Software and Systems Modeling (2024) 23:1003–1023
https://doi.org/10.1007/s10270-023-01137-x

REGULAR PAPER

Modeling more software performance antipatterns in cyber-physical systems

Riccardo Pinciroli[1] · Connie U. Smith[2] · Catia Trubiani[1]

Received: 4 November 2022 / Revised: 29 September 2023 / Accepted: 18 October 2023 / Published online: 20 December 2023

Abstract
The design of cyber-physical systems (CPS) is challenging due to the heterogeneity of software and hardware components that operate in uncertain environments (e.g., fluctuating workloads), hence they are prone to performance issues. Software performance antipatterns could be a key means to tackle this challenge since they recognize design problems that may lead to unacceptable system performance. This manuscript focuses on modeling and analyzing a variegate set of software performance antipatterns with the goal of quantifying their performance impact on CPS. Starting from the specification of eight software performance antipatterns, we build a baseline queuing network performance model that is properly extended to account for the corresponding bad practices. The approach is applied to a CPS consisting of a network of sensors and experimental results show that performance degradation can be traced back to software performance antipatterns. Sensitivity analysis investigates the peculiar characteristics of antipatterns, such as the frequency of checking the status of resources, that provides quantitative information to software designers to help them identify potential performance problems and their root causes. Quantifying the performance impact of antipatterns on CPS paves the way for future work enabling the automated refactoring of systems to remove these bad practices.

Keywords Software modeling · Software performance antipatterns · Model-based performance analysis · Cyber-physical systems

1 Introduction

In the software development process, there is high interest in the early validation of requirements, especially for performance-related characteristics that (recently) are considered as the new system correctness [1]. This is further motivated by the cost of fixing errors that has been demonstrated to escalate exponentially as the project matures through its life cycle [2]. Predicting performance issues early in software development is indeed valuable to avoid fixes to consolidated software artifacts [3]. Our work deals with the open problem of understanding the reason for performance degradation when evaluating different design choices (e.g., the frequency of checking the status of system resources) on the basis of their impact on the system performance [4].

Software performance engineering (SPE) [5] aims to produce performance models early in development. In recent years, several approaches have been successfully developed to automate the modeling and analysis of software performance [6, 7], and optimization techniques [8]. However, the problem of interpreting model-based performance analysis results is still critical, especially when considering modern and complex application domains, such as cyber-physical systems (CPS), where the heterogeneity of software and hardware components triggers new challenges for traditional SPE approaches. CPS share with reactive systems the typical characteristics of handling the occurrence of events [9] that determine the integration of computing and communication with the monitoring and/or control of entities in the physical world.

Communicated by Robert Pettit.

✉ Catia Trubiani
catia.trubiani@gssi.it

Riccardo Pinciroli
riccardo.pinciroli@gssi.it

Connie U. Smith
http://www.speed.com

[1] Gran Sasso Science Institute, L'Aquila, Italy

[2] Performance Engineering Services, L&S Computer Technology, Inc., Austin, TX, USA

Our research focuses on understanding the performance degradation in CPS. We present an approach that can be applied to any reactive software system that may include bad practices leading to performance problems. There exist many challenges in modeling, specifying, and verifying reactive systems [10], mainly due to the interaction of agents with their computing environment, hence we are interested in capturing those system events that may become root causes of performance issues. Our goal is to identify the system performance problems by localizing the software components that may cause such problems. To achieve this objective, we make use of software performance antipatterns [11], recently customized for CPS [12] and detected in open-source projects [13]. The rationale behind this choice is that software performance antipatterns include the description of (i) design problems that lead to performance issues and (ii) solutions to the problems that improve performance. Consider as an illustrative example the *Blob* performance antipattern. It occurs when a single component monopolizes the computation managing most of the work and becomes a system bottleneck. To solve this, it is necessary to improve the management of the system workload by delegating work to surrounding software components in a distributed fashion.

Note that the specification of software performance antipatterns in [11] is intentionally generic and not constrained to application domains, making them flexible to capture bad practices in different contexts, e.g., information systems. In this paper we focus on modeling antipatterns to address the open problem of investigating which bad practices find a counterpart for performance issues that may occur in CPS. In our previous work [14], we focused on three software performance antipatterns defined in [12]. This manuscript moves a step forward in the direction of extending the types of bad practices, to provide software developers an understanding of a larger set of problems. To this end, we investigate the specification of performance antipatterns in [11], and we provide models of antipatterns along with analysis results that give evidence of the large impact these antipatterns have on the performance of CPS. Specifically, we consider the following five additional software performance antipatterns: (i) *Circuitous Treasure Hunt*, i.e., requests that must look in multiple places to find the needed information; (ii) *One-Lane Bridge*, i.e., requests running in parallel that are temporarily restricted to execute sequentially; (iii) *More is Less*, i.e., too many processes competing for computing resources; (iv) *The Ramp*, i.e., the amount of resources required increasing over time; (v) *Traffic Jam*, i.e., a burst of requests overloading computing resources for a period of time. These antipatterns are modeled with queuing network (QN) performance models [15], since this formalism is well-established in the SPE community and widely used to analyze modern real-world applications, e.g., automotive [16], unmanned aerial vehicle [17], IoT-enhanced spaces [18], or Industry 4.0 [19].

The objective of our research is to support software developers in understanding root causes of performance degradation in CPS. To this end, we propose a model-based approach that includes a variegate set of performance models for software antipatterns. The simulation-based analysis of these models provides evidence on the impact of antipatterns, eventually resulting in fluctuations and bottleneck switches in the system performance. A network of sensors is used to assess the usefulness of the proposed QN models in analyzing performance problems that emerge in software development. The main contributions of this manuscript can be summarized as follows:

- the specification of QN models expressing the peculiarities of five additional software performance antipatterns that are newly applied to the CPS domain;
- the injection of eight software performance antipatterns in a real-world system, and empirical evidence on their benefit for interpreting the performance issues of CPS;
- the modeling of antipatterns using QN in the context of CPS advances the software modeling field since these models can be used to analyze software performance quantitatively.

The rest of the manuscript is organized as follows. Section 2 explains the connection between software performance antipatterns and real-world CPS. Section 3 describes QNs that model the antipatterns and shows their impact on the system performance. Section 4 assesses software performance antipatterns in a network of continuously-monitored sensors, and experimental results demonstrate the performance impact of antipatterns on the system response time while varying the peculiar characteristics of these bad practices. Threats to validity are argued in Sect. 5. Section 6 briefly reviews related work. Concluding remarks and future work are outlined in Sect. 7. All models, experiments, and replication data are publicly available [20].

2 Antipatterns in CPS

The demand for developers with domain expertise as well as expertise with the new CPS technology exceeds the talent pool. This combination of new technology and lack of expertise dramatically increases the risk of performance (and other) failures. Previous work contrasted characteristics of CPS of the past, particularly Real Time Embedded Systems (RTES), with today's CPS to illustrate why CPS performance problems are now occurring much more frequently

[12]. CPS performance antipatterns aim to solve these performance challenges in today's CPS.

By today's CPS we mean systems showing distinguishing characteristics, as expressed in [12]. For instance: (i) the dramatic increase in control variables and automation of tasks is confirmed in [21] by means of an open-source drone application; (ii) the usage of complex and ambitious functions are acknowledged by [22]; (iii) the need of managing large numbers of processes that require communication and coordination is studied in [23] through a smart traffic application; (iv) the adoption of built-in functionalities are investigated in [24] in the context of a smart city scenario for safe crowd monitoring and control; (v) the need for dynamic scheduling is analyzed in [25, 26].

In this section, we briefly describe the connection between software performance antipatterns and a real-world CPS example, i.e., the Smart Parking System [27]. However, we also extend the discussion to generic real-world CPS and we consider the characteristics of these systems in connection with performance antipatterns specification.

The Smart Parking System [27] consists of a server orchestrating scanning and parking cars that collaborate to identify empty parking spots. We are interested in spotting probable issues in the response time required to provide information (possibly image data retrieved by scanning cars) to parking cars looking for an empty spot. This scenario is indicative of generic CPS where a set of physical entities need to interact to acquire a resource of interest. There are several performance antipatterns that could occur, and we discuss these illustrative examples next.

For instance, the server may poll scanning cars to check if new image data is available (i.e., *Are We There Yet?* performance antipattern), and the polling interval may cause performance problems. If the time interval is too small, then the car is continuously interrupted, the server is busy with overhead rather than real work, and the overall system performance may suffer. If the time interval is too long, the images may become stale before the server acts on it. CPS may include physical entities that are continuously interrupted when doing their job, hence performance delays occur.

The *Is Everything OK?* performance antipattern occurs if the server (too) frequently contacts all cars to confirm that their cameras are functioning correctly. This delays the retrieval of images and cars may have an unexpected delay in receiving parking results. As opposite, if cars initiate communication of camera malfunctions, then fewer messages are exchanged and this may be beneficial for the overall system performance. The frequent check on the status of resources in CPS may generate performance overhead.

The *Where Was I?* performance antipattern occurs when the server does not remember previous parking results and re-starts the image analysis. If instead the server remembers "objects of interest" such as where parking spots were available, it could first make a quick check to see if it is still available. If the server forgets previous results, then it wastes considerable time recalculating and the overall system performance suffers. Traditional CPS preserved state information to prevent *Where Was I?* performance degradation. Even then, restoring the state on startup has led to excessively long boot times and thus a failure to meet performance requirements.

The *Circuitous Treasure Hunt* performance antipattern is related to the database design for frequent access. A typical example is when the number of available parking spaces in an area is frequently needed. The Circuitous Treasure Hunt occurs when the database design requires a "count" operation of the raw data to calculate the number of available spaces. It is even worse if images must be scanned to determine availability of spaces to be tallied. If instead the database stores the number of spaces available in each area, and updates that number as spaces are taken or left the performance is greatly improved when that number is requested. Traditional CPS seldom used database technology, but newer technology has led to increased use of databases. Accessing resources in CPS is expensive, better to limit the number of accesses.

The *One-Lane Bridge* antipattern is also related to database design. When all cars concurrently send images to the server, and the server appends the images to the end of the database, all processes are competing to write to the same location. However, only one process may execute at a time, and the One-Lane Bridge limits parallelism. The performance can be improved by first capturing the images to different storage locations, then updating the database with that location. This improves concurrency and shortens the time to "cross the bridge" by only updating a location in the single threaded section of code. Limiting concurrency in CPS at a specific operational point may degrade the system functionality.

The *More is Less* antipattern happens when too many processes attempt to do computation in parallel, and the associated overhead and contention delays negatively impact performance. This could occur in "smart" parking when it is necessary to update maps of the parking areas. If all cars request updates of all maps in parallel, the system will be overloaded with too many parallel requests. If instead cars request a few map updates at a time, perhaps by prioritizing the update requests by the current location of the car, the overall performance improves. Overwhelming concurrency in CPS may generate a system bottleneck in managing an extraordinary number of requests.

The Ramp antipattern can occur because the information on empty parking spots continuously evolves as the system is used. Thus the complexity and execution time of requests processed by the server increase with system operations. Data structures and algorithms tailored to the maximum operational size could help to avoid an increas-

ing and unpredictable processing time. The adoption of continuously increasing data structures in CPS should be discouraged since their management becomes too costly.

An example of the *Traffic Jam* antipattern occurs when the system must refresh the database with the status of all parking spots, either at start-up or after an outage. If the system acquires all parking spot status for all areas at the same time, the system will be overloaded for a long period of time and will be unable to respond to parking spot requests. If a phased refresh can be implemented, perhaps prioritizing the most important areas or the most likely to be needed first, the load will be spread, performance and availability will improve. Physical entities in CPS should act asynchronously to avoid a burst of large requests. Processing for initialization and for re-boot should spread the load so system availability is preserved.

3 Our approach

CPS include real-time concerns and requirements that are critical. To this end, we report the *system response time*, i.e., the average time (i.e., the sum of waiting and service time) taken by requests to be processed. This performance index represents a key factor in our analysis since it provides knowledge on the timeliness of these systems, i.e., the ability to produce the expected result by a specific deadline. Designers can compare the model-based performance analysis results with the stated requirements, thus assessing real-time concerns of CPS. We also analyze the *utilization of resources* (i.e., the percentage of time that each resource is busy) to identify which resource is the bottleneck of the system and degrades the system performance.

It is worth remarking that system response time and utilization of resources are used to monitor the system performance. However, the specification of performance antipatterns also includes bad practices that are poor design choices leading to system misbehavior. As stated in [28], predictability is a key factor of real-time systems, i.e., the timing behavior of a system has to satisfy its specifications. Our work contributes in the direction of introducing design techniques that anticipate operational uncertainties. In fact, the specification of antipatterns plays the crucial role of expressing bad practices that may contribute to errors in the timing behavior of systems.

In this section, we discuss how to model performance antipatterns [11, 12] using QNs [15]. It is worth remarking that QNs represent an abstraction of the software system under analysis. Our approach relies on strategies that have been defined in the literature to derive performance models from the software design specification [29, 30]. Hereafter, we describe our baseline QN model that is represented by a simple and abstract system with two resources, namely *Resource1* and *Resource2*. Software performance antipatterns are described by decorating the baseline model and we evaluate their impact on the system performance using Java Modelling Tools (JMT) [31] to simulate the proposed QN models. All simulations in this section stop when analyzed metrics are observed with 99% confidence interval and 3% maximum relative error with the exception of those showing the evolution of performance indices over time, see Sects. 3.8 and 3.9. Simulations are stopped despite the relative error value after 1 M samples are collected, i.e., the maximum number of analyzed samples is set to 1 M.

3.1 Baseline

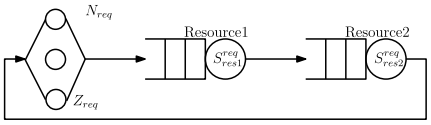
3.1.1 Modeling

The effect of performance antipatterns on a system is investigated using a single-class QN model with a delay station and two queuing centers (i.e., *Resource1* and *Resource2*), as shown in Fig. 1a. Specifically, we consider a batch (closed) system whose workload is defined by the number of requests (N_{req}) and a think time ($Z_{req} = 0$ s). This is a common assumption when modeling CPS (e.g., [19]). It does not limit the generality of our approach since equivalent closed QNs can be derived from open ones [32]. All requests are first served by *Resource1* and later processed by *Resource2* with two exponential distributions whose average service time are S^{req}_{res1} and S^{req}_{res2}, respectively. A *First Come First Served* queuing strategy is used in both stations.

The choice of using only two queuing centers as the baseline model is to simplify the modeling of software performance antipatterns and demonstrate their impact. The choice of input model parameters also follows this objective, and the numerical values are reported in Fig. 1b. We consider $N_{req} = 10$ requests, $S^{req}_{res1} = 0.02$, and $S^{req}_{res2} = 0.04$, both service times are expressed in seconds (*sec*). These values, as well as others considered in Sect. 3, are consistent with real-world CPS tasks [33] whose completion deadline ranges from a few milliseconds to several seconds. Figure 1b shows also the performance indices (i.e., R^{req}_{sys}, U_{res1}, and U_{res2}) obtained analytically using the mean value analysis (MVA) [15].

3.1.2 Analysis

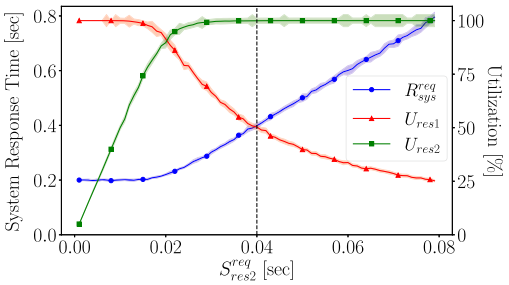
Figure 1c depicts the considered performance indices: the system response time of requests (R^{req}_{sys}, blue line with circular dots, left y-axis), the utilization of *Resource1* (U_{res1}, red line with triangular dots, right y-axis), and the utilization of *Resource2* (U_{res2}, green line with squared dots, right y-axis). All the performance indices are depicted with their 99% confidence interval (i.e., shaded areas). These indices are depicted against the service time of *Resource2*, S^{req}_{res2}, that



(a) QN model

Parameters		Indices	
Parameter	Value	Parameter	Value
N_{req}	10	R^{req}_{sys}	0.4
S^{req}_{res1}	0.02	U_{res1}	50%
S^{req}_{res2}	0.04	U_{res2}	100%
Z_{req}	0.0		
# CPUs	1		

(b) Input parameters and analytical performance indices



(c) Performance metrics

Fig. 1 Baseline—performance modeling and evaluation

varies from 0 to 0.08, while the service time of *Resource1*, S^{req}_{res1}, is set to 0.02. *Resource1* is the system bottleneck when $S^{req}_{res2} < 0.02$, otherwise the system capacity is limited by *Resource2*, see the utilization curves crossing in Fig. 1c. Performance indices observed for the baseline service time of *Resource2*, $S^{req}_{res2} = 0.04$, are indicated by a vertical dashed line, i.e., $R^{req}_{sys} = 0.4$, $U_{res1} = 50\%$, and $U_{res2} = 100\%$.

3.2 Are We There Yet?

3.2.1 Modeling

Requests that need computational power to check the occurrence of an event are modeled by defining a new request class (i.e., *Check*), as in Fig. 2a. Specifically, N_{chk} requests of the new *Check* class are initialized in the system (i.e., one for each event that is monitored). The time spent by a *Check* request in the delay station and *Resource1* is exponentially distributed with average Z_{chk} and S_{chk}, respectively. The overhead for checking is significant and requires many resources [12], $S_{chk} \simeq S^{req}_{res1}$. For the sake of simplicity and without loss of generality, we assume that *Resource2* is not affected by this software performance antipattern. Therefore, *Check* requests do not visit *Resource2* and they are routed back to the delay station after being processed by *Resource1*.

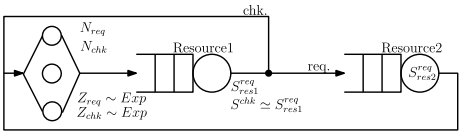
3.2.2 Analysis

Numerical values used to analyze this antipattern are shown in Fig. 2b. We assume 250 requests are sent to *Resource1* to check if an event has occurred, i.e., $N_{chk} = 250$. Since checking for the occurrence of an event is expensive, the service time of *Check* requests is set to half the time required to execute default requests (i.e., $S_{chk} = 0.01$ s). We analyze the system with $0 < Z_{chk} < 10$ s to evaluate the effect of checking frequency. Results are depicted in Fig. 2c whose x-axis is inverted to highlight the effect of more frequent checking requests (i.e., a shorter think time). As expected, the system response time increases when the event occurrence is checked more frequently (i.e., for small Z_{chk} values). Small Z_{chk} values increase the usage of *Resource1*, reduce the usage of *Resource2*, and make the request execution slower. The baseline system response time (i.e., $R^{req}_{sys} = 0.4$) is observed for $Z_{chk} > 5$ s. This is depicted by the dashed blue line and the blue arrow pointing towards the direction where R^{req}_{sys} is not longer than the baseline. For the sake of clarity, Z_{chk} values that allow observing the baseline utilization of resources are not depicted in Fig. 2c. Summarizing, checking more frequently the occurrence of an event generates performance overhead that increases resource usage, prevents other resources from accomplishing their work, most likely switches the system bottleneck, and slows down the overall computation.

3.3 Is Everything OK?

3.3.1 Modeling

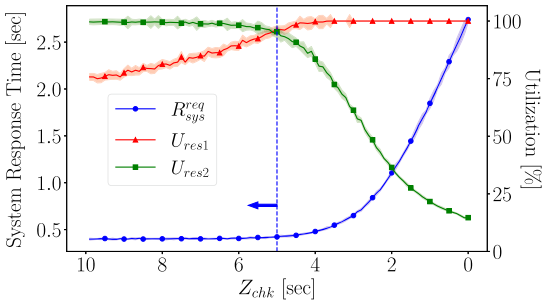
This antipattern is modeled by adding a *Check* class to the baseline, see Fig. 3a. In this case, *Check* requests are repeatedly invoked to verify the status of resources (e.g., battery). *Check* requests visit only the affected resource (i.e., *Resource1*) before being routed to the delay station where they spend Z_{chk} time units. The cyclic invocation of these requests (i.e., the think time) is modeled by a Uniform distribution with average μ and a small range of values, i.e., $\mu \cdot (1 \pm 0.02)$. Differently from [14] where a Deterministic distribution modeled the cyclic nature of this antipattern, here we use a Uniform distribution with a small coefficient of variation (approximately 0.012) to better represent the monitoring of system components without hard real-time performance requirements. There are N_{chk} *Check* requests in the system, each one monitoring the status of the resource. The time needed to verify the component status, S_{chk}, is assumed to be much smaller than the time required to process default requests, i.e., $S_{chk} \ll S^{req}_{res1}$.



(a) QN model

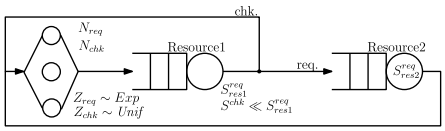
Parameter	Value	Parameter	Value
N_{req}	10	N_{chk}	250
S^{req}_{res1}	0.02	Z_{chk}	(0, 10)
S^{req}_{res2}	0.04	S_{chk}	0.01
# CPUs	1	Z_{req}	0.0

(b) Input parameters



(c) Performance metrics

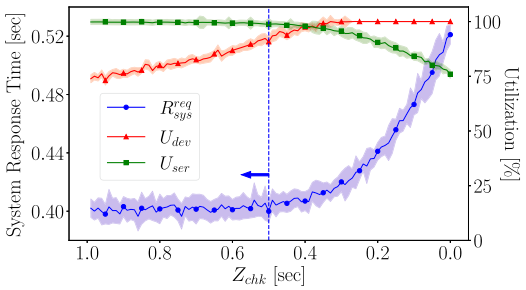
Fig. 2 Are We There Yet?—performance modeling and evaluation



(a) QN model

Parameter	Value	Parameter	Value
N_{req}	10	N_{chk}	250
S^{req}_{res1}	0.02	Z_{chk}	Unif($\mu \cdot 0.98, \mu \cdot 1.02$), $\mu \in (0, 1)$
S^{req}_{res2}	0.04	S_{chk}	0.001
# CPUs	1	Z_{req}	0.0

(b) Input parameters



(c) Performance metrics

Fig. 3 Is Everything OK?—performance modeling and evaluation

3.3.2 Analysis

Numerical values used to analyze this antipattern are shown in Fig. 3b. The number of components whose status is repeatedly checked is set to $N_{chk} = 250$. A status check (the *Is Everything OK?* antipattern) requires fewer resources than requests generated by the *Are We There Yet?* antipattern since only the status of the checked component needs to be returned. Hence, S_{chk} is assumed to be one order of magnitude smaller than the previous case (i.e., $S_{chk} = 0.001$ s). The system performance is studied against Z_{chk} and shown in Fig. 3c where the x-axis is inverted to highlight the effect of the antipattern. Due to the similarity of this antipattern with the previous one [12], the performance indices observed in the two cases show similar behaviors. Since monitoring the status of a component is a fast task, the baseline system response time (i.e., $R^{req}_{sys} = 0.4$) is observed for $Z_{chk} > 0.5$ s, i.e., much shorter than the one observed for *Are We There Yet?* ($Z_{chk} > 5$ s). This is depicted by the dashed blue line and the blue arrow pointing towards the direction where R^{req}_{sys} is not longer than the baseline. For the sake of clarity, Z_{chk} values that allow observing the baseline utilization of resources are not depicted in Fig. 3c. Summing up, *Is Everything OK?* and *Are We There Yet?* antipatterns show similar performance variations (i.e., switching the system bottleneck and increasing/decreasing the utilization of resources). In this case, the performance overhead is due to the high checking frequency rather than the checking activity itself.

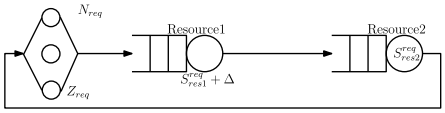
3.4 Where Was I?

3.4.1 Modeling

A process that loses its state must resume the execution from a previous checkpoint. This is modeled by increasing the service time of the process at the station affected by the antipattern. Assuming that this performance antipattern affects only *Resource1*, recomputing the lost state takes Δ time units on average that are added to the service time of the resource, i.e., $S^{req}_{res1} + \Delta$, as shown in Fig. 4a. State recalculation might be a short activity, i.e., the value of Δ may be small. However, there are cases (e.g., connectivity issues) that require extensive processing to recalculate the state [12].

3.4.2 Analysis

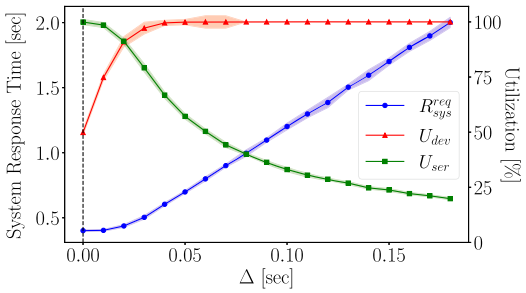
Numerical values used to analyze this antipattern are shown in Fig. 4b. Here we assume a value Δ that is added to the original service time of the affected resource (i.e., *Resource1*) to model the extra processing time required to recalculate the



(a) QN model

Parameter	Value	Parameter	Value
N_{req}	10	Δ	[0.0, 0.18]
S^{req}_{res1}	$0.02 + \Delta$	Z_{req}	0.0
S^{req}_{res2}	0.04		
# CPUs	1		

(b) Input parameters



(c) Performance metrics

Fig. 4 Where Was I?—performance modeling and evaluation

lost state. We consider $0 \leq \Delta \leq 0.18$ s and evaluate the system performance against these values. Results are shown in Fig. 4c. If $\Delta = 0$ s the antipattern has no effect on the system and we observe the baseline performance (i.e., response time and utilization) discussed in Sect. 3.1. In Fig. 4c, this is depicted by the dashed black line. Summing up, if recalculating the lost state is more expensive than the actual computation, the performance of the system deteriorates and the system bottleneck may switch. The system response time increases when requests served by *Resource1* need to recalculate their state (i.e., $S^{req}_{res1} + \Delta$) due to the *Where Was I?* antipattern. *Resource1* is the bottleneck of the system when $\Delta > S^{req}_{res2} - S^{req}_{res1}$, i.e., restoring the state requires extensive processing.

3.5 Circuitous Treasure Hunt

3.5.1 Modeling

This antipattern increases the number of visits required to satisfy a request, e.g., multiple retrievals from a database are required before obtaining the desired information [11]. The model represents this increase by specifying a probability that a request will visit a resource again. This worsens the system performance by making the service demand (i.e., the product of service time and visits) of affected resources longer. In Fig. 5a, this bad practice is modeled by changing the probability p (i.e., $0 \leq p < 1$) to visit *Resource1* again. The higher the value of p the greater the number of visits to the station.

3.5.2 Analysis

Numerical values used to analyze this antipattern are shown in Fig. 5b. To investigate the effect of this software antipattern on the system performance, we vary the probability of requests to visit *Resource1*, i.e., $0 \leq p < 1$. As depicted in Fig. 5c, high values of p increase the number of visits to *Resource1*, switch the system bottleneck (i.e., from *Resource2* to *Resource1*), and make the system response time longer. *Resource2* is the bottleneck of the system only if $p = 0$ (i.e., the baseline). The baseline performance (i.e., response time and utilization) presented in Sect. 3.1 is observed for $p = 0$. In Fig. 5c, this is depicted by the dashed black line. Summarizing, increasing the number of visits to one of the resources makes the response time longer since requests spend more time in the system.

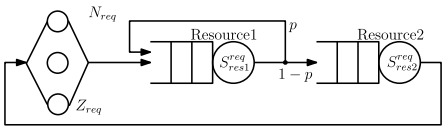
3.6 One-Lane Bridge

3.6.1 Modeling

Single-threaded programs can serve only one process at a time regardless of the available resources. A finite capacity region (i.e., FCR) is used to model the *One-Lane Bridge* antipattern and limit the number of requests that are concurrently processed by resources. Figure 6a depicts this QN model assuming that only *Resource1* is affected by the software antipattern. In this case, both *Resource1* and *Resource2* have multiple CPUs, but the FCR only limits *Resource1*.

3.6.2 Analysis

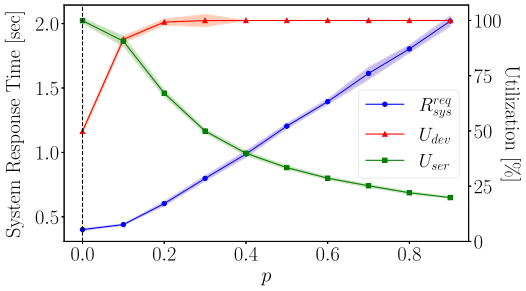
Numerical values used to analyze this antipattern are shown in Fig. 6b. This performance antipattern is studied by changing the baseline and specifying quantity 10 (processing units) for all resources (i.e., *Resource1* and *Resource2*). The computational capacity of *Resource1* (i.e., the resource affected by the antipattern) is reduced to 1 using a finite capacity region. This way, *Resource1* can serve only one request at a time even if it has ten processing units, thus limiting its degree of parallelism. The effect of *One-Lane Bridge* is analyzed against the number of requests in the system, N_{req}, and plotted in Fig. 6c. The shortest system response time is observed when $N_{req} = 1$. In this case, the antipattern (i.e., $FCR = 1$) does not slow down the execution of the request, and *Resource2* is the bottleneck of the system. Since resources might have more processing units than the number



(a) QN model

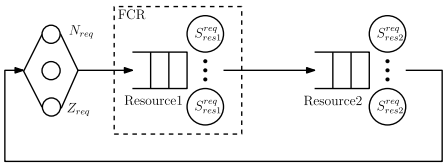
Parameter	Value	Parameter	Value
N_{req}	10	p	[0.0, 0.9]
S^{req}_{res1}	0.02	Z_{req}	0.0
S^{req}_{res2}	0.04		
# CPUs	1		

(b) Input parameters



(c) Performance metrics

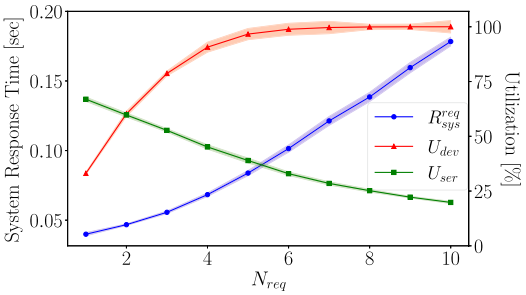
Fig. 5 Circuitous Treasure Hunt—performance modeling and evaluation



(a) QN model

Parameter	Value	Parameter	Value
N_{req}	[1, 10]	FCR	1
S^{req}_{res1}	0.02	Z_{req}	0.0
S^{req}_{res2}	0.04		
# CPUs	10		

(b) Input parameters



(c) Performance metrics

Fig. 6 One-Lane Bridge—performance modeling and evaluation

of requests in the system, the utilization of *Resource1* and *Resource2* is normalized over N_{req}. When $N_{req} > 1$, requests must take turn to be served by *Resource1* since it can process only a request at a time due to the *One-Lane Bridge* antipattern. No N_{req} values allow observing the baseline performance since the number of CPUs allocated to each resource has been increased to model this antipattern. Overall, limiting the computational capacity of resources worsens the system performance when the number of requests to be served increases.

3.7 More is Less

3.7.1 Modeling

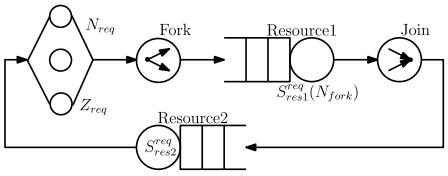
Multiple requests might be sent together to be processed in parallel with the intent to reduce overall response time for handling all requests. Resources that must handle these requests in parallel observe a load surge (due to requests arriving simultaneously) that negatively impacts the system performance. As shown in Fig. 7a, we model this antipattern by placing the affected resource (i.e., *Resource1*) between a *Fork* and a *Join*. All N_{fork} requests spend $S^{req}_{res1}(N_{fork})$ time units at *Resource1* to be processed. We assume that S^{req}_{res1} depends on the number of parallel requests due to possible overheads. When all N_{fork} requests are served by *Resource1*, they are joined together and sent to *Resource2* that handles the result of all requests in S^{req}_{res2} time units.

3.7.2 Analysis

Numerical values used to analyze this antipattern are shown in Fig. 7b. The effect of this antipattern on the system performance is observed by increasing the number of parallel requests (i.e., N_{fork}). In this case, there is a single request into the system (i.e., $N_{req} = 1$) that is forked into $1 \leq N_{fork} \leq 20$ sub-requests before being processed by *Resource1*. We assume that the service time of *Resource1* increases by 10% for every sub-request (except the first one) forked from the initial request and is defined as:

$$S^{req}_{res1}(N_{fork}) = S^{req}_{res1}(1) \cdot \left(1 + \frac{N_{fork} - 1}{10}\right), \quad (1)$$

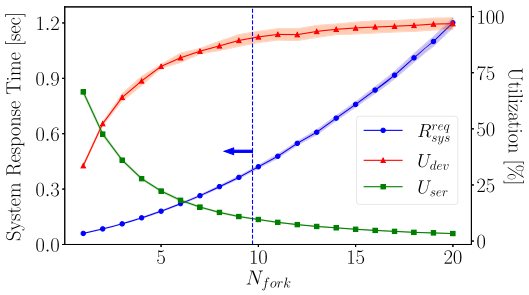
with $S^{req}_{res1}(1) = 0.02$ (i.e., as in the baseline configuration, see Sect. 3.1). Figure 7c depicts the system performance as a function of N_{fork}. The system response time keeps increasing since it accounts for the average time spent by a request at *Resource1* and *Resource2* (including the waiting time at the Join station for all sub-requests being served by *Resource1*).



(a) QN model

Parameter	Value	Parameter	Value
N_{req}	1	N_{fork}	[1,20]
S^{req}_{res1}	$0.02 \cdot \left(1 + \frac{N_{fork}-1}{10}\right)$	Z_{req}	0.0
S^{req}_{res2}	0.04		
# CPUs	1		

(b) Input parameters



(c) Performance metrics

Fig. 7 More is Less—performance modeling and evaluation

Resource1 is the system bottleneck when $N_{fork} > 1$; its utilization keeps increasing. The utilization of *Resource2* is smaller when N_{fork} increases since it processes a single request, i.e., all sub-requests together. The baseline system response time (i.e., $R^{req}_{sys} = 0.4$) is observed for $N_{fork} < 10$. This is depicted by the dashed blue line and the blue arrow pointing towards the direction where R^{req}_{sys} is not longer than the baseline. In this case, the system response time might be even shorter than the baseline value since N_{req} is now set to 1 to model this antipattern. For the sake of clarity, N_{fork} values that allow observing the baseline utilization of resources are not depicted in Fig. 7c.

3.8 The Ramp

3.8.1 Modeling

The processing time of a request might increase over time due to software operations that accumulate (an increasing) overhead while the system is in operation. In Fig. 8a, this bad practice is modeled by defining (for the affected resource, i.e., *Resource1*) a class-dependent service time, $S^{req}_{res1}(cl)$, and using a class-switch component that changes the class of requests visiting it. When the class of a request changes, its service time at *Resource1* increases.

3.8.2 Analysis

Numerical values used to analyze this antipattern are shown in Fig. 8b. Differently from other software antipatterns, here we need to consider the evolution of the system over time to observe the impact of this antipattern on the baseline performance. Assuming that *The Ramp* antipattern affects only *Resource1*, the service time of this resource depends on the class (i.e., cl) of the served request as shown in Fig. 8b. Specifically, requests that visit the class-switch (CS) change their class with a 0.01% probability. Figure 8c depicts the evolution of the response time of the baseline system (flat line) and of the system affected by *The Ramp* antipattern (sloped line) over a pre-defined time interval (i.e., 3 days). All values are obtained by averaging the system response time from 50K observations. Although the effect of *The Ramp* on the system performance is barely visible after a few hours, the system response time keeps increasing and, after only a day, it is almost 50% longer compared to the baseline. After three days, the system affected by *The Ramp* takes 125% longer than the baseline system to process incoming requests.

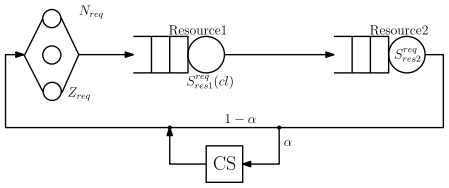
3.9 Traffic Jam

3.9.1 Modeling

Periodic load variations may deteriorate the performance of a system that cannot always provide enough computational power to process all the incoming requests. In Fig. 9a, the *Traffic Jam* antipattern is modeled by introducing a new class of requests (i.e., *Check*) with a long and Uniform think time whose average is μ and possible values range between $0.98 \cdot \mu$ and $1.02 \cdot \mu$. This way, we model a scheduled periodic event that needs a large amount of time $S_{chk} \gg S^{req}_{res2}$ to be processed by the affected resource, i.e., *Resource2* in this case. This model allows studying a system that requires an increased amount of resources for a limited time due to a load that alternates between phases (e.g., light and heavy loads).

3.9.2 Analysis

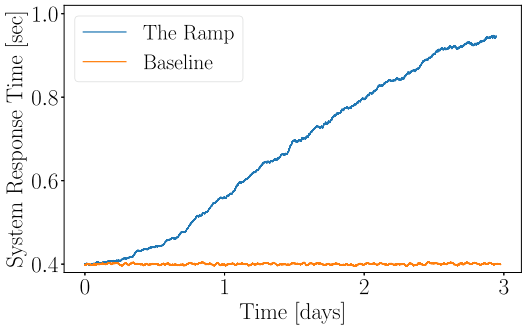
Numerical values used to analyze this antipattern are shown in Fig. 9b. The performance effect of this antipattern is studied by considering the evolution of performance metrics over time. There is $N_{chk} = 1$ *Check* request representing the *Traffic Jam* antipattern in the system. This request is executed once every hour on average, i.e., $Z_{chk} \sim \text{Unif}(3528, 3672)$ seconds, with service time set to 100 s at *Resource2* ($S_{chk} = 100$ s). All values depicted in Fig. 9c are obtained by averaging the system response time of 20K samples. The response



(a) QN model

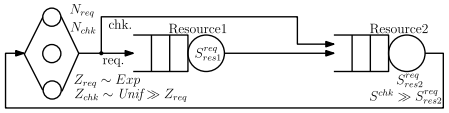
Parameter	Value	Parameter	Value
N_{req}	1	$Pr(CS)$	0.01%
S^{req}_{res1}	$0.02 \cdot (1 + \frac{cl}{10}), cl \in [0, +\infty)$	Z_{req}	0.0
S^{req}_{res2}	0.04		
# CPUs	1		

(b) Input parameters



(c) Performance metrics

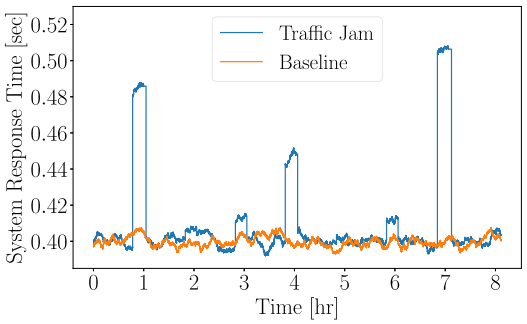
Fig. 8 The Ramp—performance modeling and evaluation



(a) QN model

Parameter	Value	Parameter	Value
N_{req}	1	N_{chk}	1
S^{req}_{res1}	0.02	Z_{chk}	Unif(3528, 3672)
S^{req}_{res2}	0.04	S_{chk}	100
# CPUs	1	Z_{req}	0.0

(b) Input parameters



(c) Performance metrics

Fig. 9 Traffic Jam—performance modeling and evaluation

time of the baseline system and the one affected by *Traffic Jam* are observed over a pre-defined time (i.e., 8 h). If a *Check* request is introduced into the system, *Resource2* needs a long time to handle it, other requests must wait for their turn to be processed, and their system response time increases up to 25% compared to the baseline. When the *Check* request is in the *Delay* station as shown in Fig. 9a, *Resource2* serves other requests, the increased traffic is assimilated, and the system response time goes back to the baseline value.

3.10 Lessons learned

The modeling of software performance antipatterns using QNs is a challenging task due to the need of introducing multiple and different components (e.g., finite capacity region, fork and join, class-switch) that are required for capturing all the performance problems that may originate. However, each antipattern model includes a set of system peculiarities that, despite the required (nontrivial) modeling effort, nicely represent probable bad practices that become actual performance problems. Moreover, QNs enable a sensitivity analysis of considered systems and allow drawing quantitative observations about the effect of antipatterns on the system performance.

The outcome of modeling and analyzing performance antipatterns raises the following main observations for generic CPS. Checking the occurrence of specific events in the systems might delay other operations, and the frequency of performing these checks becomes fundamental to avoid performance issues. Verifying the status of system resources too often is counter-productive since the system might spend too much time performing this verification. Restoring the state of a resource is not always beneficial due to the performance overhead that might prevent other operations from being processed. Accessing a resource too many times causes excessive use of the resource, it is recommended to limit the accesses to the resource. Parallel computation of system operations is beneficial as long as there are enough processors so there is minimal resource contention. A continuing increase in data size may lead to infinite response times. A sudden burst of requests might generate delays that remain in the system for a long time before going back to normal processing of incoming requests.

It is worth remarking that these considerations for generic CPS do lead to quantitative information. In fact, our model-based analysis results show the variation of performance indicators (e.g., system response time, resource utilization) while varying the specificities of CPS, such as the frequency

of checking the status of sensors. Section 4.5 discusses the percentage of performance degradation we found in a concrete case study under analysis.

4 Evaluation

The analyzed system is a SensorNet CPS [7] consisting of a controller, a database, and sensors. It is based on an actual CPS, but it is an abstract version that hides confidential details. Additionally, system details have been modified to inject all the antipatterns considered here. This is necessary because one seldom finds all of the antipatterns present in a single system, whereas we want to illustrate how they can be modeled and analyzed separately and together. The specific antipatterns we inject have been found in other CPS, just not combined in this way. This approach serves as a reference case to show how the antipattern models can be used in a plug and play manner.

4.1 Case study description

This CPS uses sensors to acquire data that are analyzed by the controller and might trigger some actions (e.g., store the observed data in the database, issue control commands, check if the system is properly working).

The network of sensors acquire data at different rates, so the system polls to see if new data is available (ie., *Are We There Yet?* performance antipattern). When sensor data is received, the next step analyzes the data and the required analysis time increases as the system operates (ie., *The Ramp* performance antipattern). This happens when previous sensor values need to be compared to current values and an inappropriate data structure is used that takes longer to access previous values as the number of values increases (e.g., a sequential search of previous values). The *Where Was I?* performance antipattern is also represented in the analysis step by assuming that the analysis needs to start from the beginning each time, and specifying a service time that accounts for this re-calculation.

Sensor values and analysis results are stored in a database. Incoming data is buffered, so when a buffer fills it is written to the database. Data in the database may be needed in the analysis. Two antipatterns are associated with database accesses: the *One-Lane Bridge* may occur when a process must lock the database before updating, and the *Circuitous Treasure Hunt* may occur when multiple database accesses are needed to retrieve the data.

Once the data is captured and analyzed, a set of Actors use the results to make control decisions and issue commands that cause actions to occur in the environment. There are no antipatterns represented by the Actors in this system, but they are representative of CPS that trigger actions based on sensor values, and they introduce contention for resources that demonstrates the effect of the considered software antipatterns. This illustrates how performance problems can propagate to unrelated processing in a system containing performance antipatterns.

Resilience is represented in this CPS by periodically executing virus scans. While traditional CPS (e.g., RTES) do not address cyberthreats, systems of today are increasingly vulnerable, as evidenced by the Stuxnet penetration of the Iranian nuclear power plant [34] and the HVAC penetration into a hospital information system [35]. So we consider the performance affect of adding a virus scan that also represents the *Traffic Jam* performance antipattern. Note that Garbage Collection is another type of *Traffic Jam* that never occurred in traditional CPS, but now also occurs in today's systems. Our case study focuses on the former to illustrate how this performance antipattern can be represented, and thus how additional *Traffic Jam* can be represented when they are present. In our case study, we focus on the performance aspects leaving aside the power constraints on edge devices, such as the sensors, that would make this type of virus scan impractical. We acknowledge this assumption may not apply to all CPS.

The *Is Everything OK?* performance antipattern is represented in this CPS by periodically checking if the sensors are functioning correctly. The performance effect is studied by varying the frequency of the status checks.

The *More is Less* performance antipattern is represented by using *Fork* and *Join* to parallelize the data retrieval and analysis that finds "objects of interest" (OOI) in the data provided by sensors.

The resulting system with all the performance antipatterns injected is somewhat contrived because it is rare to find all of these in one system. Nevertheless, in our combined experience, all but two of these examples of performance antipatterns *have been found in other CPS*, they just have not all been found together. The two exceptions are the database related antipatterns, but those are common antipatterns in database systems in general and as databases become more commonplace in CPS, and they are, it is only a matter of time until those antipatterns occur.

The resulting models illustrate how one or more of these performance antipatterns might be combined in other realistic CPS. The parameters used in the models in the next section (Table 1) are realistic values based on our experience with similar systems, yet adapted somewhat to this example to illustrate the combination of performance antipatterns.

4.2 Modeling

The CPS under analysis is modeled using a multi-class QN, i.e., the same formalism adopted to define the software performance antipatterns in Sect. 3. There are three resources,

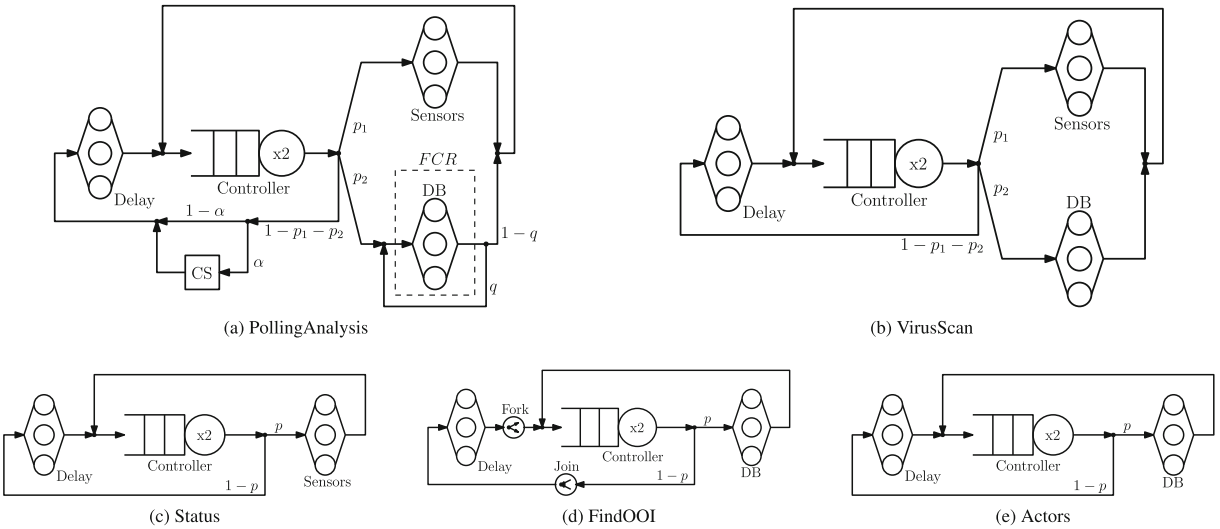


Fig. 10 QN used to study the CPS presented in [7]. Different workloads coexist in the same model and interfere with each other even if they are presented separately for the sake of clarity

i.e., the *Controller* that is modeled as a queuing center with two CPUs and a *First Come First Served* policy, *Sensors*, and *DB*, both represented as a station with infinite servers (hence without any queue). The *Delay* center is used to drive the arrival of new requests. There are five classes of requests in the system, i.e., *PollingAnalysis*, *VirusScan*, *Status*, *FindOOI*, and *Actors*, whose interactions with the resources of the CPS are depicted in Fig. 10a–e. Although these classes are depicted separately for the sake of clarity, they coexist in the system and compete for the available resources interfering with each other. Request classes used to investigate the effect of software antipatterns on the system performance are discussed in the following. Input parameters used to define the baseline system are shown in Table 1.

PollingAnalysis. This class represents data being polled from the sensors, analyzed by the controller, and stored in the database. There are N_{PoAn} requests of this class and each one spends Z_{PoAn} time units in the *Delay*. The delay represents the frequency of *PollingAnalysis* requests and is used to study the *Are We There Yet?* antipattern. These requests are then sent to the Controller where they are processed in $S_{ctrl}^{PoAn}(c)$ time units. This time increases with the value of c, i.e., the sub-class of *PollingAnalysis* requests. This way, we can evaluate the effect of *The Ramp* antipattern on the system performance. When c is fixed, we vary the value of S_{ctrl}^{PoAn} to quantify the effect of the *Where Was I?* antipattern. After having been processed by the *Controller*, *PollingAnalysis* requests are sent to the *Sensors* with probability p_1. Here, they spend $S_{sensors}^{PoAn}$ time units to acquire data. With probability p_2, requests go to the *DB* where they store information in S_{db}^{PoAn} time units. In all other cases, the requests have already polled and analyzed data, hence they go back to the *Delay* station to restart the process. Before reaching the *Delay*, there is a probability α that the request goes through the class-switch *CS* and its sub-class c is changed. The number of *PollingAnalysis* requests concurrently processed by the database is limited by the size of the FCR (requests that find the FCR busy must wait outside the region for their turn). This way, we limit the capacity of the *DB* and investigate the effect of the *One-Lane Bridge* antipattern when N_{PoAn} changes. The FCR affects only *PollingAnalysis* requests, i.e., requests of a different class enter the *DB* as soon as they are completed by the *Controller*. When *PollingAnalysis* requests are processed by the *DB*, there is a probability q that they need to visit the *DB* again before being processed by another resource. The

Table 1 Parameters of the CPS used to validate the QN model and evaluate the effect of performance antipatterns. Time values (inspired by real-world CPS tasks [33]) are in milliseconds (ms) and follow an Exponential distribution with the given mean value (unless differently indicated). The max value of p, p_1, p_2, q, and α is 1

Class	Fig.	N	Z	S_{ctrl}	S_{db}	$S_{sensors}$	N_{fork}	FCR	p	p_1	p_2	q	α
PollingAnalysis	10(a)	10	75	0.30067	0.6	0.1	–	1	–	11/15	3/15	0	0.0001
VirusScan	10(b)	1	$\text{Unif}(0.98 \cdot 10^{-14}, 1.02 \cdot 10^{-13})$	2048.78	0.6	1000	–	–	–	20/41	20/41	–	–
Status	10(c)	1	Unif(0.098, 0.102)	0.05	–	1	–	–	0.5	–	–	–	–
FindOOI	10(d)	1	1000	1.2155	0.6	–	10	–	0.5	–	–	–	–
Actors	10(e)	5	30	0.72	0.6	–	–	–	2/3	–	–	–	–

value of q is used to analyze the *Circuitous Treasure Hunt* antipattern.

VirusScan. This class visits the *DB* with probability p_2 to retrieve the latest virus definitions then scans *Sensors* with probability p_1 to check that no virus is affecting them. Results are always reported to the *Controller*. The time required by these requests at the *Controller*, *Sensors*, and *DB* (i.e., $S_{ctrl}^{VirusScan}$, $S_{sensors}^{VirusScan}$, and $S_{db}^{VirusScan}$, respectively) are longer than the service time of all other request classes to represent the impact of the *Traffic Jam* antipattern on system performance. For model validation, we set $Z_{VirusScan}$ to approximately 0 s to represent only the busy period when the virus scan is active, and to obtain sufficient completions for model precision. The variation of $Z_{VirusScan}$ is discussed in Sect. 4.4 where experiments show the performance evolution of the *Traffic Jam* antipattern.

Status. These are requests issued by the system to check that *Sensors* are working as expected. There are N_{Status} requests that spend Z_{Status} time units in the *Delay* station. Changing the think time of *Status* requests allows investigating the effect of the *Is Everything OK?* antipattern on the system performance. *Status* requests are then served by the Controller in S_{ctrl}^{Status} time units. With probability p, they must check the status of *Sensors* (i.e., $S_{sensors}^{Status}$), whereas with probability $1-p$ they have already reported the status to the controller and go back to the *Delay* station.

FindOOI. This class represents requests that find OOI: they alternate processing at the *Controller*, and retrieving data from the *DB*. These requests introduce the *More is Less* antipattern by using *Fork* and *Join* to parallelize the data retrieval and analysis. The *Controller* service time of each forked sub-request depends on N_{fork} as in Eq. (1). The service time (when only one sub-request is forked) is $S_{ctrl}^{FindOOI}(1) = 0.639737$ ms and the service time with $N_{fork} = 10$ (see Table 1) is:

$$\begin{aligned} S_{ctrl}^{FindOOI}(10) &= S_{ctrl}^{FindOOI}(1) \cdot \left(1 + \frac{10-1}{10}\right) \\ &= 0.639737 \text{ ms} \cdot (1 + 0.9) \\ &= 1.2155 \text{ ms}. \end{aligned}$$

When a forked sub-request is completed (with probability $1-p$), it waits in the Join station for all other forked sub-requests to retrieve the required data before being joined into the original request and returning to the *Delay* station.

Actors. This class models requests that need to use *Controller* and *DB* for their execution. As stated earlier, it is not meant to inject software antipatterns into the system, but it increases the contention for resources and emphasizes the effect of the considered software antipatterns.

4.3 Model-based comparison of analysis results

The baseline QN model proposed in Sect. 4.2 is verified against an extended version of the execution graph (EG) model originally proposed in [14]. The EG model is solved with SPE·ED [5], i.e., a tool designed to support SPE methods and models, that returns performance analysis results. Each request class described in Sect. 4.2 is modeled by a SPE·ED scenario derived from a sequence diagram. It is possible to generate EGs from sequence diagrams by following the flow of messages through the performance scenario and representing actions as basic nodes in the EG. For instance, Fig. 11 shows the sequence diagram of the PollingAnalysis class, and Fig. 12 depicts the most performance-wise relevant actions from the PollingAnalysis sequence diagram. Table 2 reports performance measures (i.e., system response time, system throughput, and controller utilization) obtained by simulating the EG model with SPE·ED and the QN model with JMT. All simulations (except those showing the evolution of performance indices over time, see Sects. 4.4.7 and 4.4.8) in this

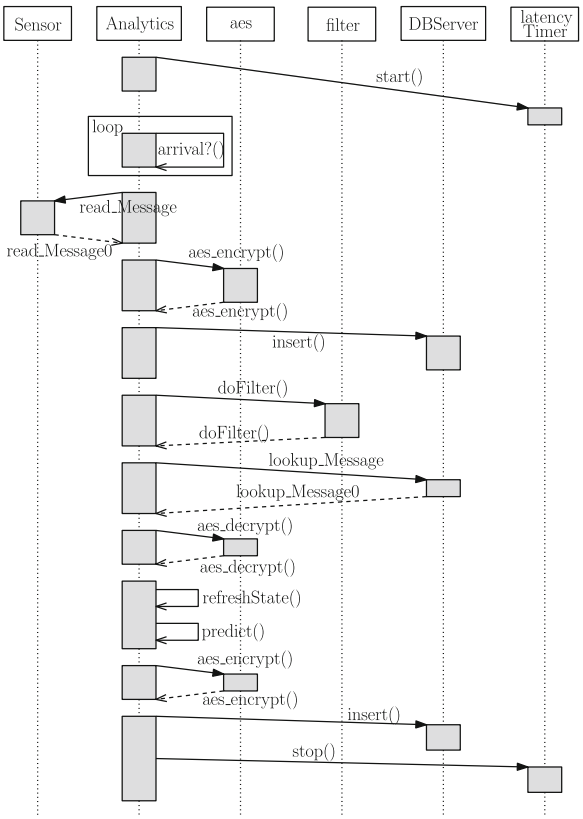


Fig. 11 Sequence diagram of PollingAnalysis scenario [14]

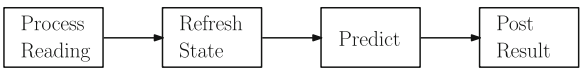


Fig. 12 Steps in the PollingAnalysis sequence diagram [14]

Table 2 EG and QN results. The 99% confidence interval of JMT simulations is shown in parenthesis. Mean absolute percentage errors (MAPEs) are computed wrt. average values

Class	System Response Time			System Throughput			Controller Utilization		
	EG [msec]	QN [msec]	MAPE [%]	EG [req/sec]	QN [req/sec]	MAPE [%]	EG [%]	QN [%]	MAPE [%]
PollingAnalysis	17.700	18.374 (± 0.376)	3.808	107.782	107.096 (± 0.431)	0.636	23.86	24.23 (± 0.71)	3.539
VirusScan	98,091.900	101,365.617 (± 2,746.938)	5.527	0.010	0.010 (± 0.000)	0.000	40.79	40.57 (± 1.08)	0.542
Status	2.000	1.949 (± 0.050)	2.550	469.872	487.997 (± 11.843)	3.857	2.34	2.43 (± 0.05)	3.539
FindOOI	33.500	31.976 (± 0.815)	4.549	0.959	0.969 (± 0.001)	1.043	1.16	1.18 (± 0.03)	1.356
Actors	6.100	5.912 (± 0.148)	3.082	138,564	139.231 (± 0.574)	0.481	14.96	15.04 (± 0.33)	0.525

section stop when every analyzed metric is observed with 99% confidence interval and 3% maximum relative error. Simulations are stopped despite the relative error values when 100 M samples are collected, i.e., the maximum number of samples to analyze is set to 100 M. The mean absolute percentage error (i.e., MAPE) made by the QN when compared to the EG is also reported in Table 2. Specifically, the MAPE is computed as:

$$\text{MAPE}\ (\%) = \frac{|M_{EG} - M_{QN}|}{M_{EG}} \cdot 100,$$

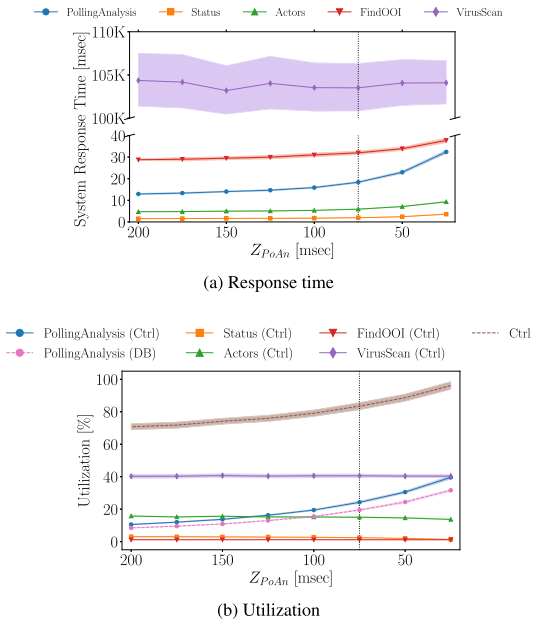
where M_{EG} is the measure obtained from the EG, M_{QN} is the measure obtained from the QN, and the result is multiplied times 100 to give a percentage error. Observed MAPEs are never larger than 6%, meaning that the QN model discussed in Sect. 4.2 is a faithful representation of the extended CPS used in [14]. This strengthens the adoption of QNs as a valid modeling notation to predict the performance of real-world systems.

4.4 Antipattern experiments

Figures 13, 14, 15, 16, 17, 18, 19 and 20 show the effect of the analyzed software antipatterns on the system response time and controller utilization of the considered CPS. For antipatterns associated with the evolution of the system performance (i.e., *The Ramp* and *Traffic Jam*, see Sect. 3), only the system response time is shown since it is derived from simulation logs. The baseline performance used to verify the QN model of the CPS in Sect. 4.3 is indicated by a dotted vertical line. Figures depicting utilization show the overall *Controller* utilization, the *Controller* utilization of each class, and the *DB* utilization for the *PollingAnalysis* class (i.e., the only class for which the *DB* is not modeled by an infinite server station due to the Finite Capacity Region).

4.4.1 Are We There Yet?

This software antipattern is injected into the CPS by decreasing the time between two consecutive *PollingAnalysis* requests, i.e., changing the Z_{PoAn} value. Consistently with



(a) Response time

(b) Utilization

Fig. 13 Are We There Yet?

what is observed in Sect. 3.2, Fig. 13a (note the inverted x-axis) shows that as the frequency of *PollingAnalysis* requests increases (i.e., the smaller is Z_{PoAn}), the time required to process all requests increases exponentially. Figure 13b (that also shows an inverted x-axis) depicts the *Controller* utilization whose trend follows that of the *PollingAnalysis* class. The *Controller* and *DB* usages decrease when Z_{PoAn} increases due to the longer time spent by *PollingAnalysis* requests in the *Delay* station. The *Controller* utilization of other classes is not affected by Z_{PoAn}.

4.4.2 Is Everything OK?

The effect of this performance antipattern is studied by varying the frequency of status checks, i.e., changing the Z_{Status} value. To better highlight it, N_{Status} (i.e., the number of *Status* requests) is set to 20. In this case, no dotted line is depicted in Fig. 14a and b since the baseline performance is obtained with $N_{Status} = 1$. Considering the system response time in Fig. 14a (note the inverted x-axis), *Is Everything OK?* mainly

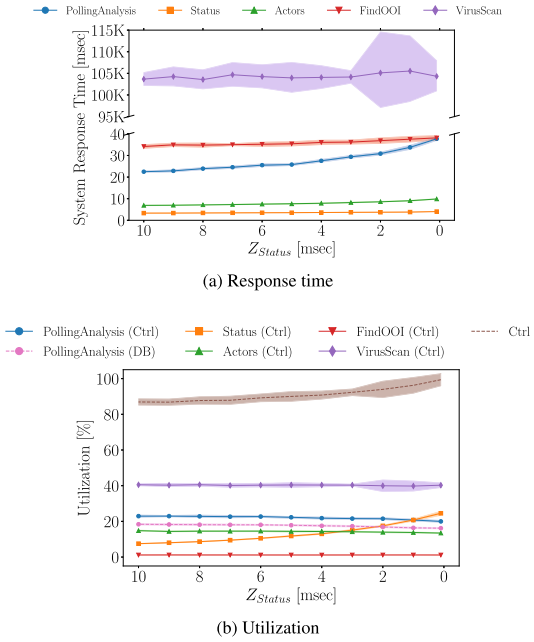


Fig. 14 Is Everything OK?

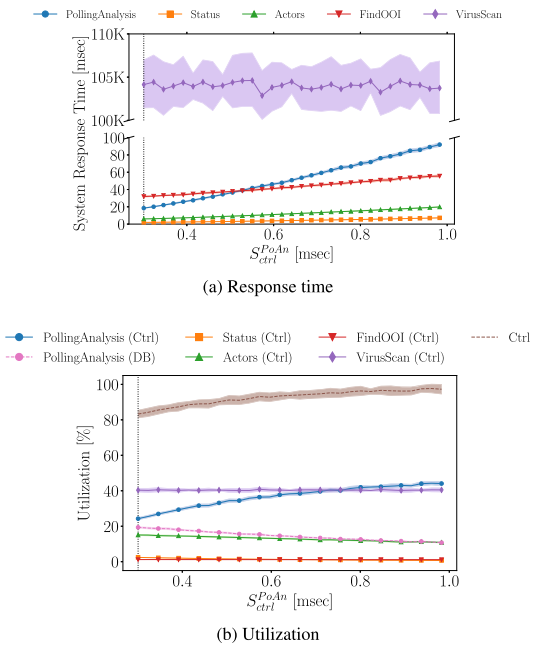


Fig. 15 Where Was I?

affects the *PollingAnalysis* class, and light effects are also observed for *Actors* and *FindOOI* classes. When the status of system components is checked too frequently, all requests compete with *Status* for *Controller* resources. Looking at the utilization of the *Controller* in Fig. 14b (its x-axis is also inverted), the curve for the *Status* class shows the largest variation since requests of this class spend more time in the *Controller* when Z_{Status} is short. The overall *Controller* utilization decreases when Z_{Status} increases; this improves the performance of all system classes.

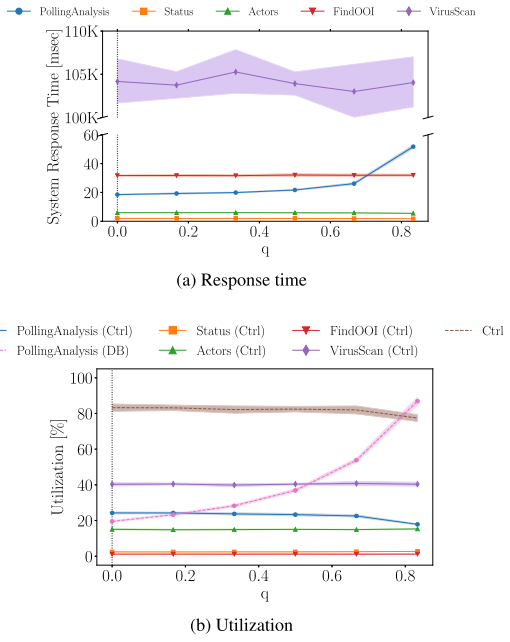


Fig. 16 Circuitous Treasure Hunt

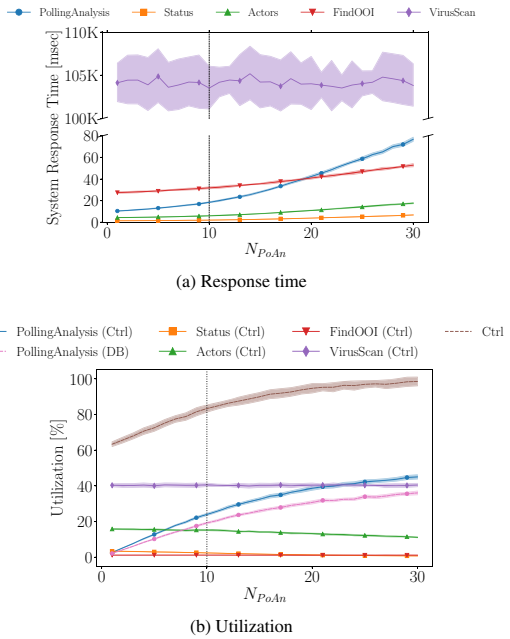
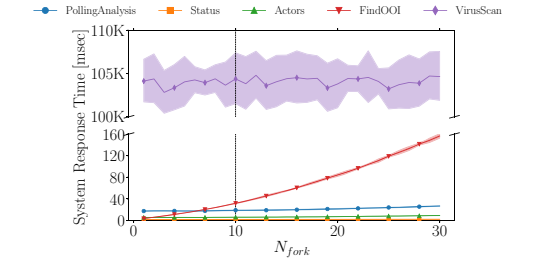


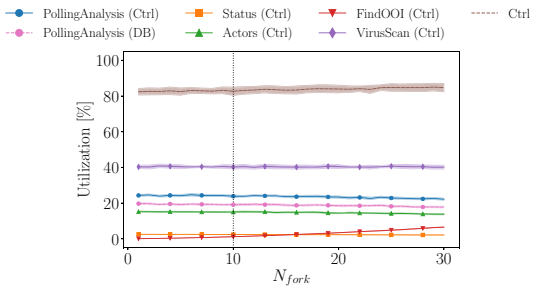
Fig. 17 One-Lane Bridge

4.4.3 Where Was I?

This antipattern is injected by increasing the time spent by *PollingAnalysis* requests in the *Controller*. Figures 15a and b show that both the system response time and the *Controller* utilization of the *PollingAnalysis* class increase with S_{ctrl}^{PoAn}



(a) Response time



(b) Utilization

Fig. 18 More is Less

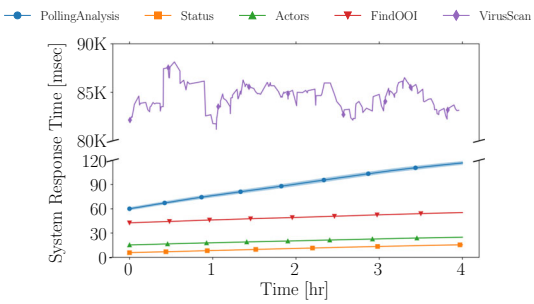


Fig. 19 The Ramp—response time

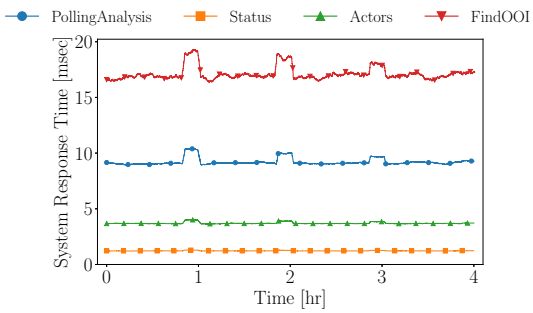


Fig. 20 Traffic Jam—response time

due to the longer time spent by these requests in the *Controller*, while the *DB* utilization of the *PollingAnalysis* class decreases. As a consequence, the system response time of other classes also increases with S_{ctrl}^{PoAn}.

4.4.4 Circuitous Treasure Hunt

To analyze this performance antipattern, we change the probability q that *PollingAnalysis* requests visit the *DB*. Its effect is visible on the system response time of *PollingAnalysis* requests, see Fig. 16a; it is longer when there is a high probability that these requests visit the *DB* multiple times. The effect on the *Controller* utilization, Fig. 16b, is almost negligible since the antipattern overhead is on the *DB* whose utilization increases with q and becomes the system bottleneck for large values of q.

4.4.5 One-Lane Bridge

As described in Sect. 3.6, the effect of this antipattern is observed by increasing the number of requests. Here, we change N_{PoAn}, i.e., the number of *PollingAnalysis* requests. The larger the load intensity (i.e., N_{PoAn}), the longer the system response time of all classes in the system, see Fig. 17a, due to the increased resource contention at the *Controller*. As expected, the *Controller* and *DB* usages also increase due to the larger number of *PollingAnalysis* requests, see Fig. 17b. The *Controller* utilization of other classes is barely affected by this antipattern.

4.4.6 More is Less

This antipattern is studied against the number of sub-requests (i.e., N_{fork}) forked from the *FindOOI* request (see Table 1). Only *FindOOI* requests are affected by *More is Less*; Fig. 18a shows longer system response times when N_{fork} increases. The increased system response time of *FindOOI* requests is due to the time spent by the forked sub-requests at the *Join* station waiting for the completion of all other sub-requests. This is visible from Fig. 18b which shows that the *Controller* utilization does not change with N_{fork}. This illustrates the performance effect of the guilty *More is Less* antipattern when the number of forked sub-requests exceeds the number of processors.

4.4.7 The Ramp

The effect of this antipattern is observable when the evolution of the system performance is analyzed over time. Figure 19 shows the system response time of the five classes within a 4-hour interval. All classes (except *VirusScan*) show an increasing trend during the observation period due to the service time of *PollingAnalysis* requests (i.e., the class on which *The Ramp* has the greatest impact) at the *Controller* (S_{ctrl}^{PoAn}) which might become 10% longer when these requests go through the class-switch, see Sect. 4.2. The system response time of *VirusScan* requests is not affected much by this antipattern since it is four orders of magnitude longer than the

response time of *PollingAnalysis* requests. Results shown in Fig. 19 are obtained by averaging the system response time observed for all the requests of the same class.

4.4.8 Traffic Jam

The *VirusScan* request (i.e., $N_{VirusScan} = 1$) is used to inject this antipattern into the system. To highlight the impact of *Traffic Jam*, differently from the value in Table 1, the think time of *VirusScan* is $Z_{VirusScan} \sim \text{Unif}(3528, 3672)$ seconds. This way, the system is scanned for viruses once per hour on average. The effect of the antipattern is presented in Fig. 20 where the system response time of *FindOOI*, *PollingAnalysis*, and *Actors* requests increases when *VirusScan* checks the system for viruses. The effect of *Traffic Jam* on the system response time of the *Status* class is negligible due to the short service time of this class at the *Controller*. For the sake of clarity, the system response time of the *VirusScan* request is not depicted in Fig. 20 since this class is used to inject the Traffic Jam in the system and the antipattern has no effect on it.

4.5 Lessons learned

Our experimentation shows the following main findings. When the SensorNet checks too often if sensors polled new data (*Are We There Yet?*), the response time of *PollingAnalysis* degrades by 151%. Frequently checking the status of sensors (*Is Everything OK?*) delays other system operations, e.g., the response time of *PollingAnalysis* deteriorates by 50%. If the SensorNet recalculates the lost state (*Where Was I?*), the response time of *PollingAnalysis* increases up to 398%. Repeatedly accessing the database (*Circuitous Treasure Hunt*) worsens the response time of *PollingAnalysis* up to 180%. Limiting the concurrency when accessing to the database (*One-Lane Bridge*) delays *PollingAnalysis* requests by 638%. A large number of batched requests (*More is Less*) increases by 4208% the response time of *FindOOI* requests. If size of data stored in the database increases over time (*The Ramp*), other operations that need to access the database, e.g., *PollingAnalysis* requests, take longer, e.g., up to 1870%. Checking the presence of viruses at a specific point in time for all sensors (*Traffic Jam*) clogs the system, e.g., *FindOOI* requests experience delays up to 29%. The *Controller* utilization increases for most of the antipatterns, and the maximum growth of 40% is observed for the *One-Lane Bridge*. Overall, the performance degradation is significant and performance antipatterns nicely capture the root causes of such deterioration.

Summarizing, we conclude that generic CPS do show several bad practices leading to performance issues. For instance, checking too often the status of system events leads to performance degradation. If an internal routine is in charge of verifying the functioning of resources, and the verification is executed too frequently, system requests are inevitably delayed. When processes do not remember state information, it might be necessary to look for current information. This increases computation which then increases resource utilization and response time. Further bad practices are: excessive access to the database, limiting the concurrency, allowing a large number of batched requests, establishing time synchronization among operations executed by physical entities, and ever-increasing size of data structures.

The benefit of our model-based analysis is the quantification of performance degradation. This way, designers can verify if system performance requirements are satisfied. Our antipattern models also allow relating detected problems to their root causes. This way, designers can refactor the software using remedies prescribed for antipatterns. More importantly, refactored systems can be analyzed to check if they meet the stated requirements.

5 Threats to validity

Besides inheriting all limitations of the underlying software performance engineering research [36], our approach exhibits the following main threats to validity [37].

Construct threats relate to the validity of metrics used during our experimentation. To smooth these types of threats, all simulations undergo a 99% confidence interval, so the accuracy of the presented experimental results has been monitored.

Conclusion threats deal with the reliability of collected measures. To smooth these threats, the model-based performance analysis is delegated to two well-assessed and widely-used tools for this scope, i.e., JMT [31] and SPE·ED [5].

Internal threats are related to how we designed our experiments. Queuing models include a set of input parameters whose numerical value may largely influence the observed fluctuations in the system performance. To smooth this type of threat, we provide the models as part of our replication data, and users can set their own numerical values, so that additional parameter values can be analyzed. QNs represent an abstraction of the software system under analysis, we acknowledge that the connection to the actual system design and implementation, as well as the quantitative validation against a real system, remains an open issue. We refer to [16–19] to support the validity of QN models approximating actually implemented systems. Moreover, the choice of using QNs as the target notation to model antipatterns does not reduce the applicability of our approach. In principle, any formalism can be adopted to model antipatterns as long as it is suitable to manifest performance variations. We plan to further experiment this point by investigating other languages to model antipatterns.

External threats concern the generalization of results. We are aware this is not guaranteed, since our models have been applied to the SensorNet only, however it has been used as a CPS representative example in software performance engineering [7]. Although CPS are the focus of this manuscript, the proposed abstract models generalize to other types of systems. Hence, we expect that it is possible to generalize our results and findings, and we plan as future work to investigate the applicability of proposed models in different application domains.

6 Related work

The work presented in this paper relates to two main streams of research that we review in the following.

Software Performance Antipatterns. They have been defined in the literature as bad practices leading to performance issues [5], and recently customized for the CPS domain in [12]. Performance modeling of software antipatterns is an open issue of the performance engineering domain, since many system characteristics need to be captured, and it is not trivial to derive the expected performance degradation. This paper addresses this challenge presenting QN models that give evidence of antipatterns' impact on the system performance. Our recent work focused on investigating the performance antipatterns across the operational profile space [38], previously defined with a first-order logic representation, and later applied to multiple modeling notations. A first attempt of injecting software performance antipatterns in systems is provided in [39], where the root causes of performance problems are isolated and matched with the specification of antipatterns. More recently, load testing and profiling data is exploited to detect software performance antipatterns when running Java applications in [40]. Application profiling is used also in [41] where patterns are adopted; metrics measure their architectural impact and potential performance optimization. There exist other approaches dealing with different types of antipatterns, e.g., in [42] the focus is on services, and detection algorithms are generated out of a simplified metamodel whose specification is explicitly tailored for service-based systems. In the broader context of matching the connections between (anti)patterns and quality attributes (such as reliability and security), several approaches e.g., [43, 44] are representative. Recently, the number of detected performance antipatterns has been adopted as a parameter to optimize performance and reliability properties of software systems in [45]. Complementary to performance antipatterns, a line of research focuses on monitoring the runtime performance characteristics of software systems subject to dynamic changes, e.g., in [46] monitors are used to instrument system components with the goal of diagnosing performance problems such as bottlenecks and hotspots. Run-time adaptation is tackled also in [47] where principles of designing smart CPS are reviewed, and performance is acknowledged as a characteristic that changes over time due to new operational circumstances affecting the system behavior. Overall, the main difference with state-of-the-art approaches using software performance antipatterns is that they do not provide plug-and-play models that analyze the performance characteristics of CPS as we do in this paper.

Modeling and Performance Evaluation of CPS. Model-based performance analysis of CPS is an open issue of the performance engineering domain since the interplay between cyber and physical entities is challenging. This paper advocates the introduction of performance models that capture the most common bad practices leading to performance problems. In the literature several approaches have been defined for CPS modeling (e.g., [48–50]), however most approaches investigate the security-related aspects of CPS (e.g., [51]). The performance evaluation of CPS uses a plethora of techniques [52], and there exist two macro classes: (i) *analytical* and (ii) *simulation* analysis. Analytical approaches use mathematical formulas or equations that are formal and rigorous, but they may fail to capture some system dynamics (e.g., unexpected events, uncertainties, transient states) that can be expressed in simulation environments [53] (i.e., emulating the system behavior) at the cost of less scalability [54]. A linear stochastic model is adopted in [55] to quantify the performance degradation of CPS when exposed to integrity attacks. In [56] the performance evaluation is conducted through a control law that undergoes a trade-off analysis including privacy costs. In [57] Markov models are applied in the intelligent transportation system domain, and traffic is guided by model predictions. A framework is proposed in [58] to improve the performance of heterogeneous systems at design time, but software allocation is considered only as refactoring strategy. A Markovian environment is described in [59], where queuing models are adopted to get performance indices of a CPS, with the goal of quantifying resource provisioning under uncertain workload. Uncertainty is also investigated in [60] where the sensitivity of the performance models is studied taking into account variations in the parameters of different modeling elements. A simulation-based approach is proposed in [61] to evaluate how human factors affect the performance of CPS when human interaction is required. Overall, the main difference with state-of-the-art approaches modeling and evaluating the performance of CPS is that they do not convey the root causes of performance issues as we do in this paper.

Summarizing, to the best of our knowledge, none of these approaches is specifically tailored to modeling and analyzing software performance antipatterns to support software developers in the interpretation of CPS performance issues.

7 Conclusion and future work

This paper presents a model-based approach to understand the performance issues of reactive systems, such as CPS, under development. We develop plug-and-play QN models to analyze the impact of eight software performance antipatterns on CPS. These models allow users to quantitatively determine the root causes of performance problems in reactive systems. The performance deterioration due to these antipatterns might undermine software resilience, e.g., by slowing the analysis of incoming requests and preventing the system from managing critical situations, thus failing to meet performance and other requirements. This leads to stakeholder dissatisfaction and economic loss, especially when real-time concerns are not satisfied. Experimental results obtained by applying our model-based approach show increased system response time due to a software bottleneck switch. When the abstract models are applied to a real-world CPS, quantitative results confirm that antipatterns deeply affect the system performance.

As future work, we plan to develop a framework that automatically detects antipatterns in CPS by monitoring the system performance and exploiting the provided abstract models (along with their corresponding analysis results). For instance, knowing the point where the system bottleneck switches is of key relevance to preventing it. Moreover, we want to determine which antipatterns are the major culprits in terms of performance degradation, i.e., how much antipatterns contribute to the violation of requirements, to prioritize their solution when they coexist in a CPS. This is of key relevance to enable technology for future work that automatically detects the presence of antipatterns, determines which ones are more relevant, and thus points out how to refactor systems for the removal of these bad practices.

Acknowledgements The authors would like to thank the Editor and the anonymous reviewers for their constructive comments and valuable feedback. This work has been partially funded by MUR PRIN project 20228FT78M DREAM (modular software design to reduce uncertainty in ethics-based cyber-physical systems) and the PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA - Advanced Space Technologies and Research Alliance.

Funding Open access funding provided by Gran Sasso Science Institute - GSSI within the CRUI-CARE Agreement.

References

1. Harman, M., O'Hearn, P.W.: From start-ups to scale-ups: opportunities and open problems for static and dynamic program analysis. In: Proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM), pp. 1–23 (2018)
2. Stecklein, J.M., Dabney, J., Dick, B., Haskins, B., Lovell, R., Moroney, G.: Error cost escalation through the project life cycle. NASA technical report (2004)
3. Chen, T.-H., Shang, W., Jiang, Z.M., Hassan, A.E., Nasser, M., Flora, P.: Detecting performance anti-patterns for applications developed using object-relational mapping. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 1001–1012 (2014)
4. Kolesnikov, S.S., Siegmund, N., Kästner, C., Grebhahn, A., Apel, S.: Tradeoffs in modeling performance of highly configurable software systems. Softw. Syst. Model. **18**(3), 2265–2283 (2019)
5. Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley, Boston (2002)
6. Connell, W., Menascé, D.A., Albanese, M.: Performance modeling of moving target defenses with reconfiguration limits. IEEE Trans. Depend. Secur. Comput. **18**(1), 205–219 (2021)
7. Gómez, A., Smith, C.U., Spellmann, A., Cabot, J.: Enabling performance modeling for the masses: initial experiences. In: Proceedings of the International Conference on System Analysis and Modeling (SAM), pp. 105–126 (2018)
8. Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., Meedeniya, I.: Software architecture optimization methods: a systematic literature review. IEEE Trans. Softw. Eng. **39**(5), 658–683 (2013)
9. Talcott, C.: Cyber-physical systems and events. In: Software-Intensive Systems and New Computing Paradigms: Challenges and Visions, pp. 101–115 (2008)
10. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, Cambridge (2007)
11. Smith, C.U., Williams, L.G.: More new software performance antipatterns: even more ways to shoot yourself in the foot. In: Proceedings of the International Conference on Computer Measurement Group (CMG), pp. 717–725 (2003)
12. Smith, C.U.: Software performance antipatterns in cyber-physical systems. In: Proceedings of the International Conference on Performance Engineering (ICPE), pp. 173–180 (2020)
13. van Dinten, I., Derakhshanfar, P., Panichella, A., Zaidman, A.: The slow and the furious? Performance antipattern detection in cyber-physical systems. SSRN (2023). https://ssrn.com/abstract=4459043
14. Pinciroli, R., Smith, C.U., Trubiani, C.: QN-based modeling and analysis of software performance antipatterns for cyber-physical systems. In: Proceedings of the International Conference on Performance Engineering (ICPE), pp. 93–104 (2021)
15. Lazowska, E.D., Zahorjan, J., Scott Graham, G., Sevcik, K.C.: Computer System Analysis Using Queueing Network Models. Prentice-Hall, New Jersey (1984)
16. Lu, Y., Wang, B., Huang, L., Zhao, N., Su, R.: Modeling of driver cut-in behavior towards a platoon. IEEE Trans. Intell. Transp. Syst. **23**(12), 24636–24648 (2022)
17. Zhu, Y., Bai, W., Sheng, M., Li, J., Zhou, D., Han, Z.: Joint UAV access and GEO satellite backhaul in IoRT networks: performance analysis and optimization. IEEE Internet Things J. **8**(9), 7126–7139 (2021)

18. Hassan, H.H., Bouloukakis, G., Kattepur, A., Conan, D., Belaïd, D.: PlanIoT: A framework for adaptive data flow management in IoT-enhanced spaces. In: IEEE/ACM Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 157–168 (2023). https://doi.org/10.1109/SEAMS59076.2023.00029
19. Kattepur, A.: Towards structured performance analysis of Industry 4.0 workflow automation resources. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE), pp. 189–196 (2019)
20. Pinciroli, R., Smith, C.U., Trubiani, C.: Replication package: Modeling more software performance antipatterns in cyber-physical systems. https://doi.org/10.6084/m9.figshare.15101925 (2022)
21. Mandrioli, C., Carlsson, M.N., Maggio, M.: Testing abstractions for cyber-physical control systems. ACM Trans. Softw. Eng. Methodol. https://doi.org/10.1145/3617170 (2023)
22. Giaimo, F., Andrade, H., Berger, C.: Continuous experimentation and the cyber-physical systems challenge: an overview of the literature and the industrial perspective. J. Syst. Softw. **170**, 110781 (2020)
23. Schranz, M., Di Caro, G.A., Schmickl, T., Elmenreich, W., Arvin, F., Şekercioğlu, A., Sende, M.: Swarm intelligence and cyber-physical systems: concepts, challenges and future trends. Swarm Evol. Comput. **60**, 100762 (2021)
24. Audrito, G., Casadei, R., Damiani, F., Stolz, V., Viroli, M.: Adaptive distributed monitors of spatial properties for cyber-physical systems. J. Syst. Softw. **175**, 110908 (2021)
25. Bai, Y., Huang, Y., Xie, G., Li, R., Chang, W.: Asdys: dynamic scheduling using active strategies for multifunctional mixed-criticality cyber-physical systems. IEEE Trans. Ind. Inf. **17**(8), 5175–5184 (2020)
26. Capota, E.A., Stangaciu, C.S., Micea, M.V., Curiac, D.-I.: Towards mixed criticality task scheduling in cyber physical systems: challenges and perspectives. J. Syst. Softw. **156**, 204–216 (2019)
27. Bures, T., Matena, V., Mirandola, R., Pagliari, L., Trubiani, C.: Performance modelling of smart cyber-physical systems. In: Proceedings of the International Conference on Performance Engineering (ICPE), pp. 37–40 (2018)
28. Stankovic, J.A.: Misconceptions about real-time computing: a serious problem for next-generation systems. IEEE Comput. **21**(10), 10–19 (1988)
29. Petriu, D.B., Woodside, M.: An intermediate metamodel with scenarios and resources for generating performance models from UML designs. Softw. Syst. Model. **6**, 163–184 (2007)
30. Li, C., Altamimi, T., Zargari, M.H., Casale, G., Petriu, D.C.: Tulsa: a tool for transforming UML to layered queueing networks for performance analysis of data intensive applications. In: Bertrand, N., Bortolussi, L. (eds.) Proceedings of the International Conference on Quantitative Evaluation of Systems (QEST), vol. 10503, pp. 295–299 (2017)
31. Bertoli, M., Casale, G., Serazzi, G.: JMT: performance engineering tools for system modeling. SIGMETRICS Perform. Eval. Rev. **36**(4), 10–15 (2009)
32. Dallery, Y.: Approximate analysis of general open queuing networks with restricted capacity. Perform. Eval. **11**(3), 209–222 (1990)
33. Higuera-Toledano, M.T., Risco-Martín, J.L., Arroba, P., Ayala, J.L.: Green adaptation of real-time web services for industrial CPS within a cloud environment. IEEE Trans. Ind. Inform. **13**(3), 1249–1256 (2017)
34. Alladi, T., Chamola, V., Zeadally, S.: Industrial control systems: cyberattack trends and countermeasures. Comput. Commun. **155**, 1–8 (2020)
35. Argaw, S.T., Troncoso-Pastoriza, J.R., Lacey, D., Florin, M., Calcavecchia, F., Anderson, D., Burleson, W.P., Vogel, J., O'Leary, C., Eshaya-Chauvin, B., Flahault, A.: Cybersecurity of hospitals: discussing the challenges and working towards mitigating the risks. BMC Med. Inform. Decis. Mak. **20**(1), 146 (2020)
36. Cortellessa, V., Marco, A.D., Inverardi, P.: Model-Based Software Performance Analysis. Springer, Berlin (2011)
37. Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A.: Experimentation in Software Engineering. Springer, Berlin (2012)
38. Calinescu, R., Cortellessa, V., Stefanakos, I., Trubiani, C.: Analysis and refactoring of software systems using performance antipattern profiles. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE), pp. 357–377 (2020)
39. Wert, A., Happe, J., Happe, L.: Supporting swift reaction: automatically uncovering performance problems by systematic experiments. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 552–561 (2013)
40. Trubiani, C., Bran, A., van Hoorn, A., Avritzer, A., Knoche, H.: Exploiting load testing and profiling for performance antipattern detection. Inf. Softw. Technol. **95**, 329–345 (2018)
41. Chen, Z., Chen, B., Xiao, L., Wang, X., Chen, L., Liu, Y., Xu, B.: Speedoo: prioritizing performance optimization opportunities. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 811–821 (2018)
42. Palma, F., Moha, N., Guéhéneuc, Y.-G.: Unidosa: the unified specification and detection of service antipatterns. IEEE Trans. Softw. Eng. **45**(10), 1024–1053 (2018)
43. Feitosa, D., Ampatzoglou, A., Avgeriou, P., Chatzigeorgiou, A., Nakagawa, E.Y.: What can violations of good practices tell about the relationship between GoF patterns and run-time quality attributes? Inf. Softw. Technol. **105**, 1–16 (2019)
44. Moha, N., Gueheneuc, Y.-G., Duchien, L., Le Meur, A.-F.: Decor: a method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng. **36**(1), 20–36 (2009)
45. Cortellessa, V., Pompeo, D.D., Stoico, V., Tucci, M.: On the impact of Performance antipatterns in multi-objective software model refactoring optimization. In: Proceedings of the Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 224–233 (2021)
46. Aceto, L., Attard, D.P., Francalanza, A., Ingólfsdóttir, A.: On benchmarking for concurrent runtime verification. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE), vol. 12649, pp. 3–23 (2021)
47. Tavcar, J., Horváth, I.: A review of the principles of designing smart cyber-physical systems for run-time adaptation: learned lessons and open issues. IEEE Trans. Syst. Man Cybern. Syst. **49**(1), 145–158 (2019)
48. Nuzzo, P., Li, J., Sangiovanni-Vincentelli, A.L., Xi, Y., Li, D.: Stochastic assume-guarantee contracts for cyber-physical system design. ACM Trans. Embed. Comput. Syst. **18**(1), 2–1226 (2019)
49. Heinzemann, C., Becker, S., Volk, A.: Transactional execution of hierarchical reconfigurations in cyber-physical systems. Softw. Syst. Model. **18**(1), 157–189 (2019)
50. Larsen, K.G.: Validation, synthesis and optimization for cyber-physical systems. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), vol. 10205, pp. 3–20 (2017)
51. Bakirtzis, G., Sherburne, T., Adams, S.C., Horowitz, B.M., Beling, P.A., Fleming, C.H.: An ontological metamodel for cyber-physical system safety, security, and resilience coengineering. Softw. Syst. Model. **21**(1), 113–137 (2022)
52. Bondi, A.B.: Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice. Addison-Wesley Professional, Boston (2015)
53. Matalonga, S., Amalfitano, D., Doreste, A., Fasolino, A.R., Travassos, G.H.: Alternatives for testing of context-aware software

systems in non-academic settings: results from a rapid review. Inf. Softw. Technol. **149**, 106937 (2022)
54. Zhang, Z., Eyisi, E., Koutsoukos, X., Porter, J., Karsai, G., Sztipanovits, J.: A co-simulation framework for design of time-triggered automotive cyber physical systems. Simul. Model. Pract. Theory **43**, 16–33 (2014)
55. Mo, Y., Sinopoli, B.: On the performance degradation of cyber-physical systems under stealthy integrity attacks. IEEE Trans. Autom. Control **61**(9), 2618–2624 (2016)
56. Zhang, H., Shu, Y., Cheng, P., Chen, J.: Privacy and performance trade-off in cyber-physical systems. IEEE Netw. **30**(2), 62–66 (2016)
57. Chen, C., Liu, X., Qiu, T., Sangaiah, A.K.: A short-term traffic prediction model in the vehicular cyber-physical systems. Future Gener. Comput. Syst. **105**, 894–903 (2020)
58. Švogor, I., Crnković, I., Vrček, N.: An extensible framework for software configuration optimization on heterogeneous computing systems: time and energy case study. Inf. Softw. Technol. **105**, 30–42 (2019)
59. Gong, H., Li, R., An, J., Bai, Y., Li, K.: Quantitative modeling and analytical calculation of an elasticity for a cyber-physical system. IEEE Trans. Syst. Man Cybern. Syst. **50**(11), 4746–4761 (2020)
60. Alasmari, N., Calinescu, R., Paterson, C., Mirandola, R.: Quantitative verification with adaptive uncertainty reduction. J. Syst. Softw. **188**, 111275 (2022)
61. Gil, M., Albert, M., Fons, J., Pelechano, V.: Engineering human-in-the-loop interactions in cyber-physical systems. Inf. Softw. Technol. **126**, 106349 (2020)

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

Connie U. Smith CTO of L&S Computer Technology, Inc., is known for her work on defining the field of Software Performance Engineering (SPE), integrating SPE into the development of new software systems, and creating software performance antipatterns. She received a BA from University of Colorado and MA and Ph.D. degrees in computer science from the University of Texas at Austin. She is the author of the original SPE book, Performance Engineering of Software Systems, co-author of Performance Solutions: A Practical Guide to Building Responsive, Scalable Software, and approximately 100 scientific papers. In 1998, she initiated the first ACM Workshop on Software and Performance (WOSP) now part of the International Conference on Performance Engineering (ICPE). She recently led a research effort to develop tools to automate performance modeling of software and system designs with focus on CPS and real-time embedded systems. More information is at www.spe-ed.com.

Catia Trubiani is Associate Professor at the Gran Sasso Science Institute (GSSI), Italy. Previously she collaborated with the Karlsruhe Institute of Technology in Germany, and the Imperial College of London in UK. Her research interests include software performance modeling and analysis of large-scale and interacting heterogeneous distributed systems. Recently, she is leading a research effort funded by the MUR (Ministry of University and Research in Italy) to foster modular software designs and reduce uncertainties with a focus on cyber-physical systems. For more information, please visit https://cs.gssi.it/catia.trubiani.