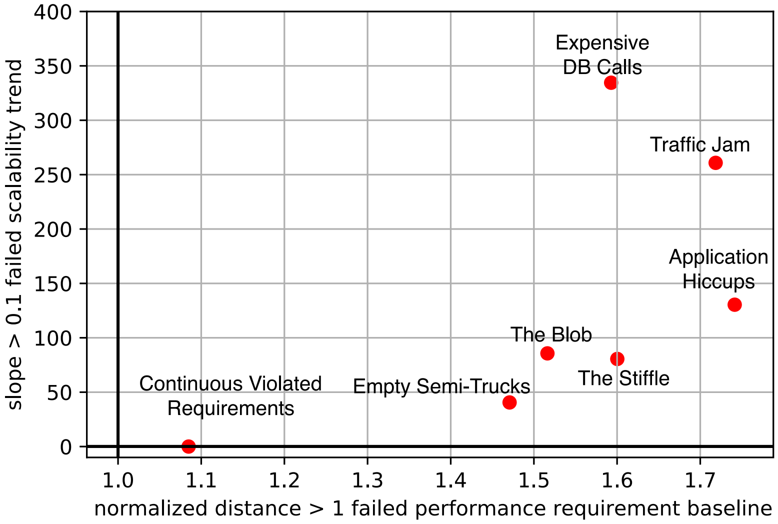


Fig. 3. Partition induced by slope and normalized distance by the *SPAs* analyzed, for loads = 90%.

method we adopted to evaluate the computational complexity of our approach, while Section 4.4 describes the measures we used to evaluate its accuracy.

4.1. The case and unit of analysis

The case and unit of analysis of our case study is a large and complex real-time telecommunication system developed by Ericsson. The system is composed of 20 subsystems which are developed using a Service-Oriented Architecture (SOA). The system is developed by many distributed teams using agile software development practices.

The system has hard performance requirements and is expected to handle millions of users per second. Furthermore, the system shows many performance indicators. Therefore, monitoring the system's performance is challenging through a manual approach. The scale of operation makes the system particularly interesting for our investigation. Furthermore, it represents a relevant and representative example of a performance-critical application.

The main hardware and software performance-critical components of our target system are: network, network interface subsystem, processor subsystem, and database. The system receives requests from a network, which are received by a load balancer and then forwarded to the two subsystems of interest. The network interface subsystem provides services that are invoked by the network. The processor subsystem processes requests that are forwarded by the network interface subsystem to our subsystem of interest running 12 performance-critical services.

4.2. Data collection

We collected data from load testing and simulation experiments using a load testing environment and a simulation model, respectively.

Load testing. The performance data was the result from a 21-hour long load testing session of the system investigated in this paper. The data was collected in December 2019. It includes data associated with 12 services provided by the case software system.

During the testing session, the load varied between 40% and 100% (where 100% represents a load of 16k transactions per second), in steps of 10%. Based on the operational data, the frequency of occurrence of such workload intensity values is as follows: $40\% = 0.24, 50\% = 0.05, 60\% = 0.14, 70\% = 0.10, 80\% = 0.14, 90\% = 0.24$, and $100\% = 0.10$.

Simulation modeling. To evaluate the precision and recall of our approach (RQ2), we present in this section a simulation model designed with Palladio (Reussner et al., 2016), i.e., a tool-supported approach to model and analyze software architectures for performance prediction among other quality attributes.

We choose Palladio because it is an established approach for performance prediction and provides mature tooling (Heinrich et al., 2018) (e.g., model editors, simulators and experiment automation) to support us in the study. It has been applied before for identification and solving of anti-pattern in simulated software architectures by Trubiani et al. (2014).

We created the following partial models using Palladio:

- The *component repository model* specifies the software components and their interfaces stored in a repository. The components' inner behaviors are specified in so-called Service Effect Specifications.
- The software architecture is described in the *system model* by assembling components from the repository.
- The processing resources (CPU, hard disk, and network) of execution containers are specified in the *resource environment model.*
- The *allocation model* describes the deployment of the components to the resources.
- User behavior and usage intensity (i.e. workload) of the system are described in the *usage model.*

The models are annotated with performance-relevant annotations, such as capacities of resources like CPUs and HDDs and resource demands of software actions, and then applied for simulation-based performance prediction.

Fig. 4 shows the base simulation model on a conceptual level taking the Enquiry operation as an example. The Enquiry operation involves calls to services offered by the Network Interface Service (NIS), Database (DB) and Processor (PROC) software components.

The Enquiry operation's Service Effect Specification describes the operation's control flow graph by specifying the components' resource demanding internal behavior as internal actions and the connection between components by external actions that connect the various services.

The Enquiry operation first calls the NIS component through the *«EntryLevelSystemCall» NIS.enquiry* action. Next follows the internal behavior modeled by the *«InternalAction» NIS.enquiry*. This step consumes an exponential distributed resource demand

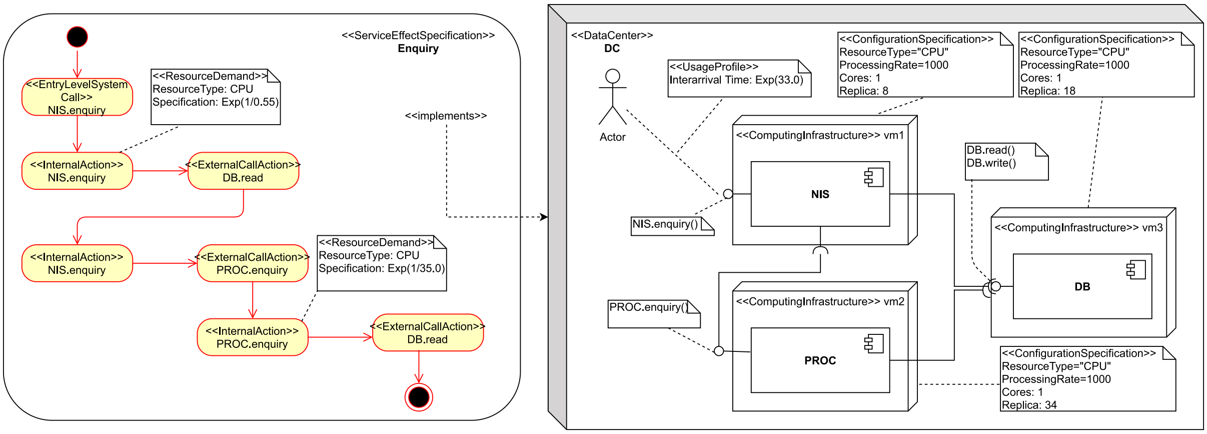


Fig. 4. Conceptual level of simulation model with usage scenario 40% load.

with a rate of 1/0.55 for internal processing. Then, the operation calls the DB component through the *«ExternalCallAction» DB.read* action. For reasons of simplicity, the service demands for the database read are not shown in the figure, but are included in the model. Afterwards it calls the PROC component by the *«ExternalCallAction» PROC.enquiry* action. Next follows the *«InternalAction» PROC.enquiry* action taking an exponential distributed CPU resource demand with a rate of 1/35. Finally it calls the DB component via *«ExternalCallAction» DB.read* action again without resource consumption.

The right-hand side of Fig. 4 illustrates the deployment of software components to hardware nodes. The model assumes that each software component is deployed on a virtual machine (VM). A resource container represents a single VM with one CPU core working at a processing rate of 1000 to reflect the use of service time specifications in milliseconds. Furthermore, there exist 8 instances of the NIS component, 34 instances of the PROC component and 18 instances of the DB component. Each component instance is deployed on a single resource container offering distinct services. The NIS component is deployed on the resource container *«ComputingInfrastructure» vm1* offering the service *enquiry*. The PROC component is deployed on the resource container *«ComputingInfrastructure» vm2* offering the service *enquiry*. The DB component is deployed on the resource container *«ComputingInfrastructure» vm3* offering the services *read* and *write*.

Based on the structural model, we approximated the stochastic distributions of resource demands of the modeled actions that resemble CPU and IO based operations. We applied different resource demand estimation techniques to create our baseline-calibrated model from the provided log data.

Our initial approach uses the minimum response times provided in the logged data as approximation for the service time without congestion. Then, we combined those results with estimates from applying the utilization law (Daniel et al., 2004) to estimate the mean service time and added additional variance to the model to match the log data.

Our more advanced approach used the LibReDe framework (Spinner et al., 2014) to create a state model of the connected call hierarchy and approximate the resource demand for each service individually based on the observed data. LibReDe uses estimation algorithms like least square regression or Kalman filters to find solutions for the formulated state model that minimize the error. However, due to high degree of aggregation (5 min averages) in the observation data, we were not able to find a suitable and plausible solution for all services using this approach.

Our calibration approach focused on creating a simulation model with similar mean response times and similar levels of CPU utilization. Thus, we neglected other performance-influencing factors like IO-based contention or limited concurrency due to thread pools as they could not be derived directly from the given observation data. For extracting a more fine-grained performance model, we could use a trace-based approach as described by Walter et al. (2017).

For each modeled system service, we defined a usage scenario in the Palladio model that describe the imposed load for each service independently. We opted for modeling the scenarios as open workload models with exponentially distributed inter-arrival times for simulating Poisson process arrivals as described in Section 3.2. By analyzing the average number of requests per period of time in the provided experiment data, we derived the rate of arrival for each level of load.

Our initial simulation results showed, for our baseline model, that while the minimum response times and CPU utilization were matching well, the mean and the overall variance in response times was too low. We therefore increased the variance in the baseline model by using exponentially distributed resource demands and added additional demands for the service calls. Fig. 5 shows the result of increasing the variance for the service *adjustment*.

4.3. Computational complexity evaluation approach

The analysis of algorithms approach (big-$\mathcal{O}$) used in this paper (Knuth, 1997), estimates an upper bound on the required number of computations that are executed by the evaluated algorithm, when presented with a large enough number of data inputs. The big-$\mathcal{O}$ notation consists of expressing an upper bound on the algorithm complexity growth function. More formally (Nasar, 2016),

Definition of big-$\mathcal{O}$. If $a(n)$ and $b(n)$ are two positive valued functions, we define that $a(n) = \mathcal{O}(b(n))$, if there exists a constant, K, which satisfies the condition $a(n) \leq Kb(n)$ for all, but finitely many n.

In Section 6.1, we present results of the application of the big-$\mathcal{O}$ (Knuth, 1997) notation summarized above to compare the computational efficiency of the proposed approach with some of the algorithms presented by Wert (2015).

The notation used in the computation efficiency assessment is presented in Table 3.

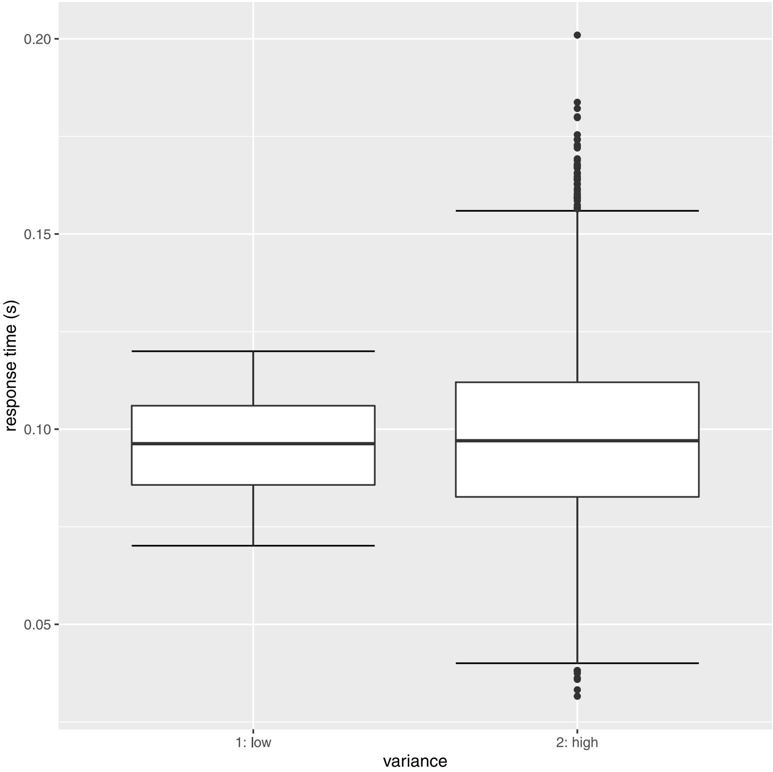


Fig. 5. Boxplots of response time for the baseline and variance increased adjustment service.

Table 3
Computational complexity notation.

Notation	Meaning
N	Number of load levels evaluated, one per load test
Φ_n	Length of response time series to be processed by *SPA* heuristics, per load test
σ_n	Length of sql response time series to be processed by *SPA* heuristics, per load test
δ_n	Length of messaging dataset to be processed by *SPA* heuristics, per load test
τ_n	Length of message tracing dataset to be processed by *SPA* heuristics, per load test
M	Number of *SPA* to be evaluated
$\mathcal{O}(s)$	Big-$\mathcal{O}$ upper bound on the algorithm complexity of function s
$\mathcal{O}(\mathcal{H}_{\mathcal{M}})$	The upper bound on the algorithm complexity of the worst-case *SPA* heuristic

4.4. Precision and recall evaluation approach

In this section, we describe the approach used to measure the precision and recall of our approach (RQ2). To do so, we used simulation. Specifically, we used simulation to inject SPA and identify the extent to which our approach was able to detect them.

The approach used to compute *SPA* precision, recall and F-measure, was to execute five simulation model experiments, one with no *SPA* injection, and four with specific *SPA* injections as follows: (1) baseline (no *SPA* injected), (2) injected Expensive database call (EDB) *SPA*, (3) injected continuously violated requirements (CVR) *SPA*, (4) injected Hiccups *SPA*, (5) injected Stifle *SPA*. For each experiment we defined a target region for *SPA* detection and we assessed true/false positive and true/false negative for the specific method where the *SPA* injection was introduced.

In this evaluation, we use the following definitions of precision, recall and the F-measure (Powers, 2011):

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{6}$$

$$\text{F-Measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{7}$$

where TP represents true positives, FP represents false positives, and FN represents false negatives. Precision assesses the impact of false positives on the approach positive detection ability, while Recall assesses the approach fraction of accurate *SPA* detection. In addition, the F-measure provides a balance between the precision and recall assessments.

4.4.1. Application Hiccup injection

For injecting application hiccups, an additional workload on the system is applied. The simulated application hiccups anti-pattern can be characterized by the following parameters:

- $hiccup_{distance}$ distribution function of time between hiccups
- $hiccup_{intensity}$ intensity of hiccup
- $hiccup_{duration}$ distribution function of active hiccup time

In our Palladio simulation model, the hiccups are mapped to an additional usage scenario that represents a background workload on the system. We decided to use an open workload where the inter-arrival time represents the $hiccup_{distance}$. The $hiccup_{intensity}$ is represented by additional load caused by the hiccup workload on the system. The $hiccup_{duration}$ is reflected by the loop count and delay between user system calls. When injecting hiccups using this approach the transient phase following the hiccup phase must not exceed $hiccup_{distance}$ to avoid overlaps.

4.4.2. Continuous Violated Requirements injection

For injecting Continuous Violated Requirements, we have to adjust service times to higher values, so that requirements cannot be met anymore. The simulated Continuous Violated Requirements anti-pattern can be characterized by the following parameter:

- $cvr_{increase}$ represented the increased service time as percentage value

This increase in service times can be mapped to a decrease of the processing rates in the Palladio resource model. Thus, we adjust the processing rate of active resources by $1/(1+cvr_{increase})$. Moreover, when adjusting the processing rate, all services deployed on the same node are affected by this *SPA*.

4.4.3. Expensive Database Call injection

The Expensive Database Call anti-pattern can be characterized by the following parameters:

- $db_{max-query-duration}$ length of tail, max duration of DB query
- $db_{long-query}$ probability of long running query

We map the expensive database call by changing the DB component: we introduce a complexity parameter for queries that when set to high imposes a load on the DB component resembling $db_{max-query-duration}$. In the calling services the complexity parameter will be characterized as high with the probability $db_{long-query}$.

4.4.4. The Stifle injection

The Stifle anti-pattern can be characterized by the following parameter:

- $stiffle_{splits}$ number of splits (n)

For injecting the stifle in the DB calling services, the call is replaced by a loop action that issues a call to the DB service. The repetition count of the loop action is set to $stiffle_{splits}$. To compensate for the additional load on the DB the resource demands in the DB service were adjusted, due to reduced query scope.

5. Approach illustration

In this section, we illustrate the application of our approach to our telecommunication system case study. The section presents response time measurements carried out in a real-world setting and it discusses the characterization of each individual component that has been analyzed using our approach.

To interpret the POSAC coordinates and to map the induced partition on the POSAC domain from measurements of load testing results to the associated *SPA*s, we define the multivariate mapping using as (x, y) coordinates the normalized distance and slope of the response time.

In the following, we instantiate the *Antipattern Characterization and Detection* approach by outlining the specific steps proposed for SPA characterization and detection in the context of our research:

1. Define a scalability requirement baseline, as introduced in Avritzer et al. (2020, 2018) that is used to provide an automated scalability assessment, i.e., the pass/fail criteria of load tests. Fig. 7 illustrates pass/fail assessment of the load tests analyzed in the case study.

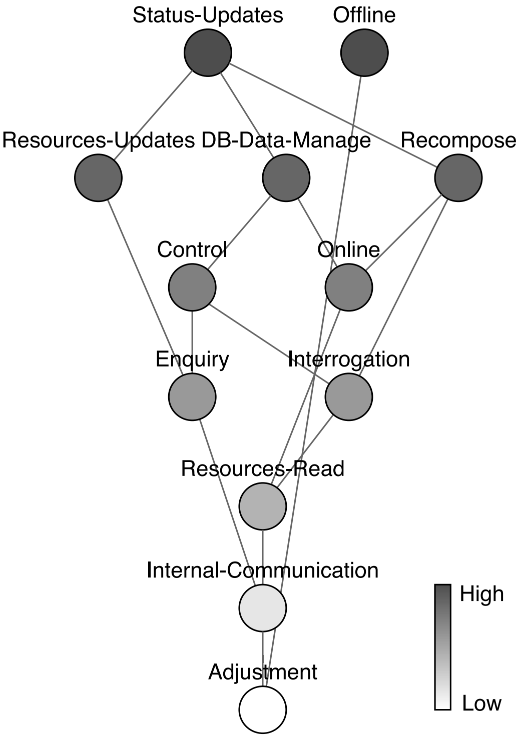


Fig. 6. Hasse Diagram using performance testing results for load = 90%.

2. Define a multivariate approach, based on the load test response time measurements that can be used to create a partial order of the evaluated services. In this paper, we have used as the multivariate system of coordinates: (1) the slope of the linear regression of response times per load, *sl*, and, (2) the normalized distance, *nd*, between the measured response time and the defined baseline requirement, as shown in Eq. (1).
3. Define a function $f(sl, nd)$ to characterize SPAs by using the defined multivariate approach. An example of the multivariate characterization of the considered *SPAs* is shown in Fig. 3.
4. Identify a set of services, $s_i(sl, nd)$, in the poset, that have the largest impact on system scalability. The set $s_i(sl, nd)$ can be found among the members of the top two rows in Fig. 6. The detailed values used to create the Hasse diagram are shown in Table 4. The members of the example set, $s_i(sl, nd)$, for the telecom application are: Status-updates, Offline, Resources-update, DB-Data-Management, and, Recompose.
5. Detect potential SPA associated with the problematic services identified in the previous step. In this paper we have introduced the use of machine learning to perform the SPA detection step.

We use the data obtained from the investigated system to illustrate our approach. The computation of the pass/fail criteria (performance/scalability requirement) uses the approach proposed by Avritzer et al. (2018), which calculates pass/fail criteria as:

$$average + 3 \cdot standard\ deviation$$

of the no-load response time measurement. Then, these values are used to calculate the two metrics used to detect the SPA: slope and normalized distance. Fig. 7 shows a plot with slope (y axis) and normalized distance (x axis) for the load 90%.

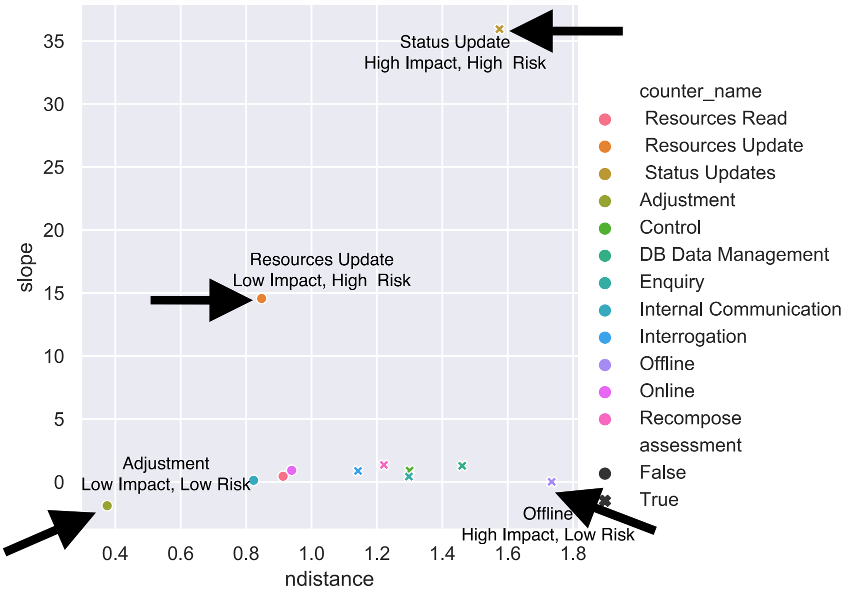


Fig. 7. System services for load = 90%.

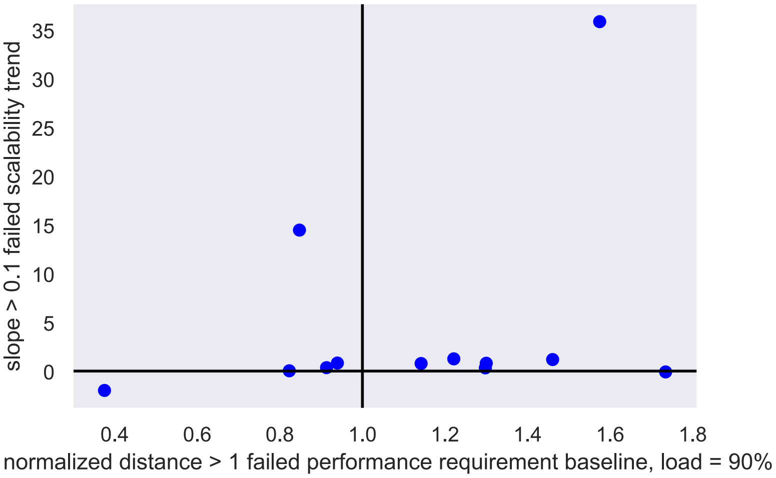


Fig. 8. Partition induced by slope and normalized distance on the investigated system services for load = 90%.

Using a risk assessment approach, we can observe that services of the investigated system can be grouped into four risk partitions (see Fig. 8 for an example of risk assessment for the 90% load in the case software):

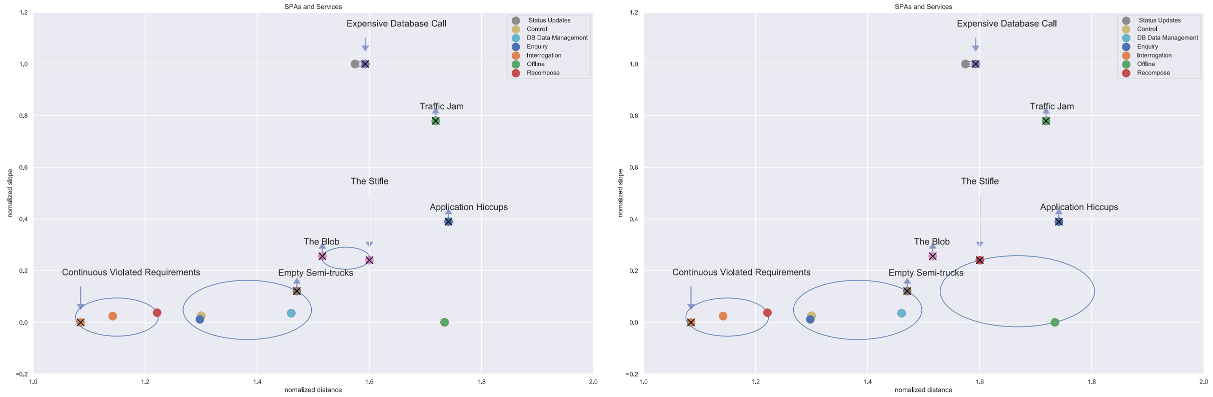
1. **Low risk/Low Impact** — Performance requirements met and no scalability degradation detected. The services that lie in the lower left quadrant have a slope less than 0.1 and a normalized distance less than 1.
2. **High risk/Low Impact** — Performance requirements met and scalability degradation detected. The services that lie in the upper left quadrant have a slope greater than 0.1 and a normalized distance less than 1.
3. **High risk/High Impact** — Performance requirements not met and scalability degradation detected. The components that lie in the upper right quadrant have a slope greater than 0.1 and normalized distance greater than 1.
4. **Low risk/High Impact** — Performance requirements not met and no scalability degradation detected. The services that lie in the lower right quadrant have slope less than 0.1 and normalized distance greater than 1.

In our previous work (Avritzer et al., 2021a), the *SPA* detection was done manually using Euclidean Distance. In this paper, we introduced the use of machine learning techniques to automate the detection and make it easier to interpret the detection results. To do so, we have evaluated two approaches: K-means (Likas et al., 2003) and K-nearest neighbors (KNN) (Mucherino et al., 2009). The motivation to use these approaches is due to their widely use by the community in clustering. Such approaches can be used to label the degree to which a given service resembles any of the considered SPA.

Figs. 9(a) and 9(b) illustrates the use of K-means and KNN for the SPA detection step, respectively. These figures only plot the services that failed the scalability assessment test, which are located to the right of the line indicating normalized distance equal to 1 (Fig. 8).

The detection using K-means follows these steps:

- The cluster centroids are initialized using the SPA's normalized distance and normalized slope values.
- K-means forms clusters of services, indicating the SPA to which it belongs. This is also illustrated in Fig. 9(a).



(a) SPA detection using K-means

(b) SPA detection using KNN

Fig. 9. K-means and KNN centroids induced by normalized slope and performance requirements normalized distance for load = 90%.

- The Euclidean distance from the new centroid to the extreme point in the cluster is calculated and represented as the radius of the cluster.
- When a new service is evaluated, K-means computes the Euclidean distance from all the cluster centroids (the SPA values) and converges to one of the local minima (Bradley and Fayyad, 1998) i.e., the new service is assigned to the cluster that is closest to the service.

The detection using KNN follows instead these steps:

- The KNN classifier is first trained with the normalized SPA coordinates generated by the analytical performance model.
- The normalized values associated with the services are classified using KNN.
- New centroids are calculated and the Euclidean distance from the new centroid to the extreme point in the classification is measured and represented as the radius of the classifier.

In the K-means implementation, the "offline" service was not assigned to an SPA cluster, while KNN was able to classify it into an *SPA*, as shown in Figs. 9(a) and 9(b).

In the KNN approach, whenever a new service is added, the *service data point* queries the k nearest neighbors and is assigned the most common value among the queried results. Due to this so-called "poll and vote" nature of the algorithm, the KNN is not prone to the outliers' problem, where a service is left out of the classification.

The time complexity of KNN is $\mathcal{O}(k \cdot M \cdot d) = \mathcal{O}(M)$, where, M is the number of *SPA* coordinates being evaluated, $k = 1$ is the number of neighbors used for voting, and $d = 2$ is the data dimension used in the multivariate assessment.

In contrast, the time complexity of K-means is $\mathcal{O}(i \cdot k \cdot M \cdot d) = \mathcal{O}(M)$, where $i = 1$ is the number of iterations, $k = 7$ is the number of clusters, M is the number of *SPA*, $d = 2$ is the number of dimensions in the data.

In summary, we found that KNN represents in our case a better choice to classify the services into possible SPAs. Thus, it is the preferred option in our approach.

6. Evaluation

In this section, we present our evaluation activity. Section 6.1 presents the results of the computational complexity, while Section 6.2 discusses the results of the accuracy our approach.

Table 4
Multivariate data for each system service for load 90%.

Service	Normalized distance	Slope
Adjustment	0.38	−1.88
Enquiry	1.30	0.43
Interrogation	1.14	0.88
Resources read	0.91	0.45
Resources updates	0.85	14.56
Status updates	1.57	35.95
Control	1.30	0.90
DB data management	1.46	1.28
Internal communication	0.82	0.13
Offline	1.73	0.01
Online	0.94	0.92
Recompose	1.22	1.35

6.1. Computational complexity

In this section, we compare the computational efficiency of the algorithmic approach presented in previous work (Wert, 2015) with the computational efficiency of the statistical approach introduced in this paper. This subsection addresses **RQ1**: *What is the computational complexity of the proposed approach?*

The approach for detection of *SPAs* is composed of two parts: (1) an offline training phase, and, (2) an online prediction phase.

In the analyzed literature, the training phase consists mostly of threshold definitions for each of the analyzed *SPAs* (Cortellessa et al., 2014; Trubiani et al., 2018; Wert, 2015; Bran, 2017), and the prediction phase employs specific algorithms defined for each of the candidate *SPAs*. Therefore, the ultimate online detection of the specified *SPAs* requires the execution of the specified prediction algorithms to detect one or more *SPA*.

In our approach, the training phase consists of *SPA* parametrization using analytical modeling to identify the partitions of the *SPA* domain that are induced by the measurement variables. This training phase is illustrated in Fig. 3, where the partitions induced by the normalized slope, and, the performance requirements normalized distance, for load 90% are shown. In the statistical approach, the online detection consists of mapping load test results into the specified *SPA* partitions, as illustrated by the blue dots in Fig. 8.

Algorithmic approach from previous research. The offline training of the heuristics presented in Wert (2015) consists of determining the parameters to be used in the *SPA* prediction heuristics, and is out of scope of this section, because the focus of this analysis is on determining the computational complexity of online prediction. The state of the art approaches presented by Wert (2015) execute heuristics to detect *SPAs*, for each load test result. Therefore, if

Table 5
SPA heuristics computational complexity summary.

SPA heuristic	$\mathcal{O}$	Justification
Application hiccups	Φ_n^2	Two nested loops
Continuous violated req.	Φ_n^2	Two nested loops
Traffic jam	Φ_n^2	Outer loop on response time series and inner loop for linear regression
The stifle	$\sigma_n \cdot \Phi_n$	Two nested loops on response time and SQL response time series
Expensive DB call	$\sigma_n \cdot \Phi_n$	Two nested loops on response time and SQL response time series
Empty semi-trucks	τ_n	Loop on message tracing dataset
The blob	δ_n	Loop on messaging dataset

there are N load tests, M *SPAs* to be evaluated, and given that the computational complexity of the worst-case heuristic is defined as $\mathcal{O}(H_M)$, the computational complexity of the state of the art algorithms is $\mathcal{O}(H_M \cdot N)$. In this section we present a summary of the results obtained from the analysis the heuristics presented by Wert (2015).

We illustrate the approach for the computational complexity evaluation of the online prediction heuristics presented by Wert (2015), by using as example, the application hiccups and continuous violated requirement *SPAs*. These *SPA* prediction heuristics are organized as two nested loops. Therefore, the computational complexity of these two anti-patterns is estimated as $\mathcal{O}(\Phi_n^2)$. The outer loop scans the response time series of length Φ_n, ordered by timestamp, and passes a starting pointer to the inner loops to execute a linear search on the series, and perform calculations to detect *SPAs*.

We present in Table 5 the justifications for the estimated worst-case computational complexity for each of the *SPA* prediction heuristics introduced by Wert (2015). Adding all the computational complexities in Table 5, we get:

$$3\Phi_n^2 + 2\sigma_n \cdot \Phi_n + \tau_n + \delta_n \tag{8}$$

Therefore, we have found that

$$\mathcal{O}(\mathcal{H}_{\mathcal{M}}) = \Phi_n^2 + \sigma_n \cdot \Phi_n + \tau_n + \delta_n \tag{9}$$

Statistical approach introduced in this paper. In the following, we evaluate the computational complexity of *SPA* detection for the approach introduced in this paper.

The number of *SPAs* to be evaluated in *offline training* is M. The approach introduced in this paper calls for analytical modeling, parametrization, and solution of the M *SPAs* for N load levels. Therefore, the computational complexity of the offline training approach introduced in this paper, can be computed using the following steps:

1. Parametrization and solution of Eq. (4), for each SPA and each evaluated load level. Therefore, the computational complexity of this step is $\mathcal{O}(M \cdot N)$.
2. Computation of normalized distance, for each *SPA*, for the evaluated load level. For example, in Fig. 3, we used the load level of 90%. Therefore, the computational complexity of this step is $\mathcal{O}(M)$.
3. The computational complexity of linear regression of one variable, slope in this case, is linear $\mathcal{O}(N)$ (Hladík and Černý, 2015). Therefore, the computational complexity of obtaining the slope of the response vs. load curve using linear regression, for each *SPA* is $\mathcal{O}(M \cdot N)$.

The computational complexity of the *online training* approach introduced in this paper, can be computed considering the following steps:

1. Compute the normalized distance for the evaluated load level. Therefore, the computational complexity of this step is $\mathcal{O}(\Phi_n \cdot N)$.
2. Compute the slope of the response time vs. load level plot using linear regression of the load testing results. Similarly to the offline computation, the computational complexity of linear regression of one variable, slope in this case, is linear $\mathcal{O}(\Phi_n \cdot N)$.
3. Implement automated *SPA* detection as described in Section 3.4 in $\mathcal{O}(M)$.

RQ1 summary: according to our evaluation, the computational complexity of the offline training approach introduced in this paper, is $\mathcal{O}(M \cdot N)$, which is $\mathcal{O}(K \cdot M) = \mathcal{O}(M)$. This is because, for *SPA* detection, we can assume N is $\mathcal{O}(K)$, where K is a small constant. The number of *SPAs* to be evaluated is likely to grow, but the number of load levels evaluated is usually a number less than 10. the computational complexity of the online training approach is $\mathcal{O}(\Phi_n \cdot N)$.

6.2. Precision and recall

We address here the research question **RQ2**: *What are the precision and recall of the proposed approach?*

We first show the results of the simulation modeling results, followed by the results of the precision and recall evaluation.

6.2.1. Simulation modeling results

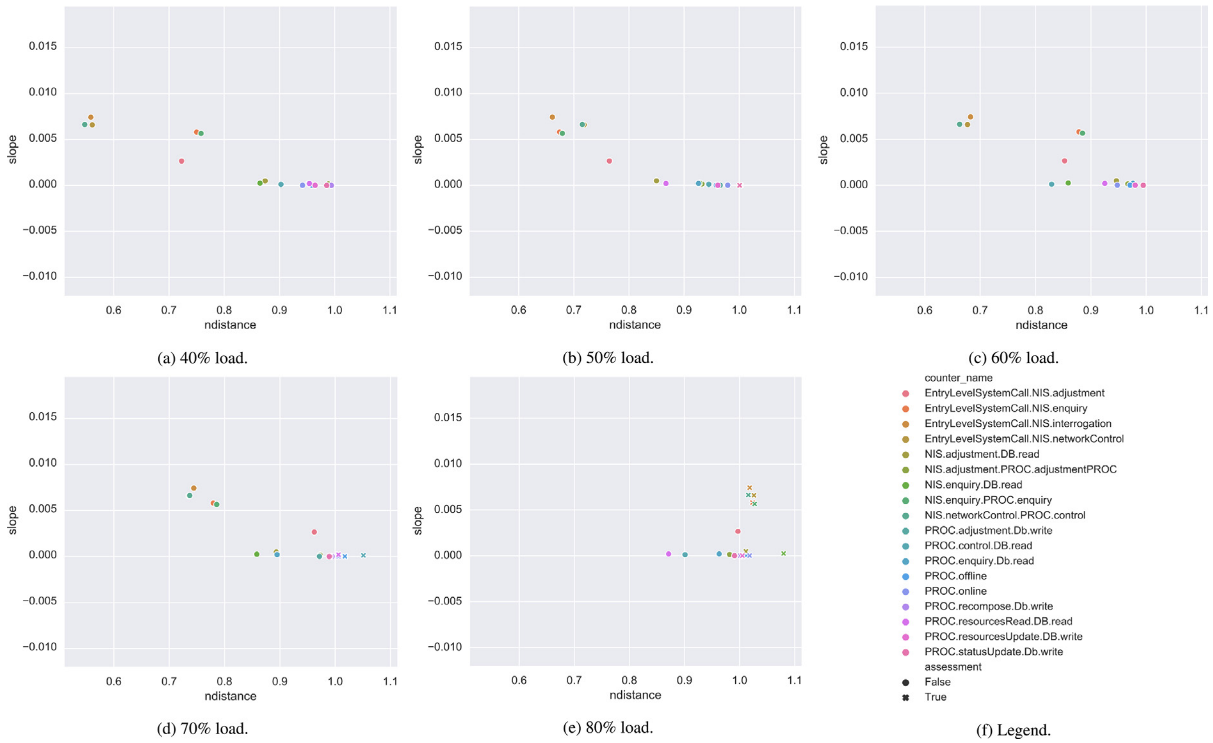
In the following, we present simulation results for the five experiments that were used to evaluate the precision and recall of detecting *SPAs*. The precision and recall evaluation is presented in Section 6.2. The precision and recall computation was performed at the 80% load. In addition, simulation results for the loads of 40%, 50%, 60%, and 70% are also shown to illustrate the impact of the load on the slope and normalized distance measures.

Baseline — We present simulation results obtained from the baseline experiment, with no SPAs, in Fig. 10. We can see from the figures that the evaluated normalized distances remains close to 1, i.e., equal to requirement, but are sensitive to offered load, and remain less than 1.15 for the whole range of offered loads. The slope behavior for the baseline case depends on the specific node/method combination.

Hiccups SPA — We present the simulation results obtained from the implementation of the Hiccups *SPA* in Fig. 11. We can see from the figures that the evaluated normalized distances are impacted by the Hiccups *SPA* with values reaching up to 1.4. The slope behavior for the Hiccups case also depends on the specific node/method combination.

Continuous Violated Requirements (CVR) SPA — We present simulation results obtained from the implementation of the CVR *SPA* in Fig. 12. We can see from the figures that the evaluated normalized distances are marginally impacted by the CVR *SPA* with values reaching up to 1.15. The slope behavior for the CVR case depends on the specific node/method combination.

The Stifle SPA — We present simulation results obtained from the implementation of the Stifle *SPA* in Fig. 13. We can see from



(a) 40% load.

(b) 50% load.

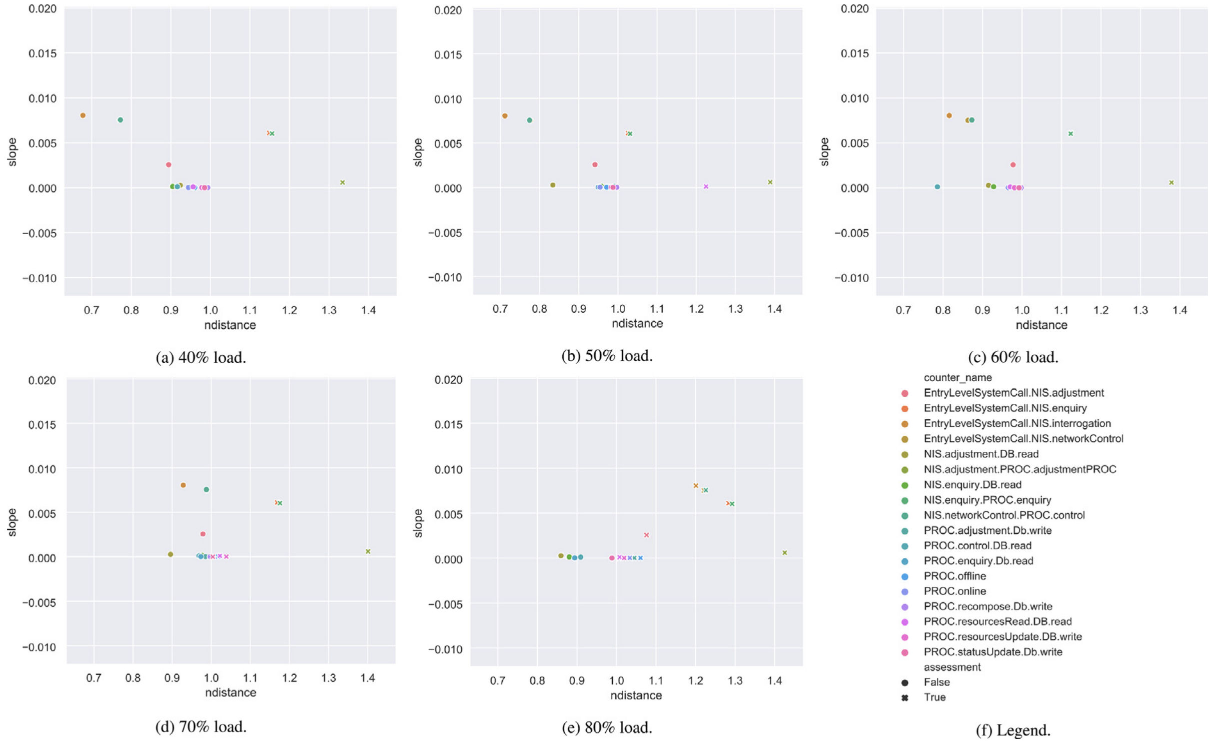
(c) 60% load.

(d) 70% load.

(e) 80% load.

(f) Legend.

Fig. 10. Simulation results for baseline case — slope and normalized distance for load from 40% to 80%.



(a) 40% load.

(b) 50% load.

(c) 60% load.

(d) 70% load.

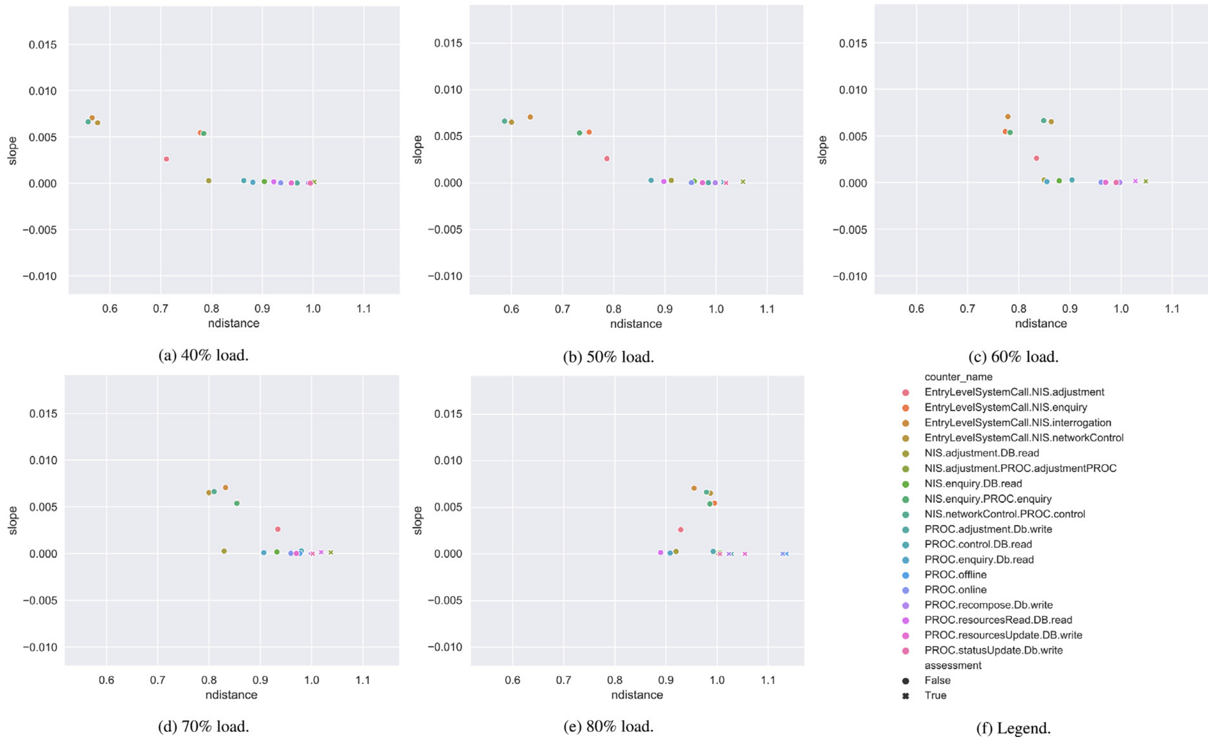
(e) 80% load.

(f) Legend.

Fig. 11. Simulation results for Hiccups SPA — slope and normalized distance for load from 40% to 80%.

the figures that the evaluated normalized distances are impacted by the Stifle *SPA* with values reaching up to 1.2. The slope behavior for the Stifle case depends on the specific node/method combination.

Expensive database Call SPA — Simulation results for the Expensive database Call *SPA* are shown in Fig. 14. We can see from the figures that the evaluated normalized distances are significantly impacted by the Expensive database Call *SPA* with values reaching up to 2.0, which is the upper range of the metric. The evaluated normalized distance is very sensitive to load. In addition, the Expensive database Call *SPA* has a major impact on the slope for several methods, as shown in the figures.



(a) 40% load.

(b) 50% load.

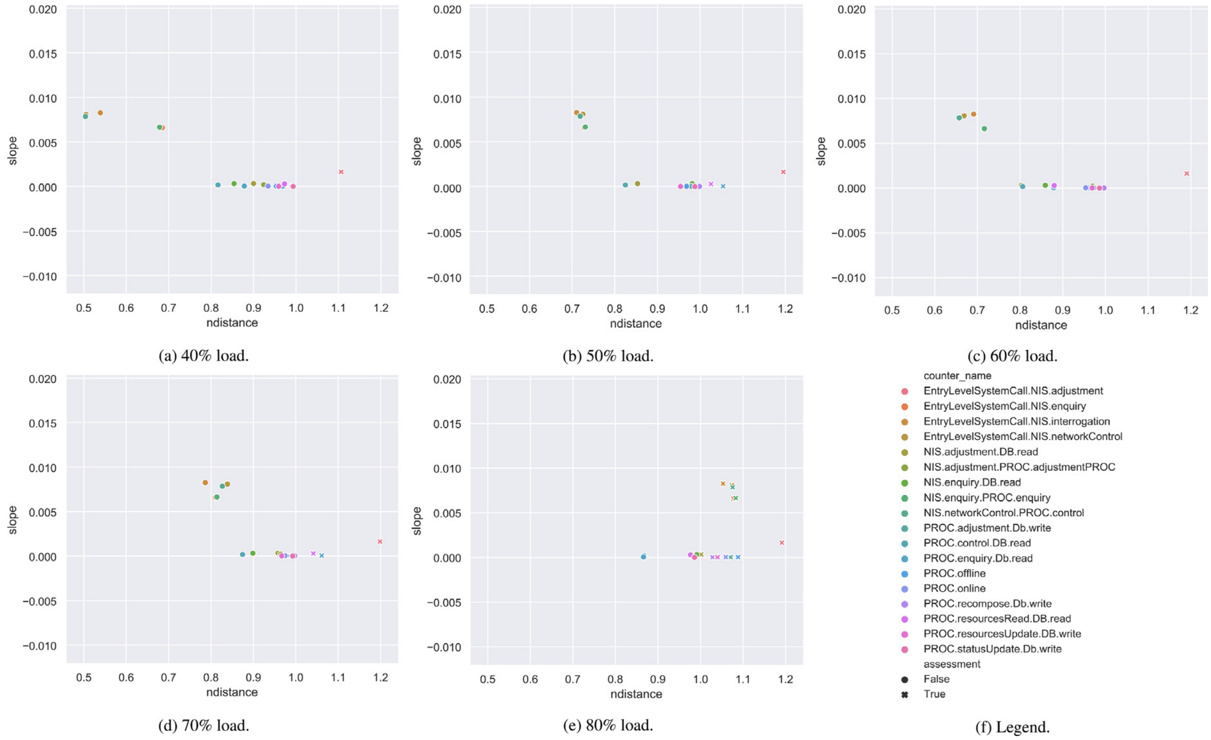
(c) 60% load.

(d) 70% load.

(e) 80% load.

(f) Legend.

Fig. 12. Simulation results for Continuous Violated Requirements SPA — slope and normalized distance for load from 40% to 80%.



(a) 40% load.

(b) 50% load.

(c) 60% load.

(d) 70% load.

(e) 80% load.

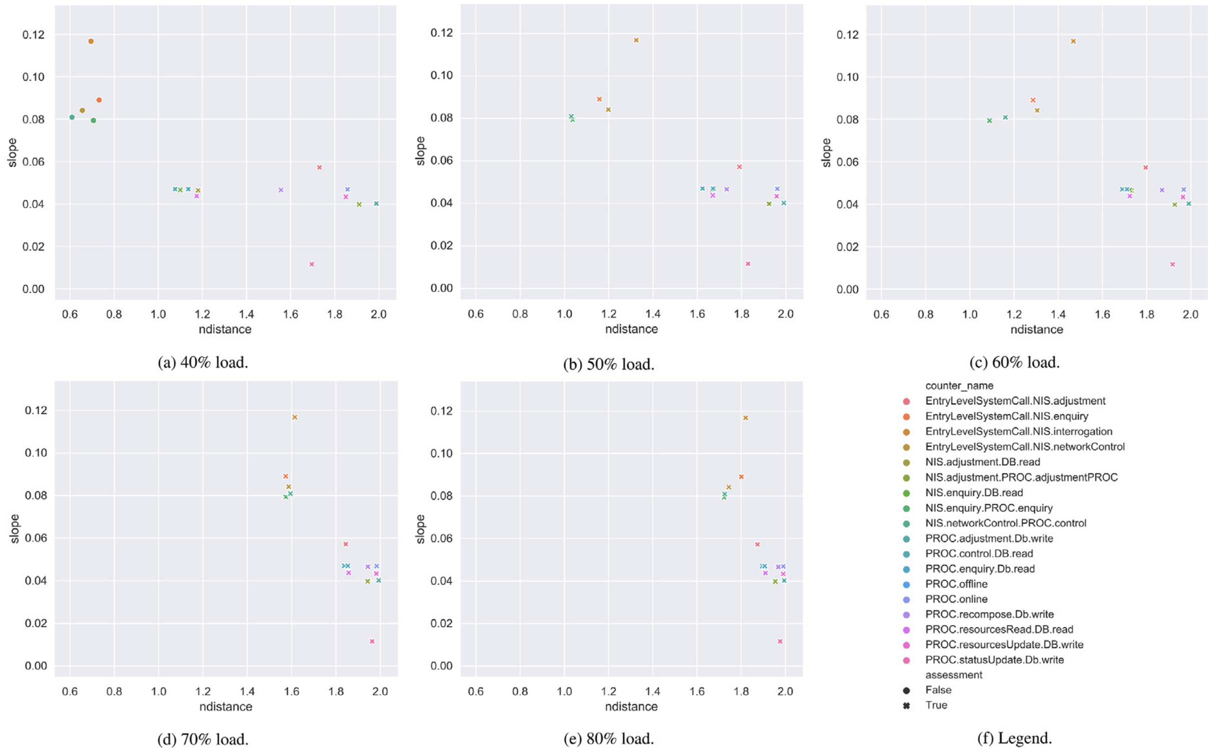
(f) Legend.

Fig. 13. Simulation results for Stifle SPA — slope and normalized distance for load from 40% to 80%.

6.2.2. *Precision and recall results*

To answer RQ2, we discuss the simulation results for the baseline case, i.e., with no injected *SPAs*, and the *SPA* injection experiments for the four *SPAs* that were characterized in the previous subsection, for several services. The naming notation used for the simulated methods is callerNode.method.calleeNode.operation. For example, "PROC.enquiry.DB.read" means that the method "enquiry" in the PROC node has called the operation "read" on the DB node.

The mapping between *SPA* injection and evaluation tables is as follows: (1) Expensive database Call results are presented in Table 6, (2) Application Hiccups results are in Table 7, (4) Stifle results are in Table 8, (5) Continuous Violated Requirements results are in Table 9.



(a) 40% load.

(b) 50% load.

(c) 60% load.

(d) 70% load.

(e) 80% load.

(f) Legend.

Fig. 14. Expensive DB Call SPA - Slope and normalized distance for load from 40% to 80%.

Table 6
Evaluation of precision and recall for the NIS.networkControl.PROC.control method. *Expensive database Call SPA* objective region objective defined by calibration as (Normalized slope near 1 and Normalized distance less than 1.6), assessment for simulation load level 80%.

Experiment	Norm. Slope/Norm. Distance	True/false positive	True/false negative	Precision	Recall	F-Measure
baseline	0.08/1.02	0/0	1/0			
EDB	1/1.73	1/0	0/0			
CVR	0.08/0.97	0/0	1/0			
hiccups	0/0.91	0/0	1/0			
stifle	0.097/1.075	0/0	1/0			
Summary		1/0	4/0	1	1	1

These tables list precision, recall, and F-measure for each one of the five methods we have evaluated for true/false positive/negative after *SPA* injection. To compute such metrics, in each table we present the baseline assessment for the evaluated method and the *SPAs* detection assessment for the four characterized *SPAs*.

The experiments were able to detect true positives for the Expensive database Call *SPA* and Continuous Violated Requirement *SPA*, for a total number of 2 true positives out of four injected *SPAs*. Overall we found 2 true positives, 2 false negatives, and 3 false positives. The overall precision of the *SPA* detection approach was found to be 0.4, while the overall recall of the approach was found to be 0.5. In addition, the overall F-measure was found to be 0.44.

The interpretation of *SPA* detection precision, as defined in Eq. (5), is that 40% of the *SPA* detected instances triggered by the approach are true instances. The interpretation of *SPA* detection recall, as defined in Eq. (6), is that 50% of the *SPAs* that should have been detected by the approach are indeed detected. The interpretation of *SPA* detection F-measure, as defined in Eq. (7), is that the balance between precision and recall is 0.44. Therefore, we can conclude that the low cost computationally efficient approach introduced in this paper can be used effectively by performance engineers, when the objective is to trade-off cost, practical applicability, and accuracy.

RQ2 summary: The simulation modeling results show that the introduced approach for *SPA* detection, is able to detect the Expensive Database Call *SPA*, as this performance anti-pattern has significant impacts on both the normalized distance and the slope measures. In addition, *SPAs* that have a minor impact on normalized distance and slope, such as the Continuous Violated Requirement *SPA*, can also be detected.
The presented approach has great potential for practical applicability at a coarse level of granularity, as it can be used to automatically differentiate between *SPAs* with significantly different normalized distance and normalized slope, as it is the case when analyzing the Expensive database Call and the Continuous Violated Requirement *SPAs*.
According to our experience, the recommended approach to apply SPA detection within CI/CDD pipelines is to: (1) detect performance degradation using normalized distance and slope, and, (2) detect a small number of *SPAs* that can be automatically classified with good precision and recall.

7. Discussion

In this section, we present a discussion of our approach by considering generalizability concerns as well as possible threats to validity.

Table 7
Evaluation of precision and recall for the NIS.enquiry.PROC.enquiry method. *Application Hiccups SPA* objective region objective defined by calibration (Normalized slope near 0.4 and Normalized distance > 1.7 and < 1.8), assessment for simulation load level = 80%.

Experiment	Norm. Slope/Norm. Distance	True/false positive	True/false negative	Precision	Recall	F-Measure
baseline	0.07/1.03	0/0	1/0			
EDB	0.98/1.72	0/1	0/0			
CVR	0.07/0.98	0/0	1/0			
hiccups	0.07/1.3	0/0	0/1			
stifle	0.08/1.08	0/0	1/0			
Summary		0/1	3/1	0	0	0

Table 8
Evaluation of precision and recall for the EntryLevelSystemCall.NIS.adjustment *Stifle SPA* objective region defined by calibration (Normalized slope < 0.3 and Normalized distance > 1.5 and < 1.8), assessment for simulation load level = 80%.

Experiment	Norm. Slope/Norm. Distance	True/false positive	True/false negative	Precision	Recall	F-Measure
baseline	0.033/0.99	0/0	1/0			
EDB	1/1.87	0/1	0/0			
CVR	0.032/0.93	0/0	1/0			
hiccups	0.032/1.08	0/0	1/0			
stifle	0.02/1.19	0/0	0/1			
Summary		0/1	3/1	0	0	0

Table 9
Evaluation of precision and recall for the PROC.recompose.Db.write *Continuous Violated Requirements SPA* objective region defined by calibration (Normalized slope < 0.2 and Normalized distance > 1.0 and < 1.2), assessment for simulation load level = 80%.

Experiment	Norm. Slope/Norm. Distance	True/false positive	True/false negative	Precision	Recall	F-Measure
baseline	0/1.00	0/0	1/0			
EDB	0.576/1.96	0/0	1/0			
CVR	0.000168/1.02	1/0	0/0			
hiccups	0.0001/1.0	0/0	1/0			
stifle	0.00019/1.02	0/1	0/0			
Summary		1/1	3/0	0.5	1	0.6666

7.1. Generalizability of our approach

In this section, we describe the generalization of the approach to other domains and we provide references to high-level guidelines for software engineers to use PPTAM in their projects. In Avritzer et al. (2021b) we presented a component diagram that basically describe the architectural concerns that were taken into account to enable generalization to further applications. Specifically, in Avritzer et al. (2021b) we have outlined a methodology for cross-domain generalization of the approach presented in this paper.

PPTAM has been designed to separate the main tool architectural concerns into individual components. In the following we summarize the main activities carried out with the goal of providing a cross-domain generalization of our approach, as presented in Avritzer et al. (2021b).

1. Configuration — this specifies the configuration setup used for load-test execution.
2. Test plan definition — this specifies the tests used in the test campaign.
3. Deployment alternatives — they are evaluated as supported by instrumented APM tools.
4. Metrics collection — this is used to specify and subsequently select the plug-in that will be activated in the test campaign.
5. User experience — this defines the dashboarding tool.

We have successfully applied this cross-domain generalization approach to support a major digital transformation initiative within a large global information services provider. This is a very different domain from telecommunications. This new project employed large grained off the shelf third party components that runs database transactions supporting a very large user community. This information services provider used a state of the art industrial approach for performance analysis that included `AppDynamics`,[2] which was used to monitor system performance flows, and `Neoload`,[3] which was used to execute performance tests at different load levels. These two tools were used to identify software bottlenecks. We generalized the PPTAM approach to improve the scalability assessment methodology used by this major digital transformation initiative. We took advantage of PPTAM component architecture to extend the state-of-the-art industrial approach used by that project (i.e., `AppDyanics` and `Neoload`). Specifically, the state of the art industrial approach was extended with the definition of operational profile, automated analysis of pass/fail tests, the integration of performance testing into the devops environment, the automated population of the scalability dashboard with devops performance testing results, and the dashboard integration using the state of the art visualization tool used by the project (PowerBI).

7.2. Threats to validity

In this section, we discuss the threats to the validity of our investigation using the categories reliability, internal, construct, and external validity described by Runeson et al. (2012).

We mitigated *reliability* threats by involving several researchers in the design and execution of our investigation. Moreover, our results were verified with the help of Ericsson experts to avoid false interpretations.

Regarding *internal* validity, the performance of the investigated system may be impacted by factors that were not present in the performance data used to illustrate our approach and to conduct part of the evaluation. This is a limitation that might have affected the related evaluation steps. We mitigated this threat by creating a PCM-based simulation model. We plan to apply our approach to additional projects.

[2] https://www.appdynamics.com.

[3] https://www.neotys.com.

The main threat to *construct* validity is that we used only one method to measure a construct. To mitigate this threat in the case study, we collected data from different sources to evaluate our approach (data triangulation).

Regarding *external* validity, although our method was developed using an analytical approach and is expected to be quite generic, the evaluation used data from just one system from one company.

8. Related work

In this section, we present an overview of the reviewed work, describing software performance testing approaches, and *SPA* detection at different stages of the software development process.

8.1. Software performance testing

A toolchain for automated software performance testing during continuous integration of Python projects is proposed in Javed et al. (2020b) where users specify testing tasks, analyze unit tests, and evaluate performance data to get feedback on the code. However, the detection of performance problems seems to refer to software bottlenecks only. The benefit of antipatterns is that they capture more complex root causes of performance issues.

An evolutionary fuzzing algorithm to generate inputs that impact on performance functions is proposed in Tizpaz-Niari et al. (2020) with the purpose of identifying a set of classes (in machine learning libraries) exhibiting different performance characteristics. An automated performance testing framework has been presented in He et al. (2020) by means of an empirical study conducted on 12 software systems, and results indicate that typically a system configuration change reflects common expectation on performance changes.

Continuous software performance assessment is pursued in Laaber (2019b) and Laaber and Leitner (2018) where the assumption is that performance problems can be detected at build time by executing microbenchmarks. This way, the detection of performance problems is anticipated before software is released and for each new commit, interestingly focusing on the temporal frequency of executing tests.

A testing approach to evaluate performance and cost requirements when porting applications to public clouds is proposed in Wang et al. (2018) where bootstrapping is adopted to get performance results of applications with unknown theoretical performance distributions.

Summary. The approaches described above complement our work. In fact we share the same goal of achieving a continuous performance assessment, but differently from the papers above, we make use of *SPA* detection.

8.2. Anomaly detection with multiple performance indices

System anomaly detection approaches like (Sahoo et al., 2003; Di Sanzo et al., 2021) have been developed to predict system failures, for example due to software aging effects and rejuvenate the system (e.g., by resetting to checkpoints or restarting affected nodes). These approaches usually use multiple system quality metrics like CPU and memory usage, number of threads and time, since such metrics are collected as inputs for machine learning unhealthy system states. As performance anti-patterns can be the root causes of system anomalies, these approaches could also be applied to potentially identify performance issues. In Di Sanzo et al. (2021) the authors also identified the slope of memory used/free and swap used/free as an important factor for their prediction model. In Peiris et al. (2014) up to 200 performance counters are collected per system to identify the cause of software anomalies. Those counters cover system information on memory and CPU, but also information about message queue length and send/received messages. Developers can visualize counters and specify thresholds on different aggregation levels to find anomalies. The approach then uses a correlation and comparative analyses to identify causes of anomalies. As result of their case study, the authors state that (i) only few performance counters were necessary to detect anomalies, and (ii) visualization and statistic aggregation are especially helpful for developers. In some cases, however, even one second aggregation was too coarse for root cause analysis.

Summary. In general, system anomaly detection usually focuses on node level and does not differentiate between services/methods. To successfully apply machine learning algorithms for anomaly detection large datasets of labeled training data are necessary.

8.3. SPA detection in real-time systems

In Avritzer et al. (2020, 2018) a methodology for the quantitative assessment of micro-service architecture deployment alternatives by automated performance testing was introduced. The methodology was developed to be integrated with CI/CDD pipelines, as it assesses scalability by incrementally incorporating results from individual tests that are run for a fixed short time-frame, as for example, 30 min.

In Wert (2015), response time requirements, measured response time, CPU, network and database utilization were used to detect *SPAs*. A set of algorithms was introduced in Wert (2015) to detect *SPAs* by analyzing the evolution over time of the monitored performance signatures. However, the detection of performance issues relies on the comparison with injected faults, some system instrumentation is needed for this scope. In contrast, in Cortellessa et al. (2014) a logic-based representation of *SPAs* is presented and a more detailed set of performance signatures is used for monitoring. Moreover, the performance signatures being monitored are compared with predefined thresholds. The set of performance signatures that are proposed to be monitored in Cortellessa et al. (2014) extends the set proposed in Wert (2015) and also includes the number of objects created/deleted, number of connections, number of messages, and other related variables. Detection algorithms are represented by the conjunction of logical formulas, but they need to be customized to specific implementation or architectural description languages, they are not directly applicable.

In Trubiani et al. (2018) and Bran (2017), the logic-based representation of performance antipatterns was applied to a large complex Java-based system. The approach presented in Cortellessa et al. (2014) was applied to the following *SPAs*: *Extensive Processing*, *Circuitous Treasure Hunt*, and *Wrong Cache*. The approach served as the basis for the implementation of a tool named *PADProf*. Logical formulas defined in Cortellessa et al. (2014) were customized to software implementation code since Java applications are targeted, and profiling data allowed to match performance indices of interest, however only three antipatterns were specified. The authors have concluded that the logic-based approach was effective to automatically detect *SPAs* in the large complex system studied. However, additional research and testing was required to generalize the application to other systems.

In Keck et al. (2016), the feasibility of injecting *SPAs* in a system under study was evaluated. It provided an example of the implementation of a proposed performance antipattern framework, where the *Ramp* and the *One Lane Bridge* implementation were demonstrated. The authors introduced an *SPA* injection framework that was designed to generate problems related to response time and memory use. However, only two simple antipatterns were implemented. This limitation, that the injected problem itself is artificial, poses a threat to the validity of the experiment.

Table 10
Comparison of the state of the art approaches on automated *SPA* detection. Considered capabilities: bottleneck detection (*bd*), proven devops support (*devops*), scalability analysis through dashboard (*dash*), SPA detection (*spa*).

Approach	Description	Capabilities			
		bd	*devops*	*dash*	*spa*
APM	Designed for production support. Examples are `AppDynamics` and `DynaTrace`. It uses detailed flow performance monitoring for applications and infrastructures in real-time. It provides integrated response time monitoring.	✓			
LoadTesting	Designed for load testing support. Examples are `Jmeter`, `NeoLoad`, `New Relic`. It allows the simulation of different load levels to uncover server side limits and estimates client side expected performance for specific performance tests.	✓			
Algorithmic approach	Algorithmic approach for *SPA* detection (Wert, 2015). Monitored variables and algorithms are defined for each *SPA* to be detected.				✓
Logic-based approach	Logic-based approach for *SPA* detection (Cortellessa et al., 2014). Monitored variables and formulas are defined for each *SPA* to be detected.				✓
PPTAM (this paper)	The approach introduced in this paper supports the automated calculation of the pass/fail criteria, the automated identification of the problematic services, and the automated analysis of the performance results to match the detected performance problems to their root causes using the specification of performance anti-patterns.	✓	✓	✓	✓

Summary. The vision of integrating automated detection of *SPAs* into CI/CDD pipelines (Avritzer et al., 2020) poses several challenges, as CI/CDD pipelines might be executed several times a day, for many components, and they have a specific performance budget for completion time.

8.4. Qualitative comparison

The state of the art on automated detection of *SPAs* (Cortellessa et al., 2014; Trubiani et al., 2018; Wert, 2015; Bran, 2017) contains algorithms that analyze detailed monitoring data to detect trends on several monitored variables (e.g., response time, CPU utilization, number of threads) and using pre-defined thresholds.

The research gap addressed in this paper is the lack of a computationally efficient approach to integrate *SPA* characterization and detection into CI/CDD pipelines. We address the aforementioned research gap by introducing a new approach to characterize and detect *SPAs* using multivariate analysis.

Table 10 compares of the state of the art approaches for the automated *SPA* detection. We consider the following capabilities: bottleneck detection, proven support for integrating performance testing with DevOps, support for scalability analysis through dashboard, and *SPA* detection. In this qualitative comparison we evaluated the following approaches:

1. Application Performance Monitoring (APM) — AppDynamics, Dynatrace;
2. Performance Load Testing Tools (LoadTesting) — Jmeter, NewRelic, NeoLoad;
3. Academia research considers mainly algorithms and logic-based representation for implementing the detection of *SPAs* (Wert, 2015; Cortellessa et al., 2014);
4. PPTAM — the approach presented in this paper.

We have concluded that the approach presented in this paper is able to complement and advance the state of industrial practice by enabling devops integration of performance testing, scalability analysis through dashboard support, and *SPA* detection at low computational cost.

Precision and Recall analysis of our *SPA* detection approach shows good results for two *SPAs*, i.e., Expensive Database Call (EDC) and Continuous Violated Requirements (CVR). This is a very valuable result, because inefficient database calls are one of the most difficult performance problems to detect and are usually detected after production deployment according to the experience of engineers at Ericsson. Our approach of integrating the automated detection of the Expensive Database Call (EDC) at low cost in a devops environment is very effective to uncover performance problems early in the software development process. In addition, as future research, we are extending the PPTAM methodology to improve the effectiveness of the detection ability of Application Hiccups and Stifle *SPAs*.

9. Conclusion and future work

In this paper, we have extended our previously proposed approach for the characterization and the detection of *SPAs* that was designed to be integrated into CI/CDD pipelines, and its implementation is computationally efficient.

We extended our previous work as follows: (1) we improved the detection step by using machine learning techniques, (2) we have evaluated the computational efficiency of the application of machine learning algorithms to the automated detection of *SPA*, and we have found that the evaluated machine learning algorithms are a good fit to this task, (3) we have also developed a simulation model of the Ericsson system to measure our approach's precision and recall. To do so, we injected four *SPAs*. We were able to evaluate the *SPA* detection approach precision and recall by comparing the simulation and analytical results. We have obtained good results for the Extensive Database Call and Continuous Violated Requirement *SPA*. As future research, we plan to experiment with several calibration methods to evaluate their impact on overall precision and recall of the *SPA* detection approach.

We have compared the computational efficiency of the proposed approach with state-of-the-art heuristics, and we have found that the approach introduced in this paper grows linearly as $\mathcal{O}(\Phi_n)$, while the computational complexity of the state-of-the-art heuristics grows as $\mathcal{O}(\Phi_n^2)$.

Moreover, because our approach characterizes each *SPA* using a multivariate approach that is based on the response time measurement, it can be efficiently implemented and deployed in industry, as response time measurements are normally logged in load testing experiments. In contrast, existing *SPA* approaches may require additional datasets for *SPA* analysis that might not be readily available.

The introduced approach can be applied to continuous development ecosystems that collect response time measurements logs, and are required to meet performance and scalability requirements. The application domains of interest are related to critical infrastructures by Avritzer et al. (2012), such as telecommunication and banking.

We are currently planning to extend this research in several ways. Our research agenda includes the following points:

- *Accuracy analysis* – While we have measured the precision and recall of our approach using a simulation-based approach, it is necessary to do the same analysis using real systems. We plan to deploy the approach in Ericsson to evaluate it through a few development sprints.
- *Integration* – We plan to integrate the introduced approach for automated *SPA* detection into automated performance testing and analysis tools, such as the one introduced at Avritzer et al. (2020).

We expect that the new approach for *SPA* detection introduced in this paper will trigger research and development on new methods and tools to automatically characterize *SPA*s using the introduced multivariate characterization approach.

Artifacts

Artifacts used in this paper are publicly available in Zenodo: https://doi.org/10.5281/zenodo.6521707.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been partially funded by eSulabSolutions, the MUR PRIN project 2017TWRCNB SEDUCE (Designing Spatially Distributed Cyber-Physical Systems under Uncertainty), the German Federal Ministry of Education and Research under grant 01IS18067D (RESPOND), the KASTEL institutional funding, and by the Baden-Württemberg Stiftung (ORCAS). We would like to thank the anonymous reviewers for their valuable and constructive comments, and all the experts at Ericsson who supported this research.

References

Avritzer, A., Britto, R., Trubiani, C., Russo, B., Janes, A., Camilli, M., van Hoorn, A., Heinrich, R., Rapp, M., Henß, J., 2021a. A multivariate characterization and detection of software performance antipatterns. In: Proceedings of the 12th ACMS/SPEC International Conference on Performance Engineering (ICPE). ACM.

Avritzer, A., Camilli, M., Janes, A., Russo, B., Jahic, J., van Hoorn, A., Britto, R., Trubiani, C., 2021b. PPTAM$^{\lambda}$: What, Where, and How of Cross-domain Scalability Assessment. In: Proceedings of International Conference on Software Architecture (ICSA), Companion Volume, pp. 62–69.

Avritzer, A., Ferme, V., Janes, A., Russo, B., van Hoorn, A., Schulz, H., Menasché, D., Rufino, V., 2020. Scalability assessment of microservice architecture deployment configurations: A domain-based approach leveraging operational profiles and load tests. J. Syst. Softw. 165, 110564.

Avritzer, A., Ferme, V., Janes, A., Russo, B., Schulz, H., van Hoorn, A., 2018. A Quantitative Approach for the Assessment of Microservice Architecture Deployment Alternatives by Automated Performance Testing. In: Proceedings of the European Conference on Software Architecture (ECSA), pp. 159–174.

Avritzer, A., Giandomenico, F.D., Remke, A.K.I., Riedl, M., 2012. Assessing dependability and resilience in critical infrastructures: challenges and opportunities. In: Wolter, K., Avritzer, A., Vieira, M., van Moorsel, A. (Eds.), Resilience Assessment and Evaluation of Computing Systems. Springer, pp. 41–63.

Avritzer, A., Weyuker, E.J., 1995. The automatic generation of load test suites and the assessment of the resulting software. IEEE Trans. Softw. Eng. 21 (9).

Avritzer, A., Weyuker, E.J., 2004. The role of modeling in the performance testing of e-commerce applications. IEEE Trans. Softw. Eng. 30 (12), 1072–1083.

Bertsekas, D., Gallager, R., 1992. Data Networks, second ed Prentice-Hall, Inc., USA.

Bradley, P.S., Fayyad, U.M., 1998. Refining initial points for k-means clustering. In: ICML, Vol. 98. Citeseer, pp. 91–99.

Bran, A., 2017. Detecting software performance anti-patterns from profiler data (B.S. thesis).

Brown, W., Malveau, R., McCormick, H., Mowbray, T., 1998. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons.

Camilli, M., Russo, B., 2022. Modeling performance of microservices systems with growth theory. Empir. Softw. Eng. 27 (39), 1–44.

Chen, J., Shang, W., Shihab, E., 2020. Perfjit: Test-level just-in-time prediction for performance regression introducing commits. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/TSE.2020.3023955.

Cortellessa, V., Di Marco, A., Trubiani, C., 2014. An approach for modeling and detecting software performance antipatterns based on first-order logics. Softw. Syst. Modeling 13 (1), 391–432.

Daniel, M.A.A., Almeida, V.A., Dowdy, L.W., Dowdy, L., et al., 2004. Performance by Design: Computer Capacity Planning by Example. Prentice Hall Professional.

Di Sanzo, P., Avresky, D.R., Pellegrini, A., 2021. Autonomic rejuvenation of cloud applications as a countermeasure to software anomalies. Softw. - Pract. Exp. 51 (1), 46–71.

He, H., Jia, Z., Li, S., Xu, E., Yu, T., Yu, Y., Wang, J., Liao, X., 2020. CP-detector: using configuration-related performance properties to expose performance bugs. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 623–634.

Heger, C., van Hoorn, A., Mann, M., Okanovic, D., 2017. Application Performance Management: State of the Art and Challenges for the Future. In: Proceedings of the 2017 ACM/SPEC International Conference on Performance Engineering (ICPE), pp. 429–432.

Heinrich, R., Werle, D., Klare, H., Reussner, R., Kramer, M., Becker, S., Happe, J., Koziolek, H., Krogmann, K., 2018. The palladio-bench for modeling and simulating software architectures. In: Proceedings of the International Conference on Software Engineering: Companion Proceeedings. ACM, pp. 37–40.

Hladík, M., Černý, M., 2015. Total least squares and Chebyshev norm. Procedia Comput. Sci. 51, 1791–1800. http://dx.doi.org/10.1016/j.procs.2015.05.393, International Conference On Computational Science, ICCS 2015.

Javed, O., Dawes, J.H., Han, M., Franzoni, G., Pfeiffer, A., Reger, G., Binder, W., 2020a. PerfCI: A toolchain for automated performance testing during continuous integration of python projects. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1344–1348.

Javed, O., Dawes, J.H., Han, M., Franzoni, G., Pfeiffer, A., Reger, G., Binder, W., 2020b. PerfCI: a toolchain for automated performance testing during continuous integration of Python projects. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 1344–1348.

Keck, P., van Hoorn, A., Okanović, D.s., Pitakrat, T., Düllmann, T.F., 2016. Antipattern-Based Problem Injection for Assessing Performance and Reliability Evaluation Techniques. In: Proceedings of the International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 64–70.

Kleinrock, L., Collection, K.M.R., 1974. Queueing Systems, Volume I. In: A Wiley-Interscience publication, Wiley.

Knuth, D.E., 1997. The Art of Computer Programming, Volume 1, third ed In: Fundamental Algorithms, Addison Wesley Longman Publishing Co., Inc., USA.

Laaber, C., 2019a. Continuous software performance assessment: Detecting performance problems of software libraries on every build. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2019, Association for Computing Machinery, New York, NY, USA, pp. 410–414. http://dx.doi.org/10.1145/3293882.3338982.

Laaber, C., 2019b. Continuous software performance assessment: detecting performance problems of software libraries on every build. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp. 410–414.

Laaber, C., Leitner, P., 2018. An evaluation of open-source software microbenchmark suites for continuous performance assessment. In: Proceedings of the International Conference on Mining Software Repositories (MSR), pp. 119–130.

Liao, L., Chen, J., Li, H., Zeng, Y., Shang, W., Sporea, C., Toma, A., Sajedi, S., 2021. Locating performance regression root causes in the field operations of web-based systems: An experience report. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/TSE.2021.3131529.

Likas, A., Vlassis, N., Verbeek, J.J., 2003. The global k-means clustering algorithm. Pattern Recognit. 36 (2), 451–461.

Maggino, F., 2014. Guttman scale. In: Michalos, A.C. (Ed.), Encyclopedia of Quality of Life and Well-Being Research. Springer Netherlands, Dordrecht, pp. 2626–2630.

Microsoft, 2019. Performance tuning a distributed application. https://docs.microsoft.com/en-us/azure/architecture/performance/, (accessed on June 25, 2020).

Mucherino, A., Papajorgji, P.J., Pardalos, P.M., 2009. K-nearest neighbor classification. In: Data Mining in Agriculture. Springer, pp. 83–106.

Nasar, A., 2016. The history of algorithmic complexity. Math. Enthus. 13 (3), 217–242.

Parsons, T., Murphy, J., et al., 2008. Detecting performance antipatterns in component based enterprise systems. J. Object Technol. 7 (3), 55–91.

Peiris, M., Hill, J.H., Thelin, J., Bykov, S., Kliot, G., Konig, C., 2014. Pad: Performance anomaly detection in multi-server distributed systems. In: 2014 IEEE 7th International Conference on Cloud Computing. IEEE, pp. 769–776.

Pinciroli, R., Smith, C.U., Trubiani, C., 2021. QN-based Modeling and Analysis of Software Performance Antipatterns for Cyber-Physical Systems. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering, pp. 93–104.

Powers, D.M.W., 2011. Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. J. Mach. Learn. Technol. 2 (1), 37–63.

Reussner, R.H., Becker, S., Happe, J., Heinrich, R., Koziolek, A., Koziolek, H., Kramer, M., Krogmann, K., 2016. Modeling and Simulating Software Architectures – The Palladio Approach. MIT Press, ISBN: 9780262034760.

Runeson, P., Höst, M., Rainer, A., Regnell, B., 2012. Case Study Research in Software Engineering: Guidelines and Examples. John Wiley & Sons.

Sahoo, R.K., Oliner, A.J., Rish, I., Gupta, M., Moreira, J.E., Ma, S., Vilalta, R., Sivasubramaniam, A., 2003. Critical event prediction for proactive management in large-scale computer clusters. In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 426–435.

Shye, S., 1985. Multiple Scaling: The Theory and Application of Partial Order Scalogram Analysis. North-Holland, Amsterdam.

Shye, S., 2009. Partial order scalogram analysis by coordinates (POSAC) as a facet theory measurement procedure: how to do posac in four simple steps pp. 295–310.

Shye, S., 2014. Systemic quality of life model (SQOL). pp. 6569–6575.

Skiena, S., 1990. Hasse diagrams. In: §5.4.2 in Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Reading, MA: Addison-Wesley, p. 163, 169-170 and 206-208.

Smith, C.U., 2020. Software performance antipatterns in cyber-physical systems. In: Proceedings Od the International Conference on Performance Engineering (ICPE). ACM, pp. 173–180.

Smith, C.U., Williams, L.G., 2000. Software performance antipatterns. In: Proceedings of the 2nd International Workshop on Software and Performance. In: WOSP '00, Association for Computing Machinery, New York, NY, USA pp. 127–136. http://dx.doi.org/10.1145/350391.350420.

Smith, C.U., Williams, L.G., 2012. Software performance antipatterns for identifying and correcting performance problems. In: International Computer Measurement Group Conference.

Spinner, S., Casale, G., Zhu, X., Kounev, S., 2014. Librede: A library for resource demand estimation. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, pp. 227–228.

Tizpaz-Niari, S., Cerný, P., Trivedi, A., 2020. Detecting and understanding real-world differential performance bugs in machine learning libraries. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp. 189–199.

Trubiani, C., Bran, A., van Hoorn, A., Avritzer, A., Knoche, H., 2018. Exploiting load testing and profiling for Performance Antipattern Detection. Inf. Softw. Technol. 95, 329–345.

Trubiani, C., Koziolek, A., Cortellessa, V., Reussner, R.H., 2014. Guilt-based handling of software performance antipatterns in palladio architectural models. J. Syst. Softw. 95, 141–165.

Walter, J., Stier, C., Koziolek, H., Kounev, S., 2017. An expandable extraction framework for architectural performance models. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, pp. 165–170.

Wang, W., Tian, N., Huang, S., He, S., Srivastava, A., Soffa, M.L., Pollock, L., 2018. Testing cloud applications under cloud-uncertainty performance effects. In: Proceedings of the International Conference on Software Testing, Verification and Validation (ICST), pp.81–92.

Wert, A., 2015. Performance Problem Diagnosis by Systematic Experimentation. (Ph.D. thesis). KIT.

Wert, A., Happe, J., Happe, L., 2013. Supporting swift reaction: automatically uncovering performance problems by systematic experiments. In: International Conference on Software Engineering (ICSE). pp. 552–561.