Performance Modeling and Analysis of Design Patterns for Microservice Systems

Riccardo Pinciroli
Gran Sasso Science Institute
L'Aquila, Italy
riccardo.pinciroli@gssi.it

Aldeida Aleti
Monash University
Clayton, Australia
aldeida.aleti@monash.edu

Catia Trubiani
Gran Sasso Science Institute
L'Aquila, Italy
catia.trubiani@gssi.it

***Abstract*—The adoption of design patterns in the microservice architecture and cloud-native development scope was recently reviewed to investigate the industry practice. Interestingly, when considering performance-related aspects, practitioners focus on specific metrics (e.g., the time taken to handle requests) to identify sources of performance hindrance. This paper investigates a subset of seven design patterns that industrial practitioners indicate as relevant for system performance. We are interested to quantify the impact of these patterns while considering heterogeneous workloads, thus supporting software architects in understanding the root causes of performance issues. We use queuing networks to build the performance models of the seven design patterns and extract quantitative insights from model-based performance analysis. Our performance models are flexible in their input parameterization and reusable in different application contexts. We find that most design patterns confirm the expectation of practitioners, and our experimental results assess the identified performance gains and pains. One design pattern (i.e., Gateway Offloading) shows the peculiar characteristic of contributing to performance pains in some cases, leading to novel insights about the impact of design patterns in microservice systems.**

***Index Terms*—Software Architecture, Model-based Performance Analysis, Microservices, Design Patterns.**

I. Introduction

Microservices have grown increasingly popular in the last years due to their key advantage of breaking down software into modules that become independent processes interacting via lightweight mechanisms [1]–[3]. Cloud-native applications are commonly built as microservices by leveraging containers since they have a smaller footprint than a full virtual machine, thus isolating microservices and their dependencies from the underlying infrastructure [4]–[6]. This peculiar aspect of microservices, along with the promises of larger agility, scalability, maintainability, and performance [7], attracted major vendors, e.g., Netflix, Amazon, and Uber [8]. Their experience points out faster development and more flexible deployment of applications. This leads to cost savings and inspires researchers and practitioners to further investigate the benefits of the microservice architecture.

Design patterns find their roots in [9] where the main concepts of design principles have been sketched. The specification of design patterns has undergone many variations across the years, especially the relationship between usage of patterns and quality attributes [10]. More recently, Vale at al. [11] presented an empirical study on how industry experts perceive the impact of design patterns on quality attributes of microservice systems, focusing on their relevance. The authors state that there is a need for quantitative approaches that support the evaluation of design patterns, and this constitutes the motivation for our work. In this paper, we propose an approach to quantify the performance characteristics of design patterns, with the goal of understanding the system ability to meet performance requirements when managing heterogeneous workloads [12]. Performance is recognized as a prominent aspect of microservices [13], hence we extract the seven design patterns deemed in [11] as relevant to performance. Our research focuses on modeling and analyzing the performance characteristics of these patterns.

Software performance engineering techniques aim to produce performance models [14], such as Queueing Networks [15], Stochastic Petri Nets [16], and Markov Chains [17], early in the development cycle. These techniques are unexplored in the context of microservices [18], and *performance modeling* is a prominent research challenge due to many difficulties, e.g., finding appropriate modeling abstractions. This observation leads to formulating our first research question *RQ_1*: How can we model design patterns with Queuing Network performance models? We answer *RQ_1* by modeling seven design patterns, and we find some system characteristics triggering the adoption of hybrid performance models for one design pattern. This investigation confirms the difficulty of performance modeling and provides the first contribution in this area of research.

Performance models are beneficial for guiding software architects on the early detection and diagnosis of performance problems, and reducing the waste of resources [19]. The detection of performance issues in microservices is largely advocated in the literature [20]–[22], however it is mainly addressed with expensive monitoring techniques [23] or machine learning predictions [24]. On the other hand, *model-based performance analysis* [25] is a promising technique in this domain due to its flexibility in analyzing heterogeneous workloads, i.e., multiple and variegate requests. This motivates our second research question *RQ_2*: How does the performance of design patterns change with heterogeneous workloads? We answer *RQ_2* by varying the numerical values of input parameters in performance models and deriving metrics of interest, i.e., the system response time and resource utilization undergo a sensitivity analysis on the load or ratio of heterogeneous requests. This investigation demonstrates the relevance of considering variegate workloads and supports the software architects in understanding performance fluctuations.

In summary, the main contributions of this paper are: (i) we develop performance models of seven design patterns that industrial practitioners indicate as relevant to performance [11]; (ii) we perform a model-based performance analysis to support software architects in quantitatively understanding the performance characteristics of design patterns. Performance models and replication data are publicly available [26].

II. Preliminaries

Design Patterns. Our selection of design patterns is motivated by industry relevance. As in [11], we select patterns from the Azure Architecture Center (AAC), i.e., a repository of software architectures hosted by Microsoft. In particular, we investigate the seven design patterns that are indicated in [11] as relevant for *performance*. The remaining seven design patterns reported in [11] are excluded from our study since they are explicitly associated with other quality attributes, e.g., maintainability, reusability, or security rather than performance.

The detailed description of the selected seven patterns is reported in [27]. A brief description of their main performance implications on microservice architectures is as follows:

(i) *Anti-corruption Layer*, an adapter manages requests between different microservices. This can result in performance pain due to the overhead of translating requests;

(ii) *Backends for Frontends* avoids customizing a single backend microservice for multiple interfaces. This is a possible performance gain since it improves the autonomy and flexibility of the management w.r.t. generalized backends;

(iii) *Command and Query Responsibility Segregation*, i.e., separating read and update operations for a data store microservice. This can be a performance gain since it speeds up read and write requests in case of no data overlap;

(iv) *Gateway Aggregation*, a gateway aggregates requests to multiple microservices in a single inquiry. This can result in a performance gain since communication overhead is reduced;

(v) *Gateway Offloading* offloads common functionalities of multiple microservices to a proxy gateway. This is a performance gain since some tasks are offloaded to the gateway;

(vi) *Pipe and Filters* decomposes a complex task (i.e., single service) into separate elements (i.e., microservices) that can be independently managed. This might represent a performance gain due to the parallel computing of separate components;

(vii) *Static Content Hosting* deploys static content into cloud-based storage microservice. This is perceived as a performance gain since some requests are managed more rapidly.

Queuing Networks (QNs). This modeling formalism is widely applied to represent and analyze resource-sharing systems [15]. A QN model is composed of (i) interacting *service stations* representing system resources and (ii) *jobs* representing users' requests that share the resources. Service stations include at least a *server* and can be of two types: (i) processing stations with a *queue* (whose length can be finite or infinite) and a finite number of servers; (ii) delay stations without a queue and an infinite number of servers. Service stations are connected through *links* that form the network topology. Each server of a service station picks the next job from the queue (if not empty), processes it, and routes the job to another service station. The time spent in every server by each request (S) follows an exponential distribution [15]. A *fork-join* mechanism splits the jobs into several tasks that are executed in parallel. No service time is allocated for this operation, tasks are routed to the outgoing links of the fork station and they reconvene in the join station.

For the modeling purpose of this paper, we consider closed and heterogeneous workloads (i.e., multiple types of requests that iterate in the network), and each type of requests is specified through the number of users (N) and their thinking time (Z), i.e., the delay before re-iterating the request. The QN representation can be considered as a direct graph whose nodes are stations and their connections are represented by the graph edges. Jobs go through graph edges based on the required service.

III. Performance Models of Design Patterns

In this section, we describe the seven design patterns introduced by Vale et al. [11]; for each pattern, we discuss the problem it tackles and the proposed solution as presented in [27]. To answer *RQ$_1$*, we consider the architectural diagrams provided in [27], and we present the derived performance models. Hereinafter, we describe the rationale behind the performance models, along with the underlying assumptions.

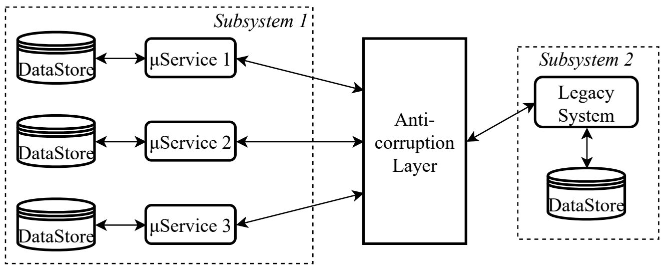
A. Anti-corruption Layer

Problem. Applications leverage different systems for their operations, which means that systems need to communicate to provide value to the end customer. For example, a (recently) migrated system might interact with legacy systems for function execution, or data warehousing tools may need relational or NoSQL databases for managing a large amount of data. It may happen that two systems are not compatible (e.g., due to the usage of different technologies for their development) or that legacy systems are affected by quality issues that threaten the efficiency of other systems.

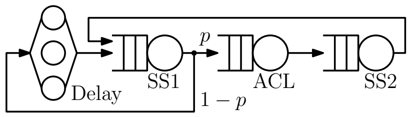
Solution. Systems should be isolated by means of an anti-corruption layer, which takes care of mediating and translating their communications. For example, let us consider the case of two systems (e.g., new and legacy) that need to interact. The anti-corruption layer translates all communications between the two systems and allows (i) the legacy system to remain unchanged and (ii) the new system to be developed using the most modern techniques and approaches.

Performance model. The architectural diagram representing the deployment of an anti-corruption layer between a microservice-based subsystem and a legacy one is shown in Fig. 1(a).

Starting from this representation of the considered pattern, we define the QN model presented in Fig. 1(b). Specifically,



(a) Architectural diagram [27]



(b) QN model

Fig. 1: Architectural diagram and performance model of the *Anti-corruption Layer* design pattern.

the two subsystems (i.e., `SS1` and `SS2`) and the anti-corruption layer (i.e., `ACL`) are modeled by three service stations with their service time that depends on the hardware/software implementation of the system.

For example, the performance of *Subsystem 1* (`SS1`) is affected by the number of microservices used for its deployment since a greater number of microservices generally requires more inter-service communication [11], whereas the performance of *Subsystem 2* (`SS2`) might depend on the hardware used for its set up. We model each subsystem as a service station to let software architects abstract from the implementation of their system and only consider the time required to process requests. We assume the analyzed system serves unbalanced requests, e.g., Fast and Slow requests which have different requirements. Specifically, Fast requests takes a small amount of time to be processed by the three services, whereas Slow ones need longer service times before being completed.

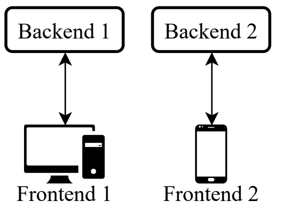
All requests (initialized in the `Delay` station) are processed at least once by `SS1`. After such processing, there is a probability p that requests need to execute some actions (e.g., retrieve data or invoke a function) on the legacy system (i.e., `SS2`). In this case, requests are first managed by the anti-corruption layer component, then they are served by `SS2`. Hence, requests go back to `SS1` where the processing is completed with probability $1 - p$ (i.e., the request returns to the `Delay` station) or repeated (i.e., the request visits `ACL` and `SS2` again for further execution).

B. Backends for Frontends

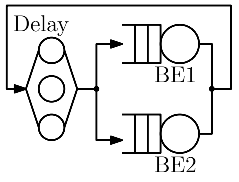
Problem. A single backend service processes heterogeneous requests, e.g., requests from Desktop and Mobile user-interfaces. Therefore, requests with different requirements and constraints compete for the same service, the single backend service needs to frequently change and adapt to the different requests. The backend service quickly becomes the bottleneck and the performance of the system deteriorates.

Solution. Heterogeneous requests should be served by different backends. This way, requests do not compete for the same service and the bottleneck is relaxed. Backends can also be optimized to serve a specific type of requests to further improve the system performance.

Performance model. Fig. 2(a) depicts the architectural diagram of the *Backends for Frontends* design pattern, where each frontend communicates only with the corresponding backend.



(a) Architectural diagram [27]



(b) QN model

Fig. 2: Architectural diagram and performance model of the *Backends for Frontends* design pattern.

The performance model in Fig. 2(b) considers two types of request (i.e., Desktop and Mobile requests) with different requirements. Only backends (i.e., `BE1` and `BE2`) are modeled as service stations, whereas the behavior of frontends is included in the `Delay` station since they are not the subject of this performance analysis. Since this design pattern suggests serving heterogeneous requests with different backends, Desktop requests are always routed to (and processed by) `BE1`, whereas Mobile requests are routed to `BE2`.

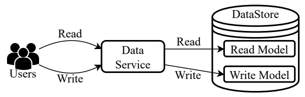
C. Command and Query Responsibility Segregation (CQRS)

Problem. Traditional architectures query and update a database (DB) using the same (software and hardware) model. While this may be efficient for simple applications, using this approach with more complex applications can result in a waste of resources due to different features (e.g., requirements and volume) of Read and Write requests.

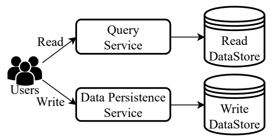
Solution. CQRS suggests using different models for reading and updating a DB to isolate the two operations. This allows simplifying the design and implementation of the DB. Isolation can be software or hardware. In the former case, queries and updates work on different data models located in the same DB. In the latter case, one can physically separate read data from write data (i.e., use different DBs). When physical separation is implemented, the two DBs must be kept synchronized, e.g., propagating updates from the write DB to the read one every time data are written/updated.

Performance model. Since *CQRS* can be implemented in two different ways, i.e., software and hardware, see Figs. 3(a)–(b), respectively, we define two performance models.

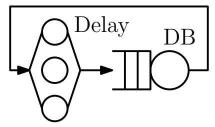
Software separation is modeled, see Fig. 3(c), as a single service (i.e., the `DB`) processing both Read and Write requests (i.e., the heterogeneous requests in the system). Once `DB` executes a request, the request returns to the `Delay` station where it spends some time before visiting `DB` again.



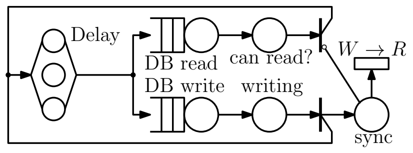
(a) Architectural diagram (SW separation) [27]



(b) Architectural diagram (HW separation) [27]



(c) QN model (SW separation)



(d) Multi-formalism (QN+PN) model (HW separation)

Fig. 3: Architectural diagram and performance model of the *CQRS* design pattern.

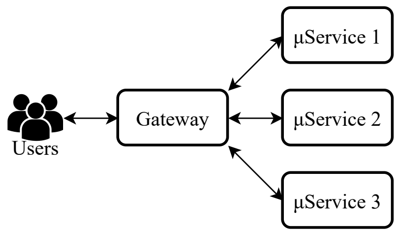
Hardware separation requires a mechanism that keeps the two DBs (i.e., the one for queries, `DB read`, and the one for updates, `DB write`) in sync when Read and Write requests are processed by `DB read` and `DB write`, respectively. To model such a mechanism, see Fig. 3(d), we need to extend our QN model using Petri Nets (PN) [16], a modeling abstraction suitable to handle synchronization among requests [28]. Specifically, when a Write request is processed by `DB write`, a token is generated into the `writing` place. Immediately (i.e., after zero time units, as indicated by the thin black transition between `writing` and `sync` places) a Write request is returned to the `Delay` station and a token is placed in the `sync` place. Such a token is removed from the `sync` place only when the synchronization is completed, i.e., when the transition $W \to R$ fires after an exponentially distributed time defined by the user. Tokens in the `sync` place preclude Read requests from being completed. Indeed, requests served by `DB read` return immediately to the `Delay` station only if the `sync` place is empty (see the inhibitor arc, i.e., the arc with a circle head connecting the `sync` place to the immediate transition). Otherwise, Read requests wait in the `can read?` place to be completed until there are no token in the `sync` place (i.e., when all updates are propagated).

D. Gateway Aggregation

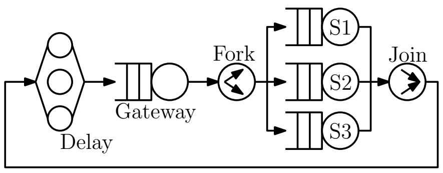
Problem. An application might need to interact with multiple services before being able to return the desired result to the end user. The number of interactions increases when new services or features are added to the application. The performance of the application is affected by the high number of interactions, and the current trend of resorting to smaller services makes such a problem even more relevant.

Solution. Interactions between the application and available services can be reduced by placing a gateway in the middle. This way, the performance of the application improves since the gateway takes care of interacting with the microservices, collecting their output, and returning aggregated results (i.e., reduced communication) to the application.

Performance model. All services considered by this design pattern, i.e., Gateway and microservices, see Fig.4(a), are modeled by a service station in the QN model, see Fig.4(b).



(a) Architectural diagram [27]



(b) QN model

Fig. 4: Architectural diagram and performance model of the *Gateway Aggregation* design pattern.

We assume these services are required to process heterogeneous requests, e.g., some requests spend a lot of time in `S3` (i.e., S3Intensive), while others wait a long time in `S1` (i.e., S1Intensive). When the `Gateway` receives a request, it determines which microservices need to be involved in the request processing (i.e., `S1`, `S2`, and `S3` in the example) and sends to each of them a sub-request. In the QN, this is modeled by means of a fork-join mechanism: (i) requests going through the `Fork` are split into sub-requests, one for each outgoing connection; (ii) sub-requests are merged back into the original request when reaching the `Join`. After being served by all the microservices, the request goes back to the `Delay` station where it waits some time before starting a new process.

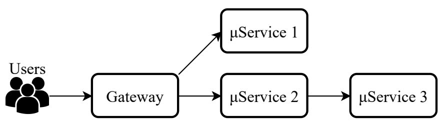
E. Gateway Offloading

Problem. Different services may need to include the same type of feature (e.g., encryption, authentication, authorization) in their operation pipeline. The requests processing is slowed down if the same operation is executed multiple times, e.g, two sequential microservices both needing to authenticate requests.

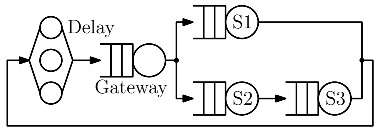
Solution. Features that are shared by all services should be deployed in a gateway. This way, the processing time of requests decreases since the gateway allows reducing the number of invocations to the common features.

Performance model. Fig. 5(a) shows the analyzed scenario, where users generate two types of request, e.g., Dashboard and Monitoring requests.

The former request type (i.e., processed by *µService 1*, `S1` in the QN) allows users to view their dashboard, the latter one (i.e., processed by *µService 2* and *µService 3*, `S2` and `S3` in the QN, respectively) allows users to profile some processes and download collected data. All these microservices (i.e., dashboard visualization, process profiling, and data download)



(a) Architectural diagram [27]



(b) QN model

Fig. 5: Architectural diagram and performance model of the *Gateway Offloading* design pattern.

require users to authenticate before executing the requests. When using the Gateway Offloading design pattern, the authentication feature is deployed in the Gateway since such a feature is invoked by all the involved microservices. The QN in Fig. 5(b) is obtained by mapping every service (i.e., gateway and microservices) with a service station. `Gateway` serves all requests, `S1` serves Dashboard requests, while `S2` and `S3` serve Monitoring requests.

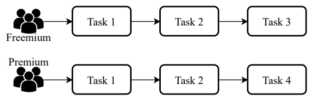
F. Pipes and Filters

Problem. An application needs to execute some tasks on requests generated from different sources (e.g., freemium and premium users). If the application is implemented as monolithic modules (i.e., one of them serving Freemium requests, the other processing Premium ones), different quality attributes of the system (including performance) experience deterioration.

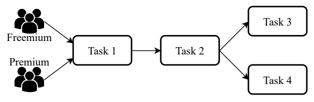
Solution. Code reuse should be facilitated to maximize the efficiency of an application. Specifically, each monolithic module should be broken down into a set of separated components, each one performing a different task. This way, tasks that are common to both request types can be reused.

Performance model. To model this design pattern, we consider two different implementations. The former, see Fig. 6(a) for its representation, is shown in [27]; such implementation replicates *Task 1* and *Task 2* that are used to process both types of requests, while uses *Task 3* and *Task 4* only for Freemium and Premium requests, respectively.

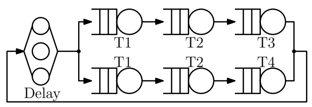
The QN model of this implementation of *Pipes and Filters* is shown in Fig. 6(c), where every task is represented by a service station. All Freemium requests are processed by the tasks on the top (i.e., `T1`, `T2`, and `T3`), while Premium requests are routed to and served by tasks on the bottom of the figure (i.e., `T1`, `T2`, and `T4`). We also envision and analyze another possible implementation of this design pattern, see Fig. 6(b). In this case, common tasks (i.e., *Task 1* and *Task 2*) use the same services and serve both request types. Therefore, in the QN model, see Fig. 6(d), all requests go through `T1` and `T2`. After being processed by `T2`, Freemium requests are served by `T3`, while `T4` executes Premium requests.



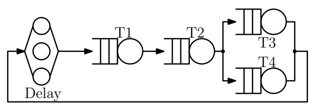
(a) Architectural diagram (separated services) [27]



(b) Architectural diagram (joint services)



(c) QN model (separated services)



(d) QN model (joint services)

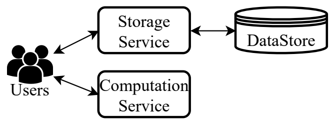
Fig. 6: Architectural diagram and performance model of the *Pipes and Filters* design pattern.

G. Static Content Hosting

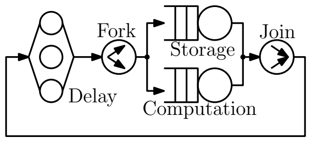
Problem. Requests might need to retrieve some static contents besides running computations. If a single service is in charge of computation and static content download, the performance of the application may deteriorate due to the excessive load of the service.

Solution. Static content should be placed in a storage service that is accessed by requests without the need for extra computation from other services. This way, the storage service can handle downloading requests and relax the load of other services that can focus on computation.

Performance model. A representation of this design pattern is shown in Fig. 7(a), where users' requests trigger two services, i.e., the *Computation Service*, which is in charge of all computations, and the *Storage Service* that contacts the storage device to retrieve static contents.



(a) Architectural diagram [27]



(b) QN model

Fig. 7: Architectural diagram and performance model of the *Static Content Hosting* design pattern.

The QN model is depicted in Fig. 7(b). Heterogeneous requests are considered, e.g., requests that need a lot of computation (i.e., CompIntensive) and requests whose computation is shorter, but need to download larger contents (i.e., StorIntensive). All requests are forked into sub-requests that are concurrently processed by `Storage` and `Computation` service stations. The `Storage` station models download operations, while the `Computation` one represents the calculation required by each request. Results are returned to the users when the computation is completed and static contents are downloaded.

Summarizing, the answer to *RQ$_1$* is as follows.

***RQ*$_1$: Performance Models of Design Patterns**

All design patterns that are identified to affect the performance of an application in [11] can be modeled using QN. The only exception is the *CQRS* design pattern (only hardware separation, i.e., DBs for read and write operations are physically separated). In that case, the QN model needs to be extended using PN to account for the synchronization of `DB read` when data in `DB write` is added or updated. Performance models also allow envisioning different implementations of the same pattern, as we do for *Pipes and Filters*.

IV. Model-based Performance Analysis

To answer *RQ*$_2$, we simulate the performance models described in Section III. We collect the system response time (R) and resource utilization (U) as output metrics. Due to the lack of space, we do not show the system throughput (X, i.e., another commonly used metric [29], [30]) since it can be easily retrieved from the response time and the number of users in the system (N) using the Little's law [15], i.e., $X = N/R$.

Simulations are performed using JSIMgraph [31] (i.e., the simulator of Java Modelling Tools, a widely used and largely validated framework for performance evaluation) and take 21.6 sec on average to converge. Simulations are executed on a commodity machine with an Intel Core i7-10750H CPU @ 2.60GHz and 16GB memory. The shortest simulation terminates after 3.3 sec, the longest one needs 250.5 sec (i.e., 4.2 min). For all output metrics, we show their average value and 99% confidence interval (i.e., lines and shaded areas, respectively). A replication package containing models, input data, and obtained results is publicly available [26].

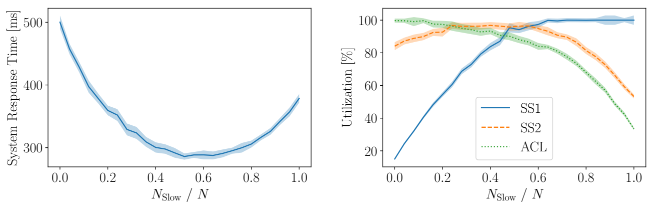
A. Anti-corruption Layer

Table I reports the input parameters of a microservice-based application with an anti-corruption layer (`ACL`) placed between two subsystems (`SS1` and `SS2`). All time values are in milliseconds (ms). Fig. 8 shows the performance analysis of the system. The average system response time (i.e., the time spent by a request into the system, or the time between the request generation and its completion) is depicted in Fig. 8(a) against the percentage of Slow requests in the system. We remind the reader that Slow requests take longer time than Fast ones to be processed by each service (i.e., `SS1`, `ACL`, and `SS2`), see Section III-A.

The percentage of Slow requests in the system is computed as N_{Slow}/N, where N is the total amount of requests in the system (e.g., $N = N_{\text{Slow}} + N_{\text{Fast}}$). Note that the ratio of Fast requests in the system when there are N_{Slow} requests is computed as $N_{\text{Fast}}/N = 1 - N_{\text{Slow}}/N$. Unlike what one would expect (i.e., the application is faster when only Fast requests are into the system), the average system response time is minimized when $N_{\text{Slow}}/N = 0.56$. The system performance deteriorates up to 80% and 34% when the application serves only Fast and Slow requests, respectively. Although surprising, this behavior is described in the literature [32] and is ascribable

TABLE I: *Anti-corruption Layer* – input parameters

Req. Type	Slow	Fast
N	[0, 25]	$25 - N_{\text{Slow}}$
Z	20	100
p	20%	80%
S_{SS1}	12	0.6
S_{ACL}	20	5
S_{SS2}	32	4.25



(a) Response time (b) Utilization

Fig. 8: *Anti-corruption Layer* – performance analysis

to different services becoming the *system bottleneck* (i.e., the service with the largest utilization [15]) when the ratio of heterogeneous requests in the system changes. To highlight the bottleneck switch experienced by our application, Fig. 8(b) depicts the utilization of the three services considered when analyzing this design pattern (i.e., `ACL`, `SS1`, and `SS2`). For $N_{\text{Slow}}/N \leq 0.28$, `ACL` is the system bottleneck since there are many Fast requests in the system which often (i.e., $p = 80\%$, see Table I) need to be served by `ACL` and interact with `SS2`. When $0.28 < N_{\text{Slow}}/N \leq 0.48$ the bottleneck is `SS2` due to a large number of Slow requests in the system that occasionally (i.e., $p = 20\%$) need to be served by `ACL` and `SS2`. The large percentage of Fast requests requiring service from `SS2` and the long time (i.e., 32 ms) spent by few Slow requests in the same service make `SS2` the bottleneck of the system. If $N_{\text{Slow}}/N > 0.48$, `SS1` becomes the bottleneck since Slow requests mainly require to be processed only by this service.

Architectural implications. When adopting this design pattern, software architects need to pay attention to heterogeneous requests to optimize the performance of their applications. Our parametrization shows that the ratio of unbalanced (e.g., Fast and Slow) requests affects the bottleneck and, consequently, the system response time.

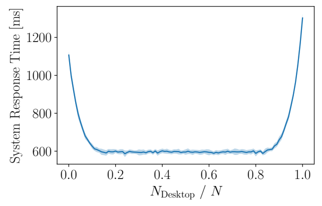
B. Backends for Frontends

The performance of an application, parameterized as in Table II (all time values are in ms), that serves incoming requests with two different backends (i.e., BE1 and BE2, as suggested by the considered design pattern), is shown in Fig. 9.

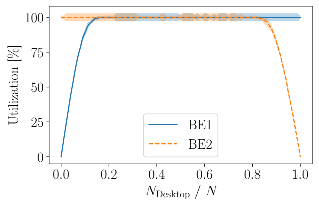
The average system response time of this application is depicted in Fig. 9(a) against the ratio of Desktop requests in the system. We assume Mobile requests are served faster (11 ms) than Desktop ones (13 ms) due to the considered design pattern (e.g., BE2 is optimized for serving Mobile requests). However, when the application serves a large number of Mobile requests (i.e., $N_{\text{Desktop}}/N \leq 0.1$), the system response time is up to 86% longer than its minimum value.

TABLE II: *Backends for Frontends* – input parameters

Req. Type	Desktop	Mobile
N	[0, 100]	$100 - N_{\text{Desktop}}$
Z	100	100
S_{B1}	13	–
S_{B2}	–	11



(a) Response time



(b) Utilization

Fig. 9: *Backends for Frontends* – performance analysis

A similar behavior is observed if the application executes a large number of Desktop requests (i.e., $N_{\text{Desktop}}/N \geq 0.9$) with the system response time being up to 119% longer than its minimum value. Fig. 9(b) shows that previous observations are due to the system bottleneck switching from BE2 to BE1. In this case, software architects have a large range of values to optimize the performance of their applications (i.e., $0.1 < N_{\text{Desktop}}/N < 0.9$) due to the two services (i.e., BE1 and BE2) saturating at the same time.

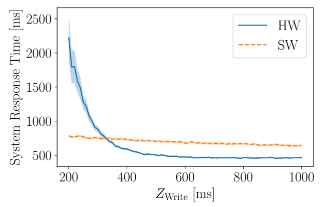
Architectural implications. Software architects should employ this design pattern when different sources (e.g., frontends) generate comparable amounts of requests. Experiments show that the system response time deteriorates when the application serves mainly requests from a single frontend.

C. *CQRS*

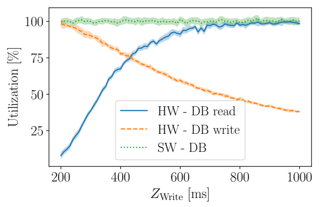
Fig. 10 depicts the performance of an application designed using the *CQRS* pattern whose parameters are shown in Table III (all time values are in ms). In Section III-C, we stress that this design pattern can be deployed with SW or HW separation.

TABLE III: *CQRS* – input parameters

Req. Type	Read	Write
N	90	10
Z	10	[200, 1000]
S_{DB}	5	40
$W \to R$	–	$S_{\text{DB,Write}}$



(a) Response time



(b) Utilization

Fig. 10: *CQRS* – performance analysis

When SW separation is preferred, the application is deployed with a single DB, see Fig. 3(a). In this case, effects of the bottleneck switch are not observed since Read and Write requests are served by a single service, see Fig. 3(c). However, performance models can be used to evaluate which implementation of *CQRS* (i.e., SW or HW) works better, e.g., when the frequency of Write requests (i.e., Z_{write}) varies. For this scenario, we consider a read-mostly workload, i.e., the Read:Write ratio is 9:1. Fig. 10(a) depicts the average system response time of SW and HW separations against Z_{write}. When Write requests are frequent in the system (i.e., $Z_{\text{write}} \leq 350$ ms), the best strategy from the performance perspective is to have a single DB with two different data models (i.e., SW separation), whereas HW separation is more convenient when Write requests are rare.

This is due to the time DBs spend syncing when HW separation is used. If many Write requests are in the system or they are issued too frequently, Read requests might wait a long time for an up-to-date version of the DB on which to perform their queries. When the number or frequency of Write requests is limited, sync operations are less common and Read requests can query the DB without waiting for synchronization. This allows observing a system response time that is 28% shorter than the one obtained using SW separation, where the only available DB needs to serve both Read and Write requests. Fig. 10(b) shows the utilization of all services (i.e., `DB read` and `DB write` for HW separation, `DB` for SW separation). While the service deployed for SW separation is always fully utilized, at least one of the services for HW separation is under-utilized leaving some space for optimizing the deployment of the application.

Architectural implications. When adopting this pattern, software architects need to carefully choose between SW and HW separation. Our parametrization shows that frequent Write requests deteriorate the performance of HW separated systems.

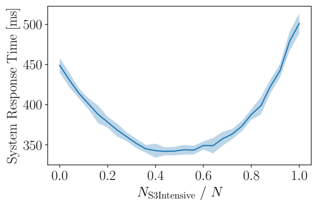
D. *Gateway Aggregation*

The performance of an application designed using this pattern is shown in Fig. 11, and input parameters are reported in Table IV (all time values are in ms).

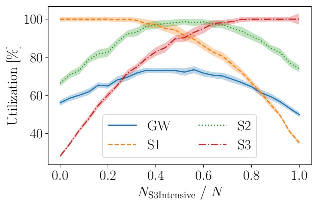
Multiple services (i.e., the gateway and 3 microservices) are involved in processing users' requests, and the system response time shows a non-monotonic trend due to the bottleneck switch. Fig. 11(a) depicts the average system response time when the percentage of S3Intensive requests in the system varies. The shortest response time is observed when $N_{\text{S3Intensive}}/N = 0.48$. The system performance deteriorates up to 31% and 46% when the number of S3Intensive and S1Intensive requests in the system decreases, respectively. Fig. 11(b) shows the services limiting the system performance (i.e., the system bottlenecks) in the considered case. `S1` is the bottleneck of the system when $N_{\text{S3Intensive}}/N \leq 0.36$, `S2` limits the system performance when $0.36 < N_{\text{S3Intensive}}/N \leq 0.64$, and `S3` is the most used service when $N_{\text{S3Intensive}}/N > 0.64$. The gateway is never the system bottleneck in this scenario.

TABLE IV: *Gateway Aggregation* – input parameters

Req. Type	S3Intensive	S1Intensive
N	[0, 25]	$25 - N_{\text{S3Intensive}}$
Z	100	100
S_{GW}	10	10
S_{S1}	7	18
S_{S2}	15	12
S_{S3}	20	5



(a) Response time



(b) Utilization

Fig. 11: *Gateway Aggregation* – performance analysis

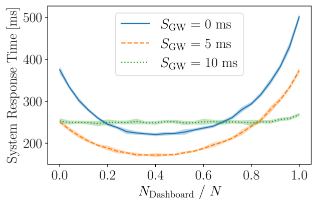
Architectural implications. Microservice-based applications that process heterogeneous workloads are inclined to bottleneck switch. When an application processes requests with different requirements, the service limiting the system performance (i.e., the bottleneck) changes with the ratio of heterogeneous requests. For this reason, software architects need to consider request requirements to optimize their applications.

E. Gateway Offloading

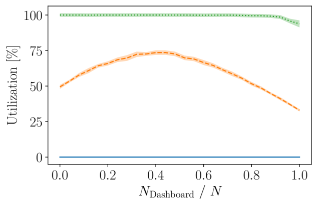
Performance models presented in Section III can be used by software architects to tune their application and choose the most efficient way to use a design pattern. This is observable when considering the performance of an application designed with the *Gateway Offloading* pattern, see Fig. 12.

TABLE V: *Gateway Offloading* – input parameters

Req. Type	Dashboard	Monitoring
N	[0, 25]	$25 - N_{\text{Dashboard}}$
Z	100	100
S_{GW}	[0, 10]	$S_{\text{GW,Dashboard}}$
S_{S1}	$20 - S_{\text{GW,Dashboard}}$	–
S_{S2}	–	$12 - S_{\text{GW,Monitoring}}$
S_{S3}	–	$15 - S_{\text{GW,Monitoring}}$



(a) Response Time



(b) Gateway Utilization

Fig. 12: *Gateway Offloading* – performance analysis

Specifically, here we try to identify which type of operation should be offloaded, i.e., how long requests should spend in the gateway. As shown in Table V (all time values are in ms), S_{GW} is the same for both request types (i.e., Dashboard and Monitoring), and the time a request spends in the gateway to execute the offloaded operation is taken away from other services, e.g., $S_{\text{S1}} = 20 - S_{\text{GW}}$ with 20 ms being the time spent in `S1` when no operations are offloaded to the gateway.

For our analysis, we consider three offloading strategies, i.e., no offloading ($S_{\text{GW}} = 0$ ms), offloading short operations ($S_{\text{GW}} = 5$ ms), and offloading long operations ($S_{\text{GW}} = 10$ ms). The effect of these three strategies on the average system response time is depicted in Fig. 12(a) against $N_{\text{Dashboard}}/N$. For the sake of readability, Fig. 12(b) shows only the gateway utilization when the three strategies are used, the utilization of other services (i.e., `S1`, `S2`, and `S3`) is omitted. If no operations are offloaded to the gateway, this service is not used (i.e., *Utilization* $= 0\%$), the bottleneck switches among other services and the system response time is U-shaped.

Similar observations are drawn when a short operation is offloaded. In our scenario, the utilization of the gateway service is between 33% and 75%, i.e., not enough to make the gateway the system bottleneck. Therefore, the bottleneck switches again among other services and the system response time is still U-shaped. If a long operation is offloaded, the gateway is always the bottleneck of the system since its utilization is 100% due to the long operation that it executes. In this case, there is not bottleneck switch and the system response time is flat when $N_{\text{Dashboard}}/N$ varies. These observations should influence software architects' decisions.

Depending on the ratio of Dashboard and Monitoring requests, it is preferable offloading short operations (i.e., $S_{\text{GW}} = 5$ ms) if $N_{\text{Dashboard}}/N \leq 0.84$, or long ones (i.e., $S_{\text{GW}} = 10$ ms) when $N_{\text{Dashboard}}/N > 0.84$ to minimize the response time of the application. Moreover, if short operations cannot be offloaded (e.g., they are not shared by all requests [27]), it might be more beneficial not using this design pattern, e.g., in our scenario the *no offloading* strategy performs better than *offloading long operations* when $0.2 \leq N_{\text{Dashboard}}/N \leq 0.64$.

Architectural implications. Software architects should wisely choose which operations to offload to the gateway. Differently from [11] in which this design pattern is perceived as a performance gain, our results show that offloading long operations may deteriorate the system performance to the extent of invalidating the usage of this pattern.

F. Pipes and Filters

The average system response time observed when applications (tuned as in Table VI where all time values are in ms) are designed using this pattern is depicted in Fig. 13 against the percentage of Freemium requests in the system.

For the sake of readability, the service utilization of this design pattern is omitted. The response time of the implementation suggested in [27] is labeled as *Separated* since services that process different requests are deployed multiple times, e.g., see *Task 1* and *Task 2* in Fig. 6(a). We label with *Joint* the response time of implementations where common tasks are executed by the same service, see Fig. 6(b). Specifically, we use *Joint (×1)* if only 1 CPU is allocated to each service that executes common tasks, while we use *Joint (×2)* if 2

TABLE VI: *Pipes and Filters* – input parameters

Req. Type	Freemium	Premium
N	[0, 50]	$50 - N_{\text{Freemium}}$
Z	100	100
S_{T1}	12	8
S_{T2}	15	9
S_{T3}	11	–
S_{T4}	–	10

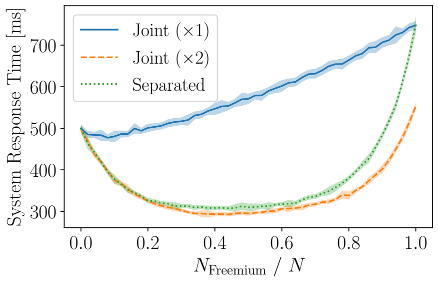


Fig. 13: System response time of an application designed using the *Pipes and Filters* pattern.

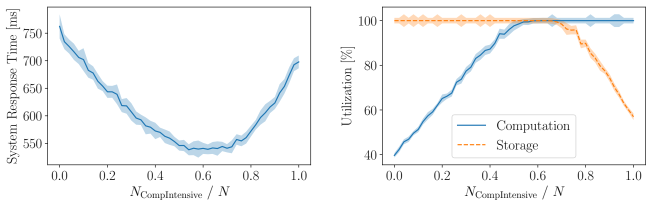
CPUs are allocated to these services. Results show that the *Joint (×1)* implementation does not perform as well as others, the response time observed in this case is always the longest one due to *Task 1* and *Task 2* processing all requests (i.e., Freemium and Premium) with limited power. *Separated* and *Joint (×2)* provides similar performance as long as some Premium requests are in the system, i.e., $N_{\text{Freemium}}/N \leq 0.7$. When the number of Freemium requests is prevalent, the *Joint (×2)* implementation performs up to 27% better than the *Separated* one. Software architects can use these performance models to determine which implementation of the *Pipes and Filters* pattern is more convenient for their applications.

Architectural implications. When adopting this design pattern, software architects need to be aware that using additional resources (e.g., services with 2 CPUs) might be beneficial when observing separate services showing bad performance.

G. *Static Content Hosting*

TABLE VII: *Static Content Hosting* – input parameters

Req. Type	CompIntensive	StorIntensive
N	[0, 50]	$50 - N_{\text{CompIntensive}}$
Z	100	100
S_{Comp}	14	6
S_{Stor}	8	15



(a) Response time (b) Utilization

Fig. 14: *Static Content Hosting* – performance analysis

The performance of this design pattern when the input parameters are as in Table VII (all time values are in ms) is shown in Fig. 14. The system response time is depicted as a function of $N_{\text{CompIntensive}}/N$, see Fig. 14(a). The bottleneck switching between Computation and Storage services is visible in Fig. 14(b) that shows the utilization of the two services. For $N_{\text{CompIntensive}}/N \leq 0.6$ the Storage service is the bottleneck of the system, whereas when $N_{\text{CompIntensive}}/N > 0.6$ the Computation service becomes the bottleneck. This affects the average system response time that shows a non-monotonic behavior and whose minimum value is up to 28% smaller than the maximum one.

Architectural implications. When adopting this design pattern, the load required for computation and storage determines the bottleneck switch. Software architects should consider re-implementing their applications to minimize computation and storage requirements (e.g., more efficient access to database).

In summary, the answer to *RQ*$_2$ is as follows:

***RQ*$_2$: Performance Analysis of Design Patterns**

The intrinsic characteristic of microservice applications which make use of multiple services to process heterogeneous requests can be a threat to the system performance due to the bottleneck switch. Our performance models can support software architects in determining the best pattern implementation (e.g., as shown for *CQRS* and *Pipes and Filters*) or understanding when a pattern provides a real advantage for the system performance (e.g., as discussed for *Gateway Offloading*).

V. Discussion

Here, we discuss the main limitations observed when modeling and analyzing the performance of design patterns. We also present threats to the validity of our approach.

A. *Performance Modeling*

The choice of using QN as the target notation for modeling the performance is motivated by the application of this formalism to many real-world systems, e.g., Industry 4.0 warehouse automation [33], Apache Cassandra [34], cloud applications [35]. This choice does not reduce the applicability of our approach. However as future work, we plan to experiment with further notations (e.g., Markov Chains [17]) to investigate the usability and scalability of other performance notations, as well as the modeling and analysis of other system attributes, e.g., dependability [36]. We are aware that performance models are not guaranteed to be correct since they are built on the basis of our interpretation of the specification of design patterns [27]. This is an open issue that we aim to investigate as part of our future research.

B. *Performance Analysis*

The choice of running simulations is motivated by our assumption that the performance analysis is conducted at design time. This means that performance-based analysis results are reported to support software architects in studying different design patterns and quantitatively understanding their performance behavior. We are confident that the results returned by JSIMgraph are accurate since the simulator accuracy is assessed in multiple domains [37], [38]. The validation of performance models calls for comparing simulation results with measurements from real-world systems. As part of our future work, we plan to measure the performance characteristics of microservice-based systems (e.g., TeaStore [39], TrainTicket [40], or Sock Shop [41]). This way, we can parameterize the proposed performance models and compare predicted results to measured values. At this stage, we aim to raise the attention of software architects toward possible performance fluctuations due to heterogeneous workloads and operational profiles.

C. Threats to validity

Besides inheriting all limitations of design patterns and software performance engineering research [42], [43], our approach exhibits the following threats to validity [44].

External validity, i.e., generalization of results, is not guaranteed, since our models have been formulated for seven design patterns only, and model-based performance results highlight the conditions (i.e., heterogeneous workloads and operational profiles) for bottleneck switches. To smooth these threats, we consider large variations in workloads, but it is still an open issue to demonstrate the applicability of design patterns to real-world applications. We plan to further investigate this point by experimenting the usage of design patterns in industrial applications, possibly from multiple domains.

Internal validity, i.e., the settings used for performance modeling and analysis, is tackled by designing experiments with the purpose of having a direct manipulation on the performance indices of interest. For instance, the seven design patterns share some input parameter values to avoid misleading effects that cannot be traced back to root causes. Setting numerical values of input parameters is indeed an open issue in the software performance engineering domain [45]. To improve this point, the service time of microservices might be derived with the introduction of a monitor that collects data for a certain time frame and produce some statistics. Besides, other parameters can be further detailed, e.g., different gateways may show diverse performance characteristics (e.g., the speed of aggregating data). We recall that performance models are publicly available [26] and software architects can easily change the numerical values of input parameters.

Construct validity, i.e., the statistical validity of the experimental results, is smoothed by setting that all simulations undergo a 99% confidence interval, thus to monitor the accuracy of presented numerical results.

VI. Related Work

The motivation for our work is supported by a large literature. Cloud design patterns and their quality attributes are widely used by industrial practitioners due to their relevance for microservice systems [46]. Di Francesco et al. [47] classify 103 studies on microservice architectures and observe that *performance* grows quickly in popularity and importance among microservice quality attributes. Nevertheless, architectures and patterns are generally described via informal languages, which makes modeling and analyzing their characteristics complex for researchers and practitioners. Wijerathna et al. [48] observe that *performance* is evaluated only at runtime, despite it being the main concern of architecture design patterns. The exponential growth of costs to fix errors at advanced stages of the project life-cycle [49] motivates the need for performance models that enable the early performance evaluation. As further confirmation of the previous assertion, other studies (e.g., Heinrich et al. [18]) indicate the performance modeling of microservice applications and their patterns as a research challenge and direction.

Li et al. [50] analyze 72 primary studies in a systematic literature review and identify *performance* as one of the most critical quality attributes when designing a microservice application. This further stresses the need for performance models to systematically study large-scale microservices. Vale et al. [11] interview practitioners at 9 different companies on which design patterns (belonging to the *Design and Implementation* category of Azure [27]) are generally adopted in real-world applications. Out of 14 design patterns considered in the survey and used by the interviewed practitioners, 7 affect the system performance. The authors highlight the need for empirical studies to evaluate the impact of design patterns on microservice quality attributes. Soldani et al. [51] analyze the industrial grey literature on microservice-based applications and observe that performance testing is the most challenging activity when developing these applications. Cortellessa et al. [52] present an approach to trace relationships between the performance monitored data of microservice-based systems and their architectural models, thus supporting the idea of exploiting models to evaluate the performance of microservices. These studies stress the importance of modeling and analyzing the performance of microservice applications in a formal manner, i.e., the principal aspect of our paper.

The closest related methodologies are listed hereafter. Khomh and Abtahizadeh [53] use a cloud-based application to investigate the performance (and energy consumption) of 6 design patterns. Differently from our investigation, the authors consider only one Azure *Design and Implementation* pattern (i.e., *Pipes and Filters*) and do not define any performance model. Akbulut and Perros [54] study the performance of 3 design patterns by deploying a microservice application on a private virtual environment. Despite the authors stressing that optimal design patterns depend on the considered scenario and organization needs, no performance models of the analyzed patterns are provided. Kousiouris [55] proposes the *Batch Request Aggregation* pattern to reduce the latency and monetary costs of cloud environments. A similar approach is successfully adopted in [56], [57] to process homogeneous and heterogeneous inference requests with the serverless paradigm. Amiri et al. [58] investigate the system reliability and performance trade-offs of 3 microservice architectures. They use a Bernoulli model for reliability and a statistical model for performance analysis. Design patterns modeled and analyzed in our paper are not considered by the authors.

Long et al. [59] empirically study the effect of the *Queue-based Load Leveling* pattern and assess its impact on the performance of a serverless application. However, the authors analyze only one design pattern and do not provide observations for different patterns. Ma et al. [60] develop a framework that detects abnormal services; it makes use of a correlation calibration mechanism to detect cloud design patterns and eliminate their negative effects by root cause analysis. Instead of monitoring the system performance, our models evaluate the system behavior at design time, i.e., in the early phases of the development cycle.

Summarizing, to the best of our knowledge, most of the approaches in the literature stress the importance of modeling and analyzing the performance of microservice applications in a formal manner. However, there is no work quantifying the performance implications of adopting design patterns and supporting software architects in this task, i.e., one of the main contributions of this paper.

VII. Conclusion

In this paper, we define performance models to analyze the characteristics of seven design patterns that can be used in microservice systems. The main findings are: (i) QN performance models are suitable abstractions, for only one pattern we leverage a hybrid model (borrowing PN modeling constructs) to handle synchronization among requests; (ii) model-based analysis results support software architects in understanding the impact of heterogeneous workloads on the performance characteristics of design patterns.

In future work, we plan to address all the limitations discussed as part of threats to validity. Besides, we are interested to further investigate the effectiveness of our models. Empirical studies involving industrial practitioners will allow evaluating how they perceive our models as support to quantify the impact of design patterns on the performance evaluation of microservice systems.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable feedback. This work has been partially funded by MIUR PRIN project 2017TWRCNB SEDUCE (Designing Spatially Distributed Cyber-Physical Systems under Uncertainty).

References

[1] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, “Microservices: Yesterday, Today, and Tomorrow,” in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[2] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, “Microservices,” *IEEE Software*, vol. 35, no. 3, pp. 96–100, 2018.

[3] A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera, and A. Sadovykh, *Microservices: Science and Engineering*. Springer, 2020.

[4] D. Liu, H. Zhu, C. Xu, I. Bayley, D. E. Lightfoot, M. Green, and P. Marshall, “CIDE: An Integrated Development Environment for Microservices,” in *Proceedings of the International Conference on Services Computing (SCC)*. IEEE, 2016, pp. 808–812.

[5] S. Taherizadeh and M. Grobelnik, “Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications,” *Advances in Engineering Software*, vol. 140, 2020.

[6] N. C. Mendonça, C. Box, C. Manolache, and L. Ryan, “The Monolith Strikes Back: Why Istio Migrated From Microservices to a Monolithic Architecture,” *IEEE Software*, vol. 38, no. 5, pp. 17–22, 2021.

[7] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, “Microservices: The Journey So Far and Challenges Ahead,” *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[8] A. Rud, “Why and How Netflix, Amazon, and Uber Migrated to Microservices: Learn from Their Experience,” https://web.archive.org/web/20230104141759/https://www.hys-enterprise.com/blog/why-and-how-netflix-amazon-and-uber-migrated-to-microservices-learn-from-their-experience/, 2019.

[9] R. C. Martin, “Design Principles and Design Patterns,” *Object Mentor*, vol. 1, no. 34, p. 597, 2000.

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

[11] G. Vale, F. F. Correia, E. M. Guerra, T. de Oliveira Rosa, J. Fritzsch, and J. Bogner, “Designing Microservice Systems Using Patterns: An Empirical Study on Quality Trade-Offs,” in *Proceedings of the International Conference on Software Architecture (ICSA)*. IEEE, 2022, pp. 69–79.

[12] X. Chang, R. Xia, J. K. Muppala, K. S. Trivedi, and J. Liu, “Effective Modeling Approach for IaaS Data Center Performance Analysis under Heterogeneous Workload,” *IEEE Transactions on Cloud Computing*, vol. 6, no. 4, pp. 991–1003, 2018.

[13] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, “Performance Evaluation of Microservices Architectures Using Containers,” in *Proceedings of the International Symposium on Network Computing and Applications (NCA)*. IEEE, 2015, pp. 27–34.

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

[15] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance - computer system analysis using queueing network models*. Prentice Hall, 1984.

[16] R. David and H. Alla, “Petri nets for modeling of dynamic systems: A survey,” *Automatica*, vol. 30, no. 2, pp. 175–202, 1994.

[17] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi, *Queueing Networks and Markov Chains - Modeling and Performance Evaluation with Computer Science Applications, Second Edition*. Wiley, 2006.

[18] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger, “Performance Engineering for Microservices: Research Challenges and Directions,” in *Proceedings of the International Conference on Performance Engineering (ICPE)*. ACM, 2017, pp. 223–226.

[19] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, “It’s not a Sprint, it’s a Marathon: Stretching Multi-resource Burstable Performance in Public Clouds,” in *Proceedings of the International Middleware Conference Industrial Track (Middleware)*. ACM, 2019, pp. 36–42.

[20] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, “MicroRCA: Root Cause Localization of Performance Issues in Microservices,” in *Proceedings of the Network Operations and Management Symposium (NOMS)*. IEEE, 2020, pp. 1–9.

[21] L. Meng, F. Ji, Y. Sun, and T. Wang, “Detecting anomalies in microservices with execution trace comparison,” *Future Generation Computer Systems*, vol. 116, pp. 291–301, 2021.

[22] R. K., P. Tammana, P. G. Kannan, and P. Naik, “A Case For Cross-Domain Observability to Debug Performance Issues in Microservices,” in *Proceedings of the International Conference on Cloud Computing (CLOUD)*. IEEE, 2022, pp. 244–246.

[23] R. Brondolin and M. D. Santambrogio, “A Black-box Monitoring Approach to Measure Microservices Runtime Performance,” *ACM Transactions on Architecture and Code Optimization*, vol. 17, no. 4, pp. 34:1–34:26, 2020.

[24] L. Wu, J. Bogatinovski, S. Nedelkoski, J. Tordsson, and O. Kao, “Performance Diagnosis in Cloud Microservices Using Deep Learning,” in *AIOps – Workshop of the Conference on Service-Oriented Computing (ICSOC)*, ser. Lecture Notes in Computer Science, vol. 12632. Springer, 2020, pp. 85–96.

[25] V. Cortellessa, A. D. Marco, and P. Inverardi, *Model-Based Software Performance Analysis*. Springer, 2011.

[26] R. Pinciroli, A. Aleti, and C. Trubiani, "Replication Package: Performance Modeling and Analysis of Design Patterns for Microservice Systems," https://doi.org/10.5281/zenodo.7503633, 2023.

[27] Microsoft Learn, "Cloud Design Patterns," https://web.archive.org/web/20221129210159/https://learn.microsoft.com/en-us/azure/architecture/patterns/, 2022.

[28] I. Grobelna, R. Wisniewski, M. Grobelny, and M. Wisniewska, "Design and Verification of Real-Life Processes With Application of Petri Nets," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 47, no. 11, pp. 2856–2869, 2017.

[29] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software architecture optimization methods: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 658–683, 2012.

[30] D. Falessi, M. A. Babar, G. Cantone, and P. Kruchten, "Applying empirical software engineering to software architecture: challenges and lessons learned," *Empirical Software Engineering*, vol. 15, no. 3, pp. 250–276, 2010.

[31] M. Bertoli, G. Casale, and G. Serazzi, "JMT: performance engineering tools for system modeling," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 10–15, 2009.

[32] E. Rosti, F. Schiavoni, and G. Serazzi, "Queueing network models with two classes of customers," in *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 1997, pp. 229–234.

[33] A. Kattepur, "Towards Structured Performance Analysis of Industry 4.0 Workflow Automation Resources," in *Proceedings of the International Conference on Performance Engineering (ICPE)*. ACM, 2019, pp. 189–196.

[34] S. Dipietro, G. Casale, and G. Serazzi, "A Queueing Network Model for Performance Prediction of Apache Cassandra," in *Proceedings of the International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS)*. ACM, 2016.

[35] G. Casale, J. F. Pérez, and W. Wang, "QD-AMVA: Evaluating systems with queue-dependent service requirements," *Performance Evaluation*, vol. 91, pp. 80–98, 2015.

[36] R. Pinciroli, K. S. Trivedi, and A. Bobbio, "Parametric Sensitivity and Uncertainty Propagation in Dependability Models," in *Proceedings of the International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS)*. ACM, 2016.

[37] P. Z. Sotenga, K. Djouani, and A. M. Kurien, "A virtual network model for gateway media access control virtualisation in large scale internet of things," *Internet of Things*, p. 100668, 2022.

[38] A. Kumar, R. Krishnamurthi, A. Nayyar, K. Sharma, V. Grover, and E. Hossain, "A novel smart healthcare design, simulation, and implementation using healthcare 4.0 processes," *IEEE Access*, vol. 8, pp. 118 433–118 471, 2020.

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

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

[41] Weaveworks, Inc., "Sock Shop: A Microservices Demo Application," https://web.archive.org/web/20230112131538/https://microservices-demo.github.io/, 2017.

[42] C. Zhang and D. Budgen, "What Do We Know about the Effectiveness of Software Design Patterns?" *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1213–1231, 2012.

[43] C. M. Woodside, G. Franks, and D. C. Petriu, "The Future of Software Performance Engineering," in *Proceedings of the International Workshop on Future of Software Engineering (FSE)*. IEEE Computer Society, 2007, pp. 171–187.

[44] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell, *Experimentation in Software Engineering*. Springer, 2012.

[45] A. B. Bondi, *Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice*. Pearson Education, 2014.

[46] T. B. Sousa, H. S. Ferreira, and F. F. Correia, "A Survey on the Adoption of Patterns for Engineering Software for the Cloud," *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 2128–2140, 2022.

[47] P. Di Francesco, P. Lago, and I. Malavolta, "Architecting with microservices: A systematic mapping study," *Journal of Systems and Software*, vol. 150, pp. 77–97, 2019.

[48] L. Wijerathna, A. Aleti, T. Bi, and A. Tang, "Mining and relating design contexts and design patterns from Stack Overflow," *Empirical Software Engineering*, vol. 27, no. 1, pp. 8:1–8:53, 2022.

[49] B. Haskins, J. Stecklein, B. Dick, G. Moroney, R. Lovell, and J. Dabney, "Error Cost Escalation Through the Project Life Cycle," in *Proceedings of the Annual International Symposium*. INCOSE, 2004, pp. 1723–1737.

[50] S. Li, H. Zhang, Z. Jia, C. Zhong, C. Zhang, Z. Shan, J. Shen, and M. A. Babar, "Understanding and addressing quality attributes of microservices architecture: A Systematic literature review," *Information and Software Technology*, vol. 131, p. 106449, 2021.

[51] J. Soldani, D. A. Tamburri, and W. van den Heuvel, "The pains and gains of microservices: A Systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215–232, 2018.

[52] V. Cortellessa, D. D. Pompeo, R. Eramo, and M. Tucci, "A model-driven approach for continuous performance engineering in microservice-based systems," *Journal of Systems and Software*, vol. 183, p. 111084, 2022.

[53] F. Khomh and S. A. Abtahizadeh, "Understanding the impact of cloud patterns on performance and energy consumption," *Journal of Systems and Software*, vol. 141, pp. 151–170, 2018.

[54] A. Akbulut and H. G. Perros, "Performance Analysis of Microservice Design Patterns," *IEEE Internet Computing*, vol. 23, no. 6, pp. 19–27, 2019.

[55] G. Kousiouris, "A self-adaptive batch request aggregation pattern for improving resource management, response time and costs in microservice and serverless environments," in *Proceedings of the International Performance, Computing, and Communications Conference (IPCCC)*. IEEE, 2021, pp. 1–10.

[56] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: machine learning inference serving on serverless platforms with adaptive batching," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE/ACM, 2020, pp. 69:1–69:15.

[57] ——, "Optimizing Inference Serving on Serverless Platforms," *Proceedings of the VLDB Endowment*, vol. 15, no. 10, pp. 2071–2084, 2022.

[58] A. Amiri, U. Zdun, and A. V. Hoorn, "Modeling and Empirical Validation of Reliability and Performance Trade-Offs of Dynamic Routing in Service- and Cloud-Based Architectures," *IEEE Transactions on Services Computing*, 2021, (Early Access).

[59] K. L. Ngo, J. Mukherjee, Z. M. Jiang, and M. Litoiu, "Evaluating the Scalability and Elasticity of Function as a Service Platform," in *Proceedings of the International Conference on Performance Engineering (ICPE)*. ACM, 2022, pp. 117–124.

[60] M. Ma, W. Lin, D. Pan, and P. Wang, "ServiceRank: Root Cause Identification of Anomaly in Large-Scale Microservice Architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 5, pp. 3087–3100, 2022.