




# Data-driven Understanding of Design Decisions in Pattern-based Microservices Architectures

J. Andres Diaz-Pace<sup>1</sup>, Catia Trubiani<sup>2</sup>, and David Garlan<sup>3</sup>

<sup>1</sup> ISISTAN Research Institute, CONICET and UNICEN University,  
Tandil, Buenos Aires, Argentina

<sup>2</sup> Gran Sasso Science Institute  
L'Aquila, Italy

<sup>3</sup> Software and Societal Systems Department, Carnegie Mellon University  
Pittsburgh, PA, USA

`andres.diazpace@isistan.unicen.edu.ar`, `catia.trubiani@gssi.it`,  
`garlan@cs.cmu.edu`

**Abstract.** The adoption of architectural patterns has recently been assessed in relation to their impact on the performance of microservice-based applications. For example, offloading common functionalities of multiple microservices to a gateway may lead to a system response time improvement. However, for a given system requirement, e.g., the latency of services or the resources' utilization, the benefit of choosing an architectural pattern is not guaranteed. Therefore, it becomes important to collect data about the parameters that contribute to the effective use of patterns, thus understanding the relationships between design decisions and performance requirements. In this work, we propose a data-driven approach to assess the quantitative impact of design decisions for a given pattern on the achievement of performance tradeoffs. Furthermore, our approach seeks to control the pattern parameters that cause variations (i.e., sensitivity) in performance tradeoffs. Starting from a dataset including parameters related to three microservices patterns (i.e., Gateway Offloading, Command and Query Responsibility Segregation, and Anti-corruption Layer) and their performance characteristics, we do apply machine learning techniques (i.e., PRIM and CART) to infer constraints on the parameter values that help to understand and reduce the performance sensitivity of pattern configurations. Our results support architects in making informed decisions about tradeoffs by providing insights on the parameters related to the behavior of microservices patterns.

**Keywords:** Data-driven Sensitivity Analysis · Design Decisions · Architectural Patterns · Microservices · Performance metrics

## 1 Introduction

Microservice-based applications have gained the attention of researchers and practitioners [27,6], and also the software architecture community has been attracted to pursue methodologies to support their successful development [1,16].

Although the adoption of patterns has a long history [13], their effective use and their impact on quality attributes are a more recent research trend [19,21].

Software architects developing microservice-based systems, especially in industry, have pointed out the need to quantitatively assess the effects of patterns on quality-based requirements [22]. For instance, focusing on performance-related characteristics such as system response time or resource utilization [11], it is important to understand how they lead to different tradeoffs depending on specific pattern configurations and related design assumptions.

To enable a quantitative analysis, a recent work [15] has shown that well-known microservices patterns can be modeled and evaluated through software performance engineering (SPE) techniques, e.g., queueing networks [10]. Interestingly, the SPE models show that the performance properties of the patterns are sensitive to design decisions and parameters. Furthermore, the variability in the values of such parameters has an effect on the performance tradeoffs achievable by the patterns. These parameters might refer to environmental conditions (e.g., incoming request rate) or design decisions of the solution (e.g., implementing a software or hardware isolation mechanism for a certain pattern). This scenario raises the notion of *sensitivity* of a design decision (belonging to a pattern), and how to exploit certain parameter ranges that allow a system to achieve desired performance properties [24]. For instance, in a microservices pattern, the heterogeneity of requests and their frequencies may influence the design decision to achieve a good tradeoff involving utilization and response time [15].

Motivated by the scenario above, we delve into the variability of the parameters that characterize microservice-based architectural patterns so that controlling for certain parameters can make the pattern behavior (or design model) less prone to performance variations. That is, our goal is to reduce the performance sensitivity of pattern configurations. We follow a data-driven approach to profile the space of configuration alternatives for a given model (e.g., a pattern), identify the most influential model parameters and performance tradeoffs, and run a sensitivity analysis with respect to the tradeoffs. Furthermore, given a target tradeoff (e.g., fast response time and average resource utilization), we provide a procedure that infers constraints on the parameter values to ensure that the tradeoff is met. To do so, we rely on machine learning (ML) techniques for scenario discovery, such as the patient rule induction method (PRIM) [7] and classification/regression trees (CART) [4].

We perform an initial evaluation of three microservice patterns [22]: (i) *Anti-corruption Layer*, (ii) *Command Query Responsibility Segregation*, and (iii) *Gateway Offloading*, and studied the effects of parameter variations on performance metrics, before and after applying parameter constraints. Our experimental results demonstrate that the proposed approach infers specific ranges of parameter values for the microservices configuration that many times reduce its performance variations, thus, achieving the desired tradeoff. Our approach provides knowledge to software architects on performance sensitivity due to key parameters of a pattern model, thus supporting them in making informed decisions.

The remainder of the paper is organized as follows. Section 2 provides background knowledge on the adopted architectural patterns and ML algorithms. Section 3 describes the proposed methodology along with a motivating example that clarifies the goal of this paper. Section 4 briefly introduces the experimental setup, followed by the evaluation that is discussed in Section 5. Related work is presented in Section 6, thus positioning our contribution in the state-of-the-art approaches. Section 7 outlines concluding remarks and future work.

## 2 Background

This section provides knowledge on foundational concepts, discussing the characteristics of the selected microservices patterns and the adopted ML algorithms.

### 2.1 Architectural Patterns

From a performance-based perspective, the selected microservices patterns [18] include a set of relevant input parameters that have been evaluated in [15], and we got inspiration from these parameters to build our dataset for this work. Table 1 lists the parameters considered in one of the patterns we analyze. Note that there are some common parameters that are replicated for all patterns, such as the number of requests in the system ( $N$ ), and the thinking time, i.e., the idle time before a new system request is issued ( $Z$ ). Next, we explain the specific parameters of the three patterns.

*Gateway Offloading (GO).* The problem occurs when different services require the same functionality (e.g., encryption) in their pipeline, thus causing a backlog of requests. The solution involves offloading shared or specialized service functionalities to a gateway proxy that manages them more efficiently, thus preventing service slowdowns. The architectural diagram [18] consists of four main components: three services (i.e.,  $S_1$ ,  $S_2$ , and  $S_3$ ) and the gateway (i.e.,  $GW$ ) that hosts common services. Table 1 shows that the performance of this pattern is affected by the service times of all components, i.e.,  $S_{GW}$ ,  $S_{S_1}$ ,  $S_{S_2}$ , and  $S_{S_3}$ .

*Command and Query Responsibility Segregation (CQRS).* The problem originates from traditional architectures that query and update the same (software or hardware) resource. As a solution, operations that read data are segregated from operations that update data. This can be performed at the software level (i.e., read and write requests use separate interfaces, but they are located on the same machine), or at the hardware level (i.e., having two different machines). The architectural diagram [18] differs in case of a software or hardware solution. The former includes a database component (i.e.,  $DB$ ) only, whereas the latter has two database components (i.e., one for queries,  $DB-read$ , and one for updates,  $DB-write$ ) that require synchronization. The pattern’s performance is affected by the service times of the  $DB$  components ( $S_{DB}$ ,  $S_{DB-read}$ , and  $S_{DB-write}$ ), and by the time needed to synchronize read and write requests ( $R \leftrightarrow W$ ).

*Anti-corruption Layer (ACL).* The problem arises when an application leverages different systems for its operation, e.g., a (recently) migrated application

<i>Pattern</i>	<i>Parameters</i>	<i>Description</i>
<i>GO</i>	$N$	Total amount of requests in the system
	$Z$	Thinking time
	$S_{GW}$	Service time of <i>GW</i> component
	$S_{S1}$	Service time of service $S_1$
	$S_{S2}$	Service time of service $S_2$
	$S_{S3}$	Service time of service $S_3$

Table 1: Gateway Offloading and its Performance-related Parameters

needs to interact with a legacy system that makes use of technologies not compatible with the primary application. This pattern is also helpful when it is necessary to isolate domain models. The solution introduces an adaptation layer that mediates the communication between a (modern) application and a legacy system. The architectural diagram [18] consists of three main components: two subsystems (i.e., **SS1** and **SS2**) and the anti-corruption layer (i.e., **ACL**) that acts as adapter and mediates the communications between the two subsystems. The pattern’s performance is affected by the probability of invoking the ACL component (*Prob*), and by the service times of the three involved components, i.e.,  $S_{SS1}$ ,  $S_{ACL}$ , and  $S_{SS2}$ .

## 2.2 Scenario Discovery Algorithms

In the following, we briefly describe two algorithms for scenario discovery (SD) [5], which is a type of ML technique used for finding regions of interest in a highly-dimensional dataset containing inputs and outputs for a model.

*PRIM (Patient Rule Induction Method)*. It is a bump-hunting algorithm [7] that searches for regions (bumps) in the input space with relatively high (or low) values for a target variable. In our dataset, this variable takes Boolean values depending on whether a performance requirement is met. PRIM describes the regions by simple rules, as they are rectangles called *boxes* in the input space. PRIM works by slowly reducing the data size by small amounts iteratively. First, candidate boxes are generated. Each box removes a data portion based on the levels of a single input variable. This stage is known as top-down peeling. Second, for each candidate box, the relative improvement in the number of outputs (i.e., performance metrics) inside the box is calculated, and the candidate box with the highest improvement is selected. Third, the data in the selected box replace the starting data, and the process is repeated. There is also a second stage, known as bottom-up pasting, which is the inverse of the peeling stage. The process ends based on stopping criteria (e.g., the current box is too small). PRIM seeks for regions having both a high density of positive instances (i.e., those satisfying the target property) and a good coverage of the space being analyzed.

*CART (Classification And Regression Trees)*. It is a decision tree algorithm [4] that recursively splits the data into subsets based on the values of input variables, until creating a tree-like structure for predicting a target variable. Similarly to PRIM, we assume that the target variable is a Boolean property, and use CART

to address a classification problem. CART creates a binary decision tree, in which nodes are split into sub-nodes based on a threshold value of an input variable. The root node is considered as the initial set and split into two subsets by considering the best input and threshold value. These subsets are split using the same logic. This process continues until the last pure subset is found in the tree or the maximum number of leaves is reached. Node splitting relies on the Gini impurity criterion [4], which measures the probability of misclassifying a random instance from a subset labeled according to the majority class. The lower the Gini impurity, the more pure a subset is. CART evaluates all possible splits and selects the one that best reduces the impurity of the subsets. Pruning is used to remove nodes that contribute little to classifier accuracy. In general, CART is faster than PRIM, although it might not identify the same rules for a dataset.

### 3 Approach

We propose a data-driven framework to support the architect’s understanding of the outcomes of certain design decisions and assumptions on performance properties of a microservices pattern. In particular, we identify the pattern parameters that contribute the most to performance variability and infer constraints for those parameters to reduce the performance sensitivity, providing an envelope of (desired) pattern behavior. Our framework is a processing pipeline as outlined in Fig. 1. The pipeline comprises four phases: pattern modeling, data collection, sensitivity assessment of pattern parameters, and inference of parameter constraints. Constraints are computed using either the PRIM or CART algorithms.

#### 3.1 Processing pipeline

The process begins with the specification (or model) of a pattern  $M$ . Our framework does not prescribe a specification formalism, so different performance models can be employed as long as performance metrics from the pattern behavior under different conditions can be gathered. In particular, the specification must distinguish the available design decisions for  $M$  (i.e., input parameters), the numeric domains for the parameters, and the performance metrics of interest.

Once  $M$  is chosen, the data collection phase refers to the generation of alternative instances in terms of different pattern configurations and different parameter values for each configuration. These instances define the so-called *configuration space*. The number of instances to sample is determined by the architect. Each instance is evaluated by a performance model, which simulates how each pattern configuration behaves under given parameter values (e.g., different workload variations) and computes a set of metrics (e.g., response time, utilization). The metric values define the *quality-attribute space* for the configuration space.

To facilitate the treatment of the quality-attribute space, we apply a discretization procedure that bins the values for each performance metric according to an ordinal (or Likert) scale. For instance, for response time, we use a 3-point scale  $< fast, average, slow >$  which converts the metric numeric values into

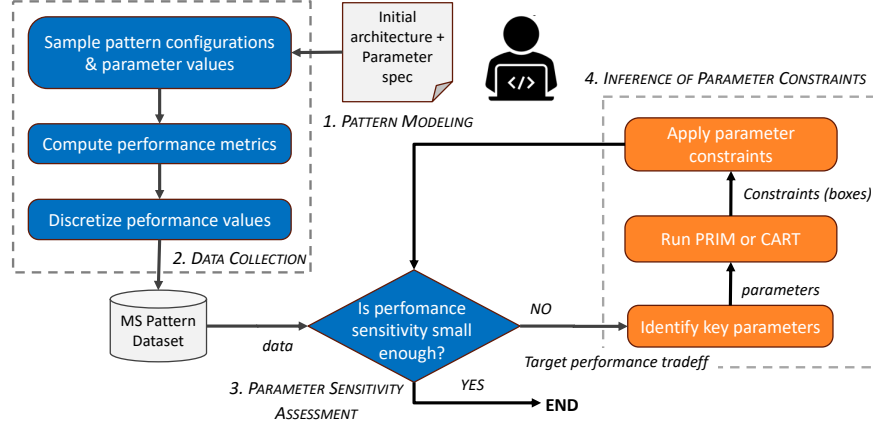


Fig. 1: Main phases of the proposed framework for analyzing and acting on the performance sensitivity of microservices patterns.

categorical ones. Any instance is mapped to a label that results from the concatenation of the categorical values for the metrics. This way, the instances are grouped in the quality-attribute space according to the tradeoffs of their labels. These tradeoffs capture quality-attribute properties for performance sensitivity analysis. Both the configuration and the (discretized) quality-attribute spaces are stored in a dataset. It is worth remarking that decisions and input parameters co-exist in the generated data. This implies that a certain design decision (e.g., no-offloading) will be mapped to a region of the space (quadrant) when exposed to a certain workload, whereas it may belong to a different region when considering another workload intensity value.

In the third phase, called parameter sensitivity assessment, the software architect chooses a target tradeoff (e.g.,  $\langle \text{average}, \text{fast} \rangle$  for utilization and response time, respectively) and determines whether the performance variability of the  $M$  configurations, with unconstrained decisions and parameters, is good enough for her preferences. If the performance variability is too high, then parameter constraints must be imposed to reduce the pattern sensitivity.

Fourth, the constraint inference phase seeks to control for the uncertainty in the parameter values, and thus reduce the variability of the performance metrics. We rely on PRIM and CART for scenario discovery (SD). In our context, SD refers to the identification of conditions that cause a set of instances (for a target configuration) to satisfy a given property with a bound variability. For simplicity, these properties correspond to the regions defined by the quality-attribute labels. Nonetheless, other types of quality-attribute properties could be formulated. To reduce the number of parameters to be taken by SD algorithms, a sensitivity analysis filter can be applied so that only the most relevant parameters are processed by the SD algorithm. To this end, we implement a correlation analysis of the parameters with respect to the performance metrics, and retain those

parameters with high correlation values. Finally, the architect can either accept the parameter constraints for the pattern configurations, or try a different design decision and repeat the process for the third and fourth phases.

### 3.2 Motivating Example

As an example of how our approach works, let us consider the *Gateway Offloading (GO)* pattern [15] introduced in Section 2.1. A *GO* instantiation for three services and two pipelines is illustrated in Fig. 2. In this example, let us assume a model with two request types, *request A* and *request B*, which need to be served by the system, one being executed by the first pipeline (*service1*) and another one being executed by the second pipeline (*service2* and *service3*). Let us also assume that the three microservices require the requests to be decrypted before executing them. When using the *GO* pattern, the encryption operation is deployed in the *gateway* component.

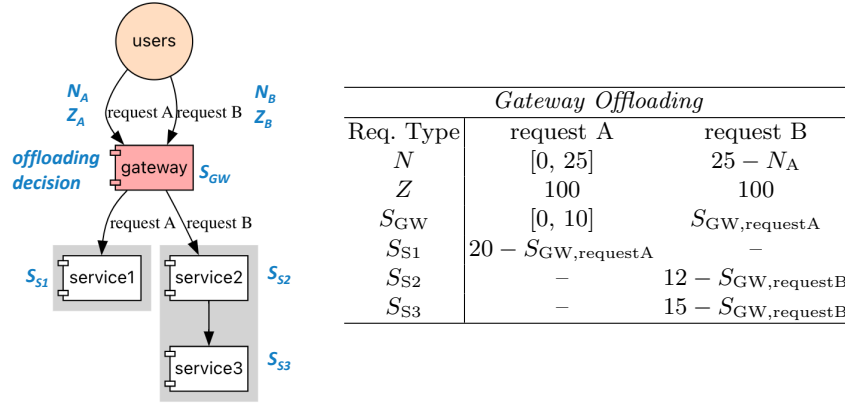


Fig. 2: Example of a *Gateway Offloading* pattern with performance parameters.

The *GO* performance depends on the operations being offloaded, as the time a request spends in the gateway to execute the offloaded operation is taken away from the other services. Thus, the offloading strategy becomes a *design decision* for the pattern. Based on [15], let us consider three alternatives for this decision, namely: *no offloading*, *offloading short operations* (e.g., those with execution times up to 5ms), or *offloading long operations* (e.g., execution times between 5ms and 10ms). As performance metrics, let us assume that architects are interested in response time and utilization at the gateway. To estimate these metrics, the architect can make assumptions about the behavior of the gateway and microservices when processing requests, and also about the system environment. These *assumptions* are captured by parameters, such as the number of requests ( $N_A$ ,  $N_B$ ), the time before requests are issued ( $Z_A$ , and  $Z_B$ ), and the execution

times for each component ( $S_{GW}$ ,  $S_{S1}$ ,  $S_{S2}$  and  $S_{S3}$ ). Based on these parameters, queuing networks (QNs) [10] can be applied to estimate performance metrics. By using the numerical values provided in [15] we can see that tradeoffs exist between utilization and response time for the three alternatives, as depicted in Fig. 3. For simplicity, we discretize the values for response time and utilization into three regions, each one exposing a feasible tradeoff category for the problem. In the general case, the quality-attribute space can be partitioned in multiple regions, depending on the variability of the metrics under analysis or the architect's needs.

A software architect might be interested in *GO* configurations with average or high utilization and average or fast response time. In the quality-attribute space in Fig. 3, we can see the performance variations in the configurations due to possible design decisions and assumptions. In this context, a design investigation for the architect is whether constraints can be imposed on the assumptions (or on the decisions) to reduce performance variability and thus, increase the chances that a pattern configuration fulfills a given tradeoff. This effort focuses on the *insensitivity* of a configuration, or a group of related configurations. In other words, our approach aims to understand how sensitive the performance of a configuration is to variations in its assumptions or design decisions. For instance, in the *GO* space in Fig. 3, the decision of off-loading short services seems to be mostly insensitive regarding the  $\langle \text{fast}, \text{average} \rangle$  tradeoff for utilization and response time, although it might deviate towards configurations with slower response time. The no-offloading decision shows a considerable sensitivity for the  $\langle \text{fast}, \text{low} \rangle$  tradeoff, while off-loading long services is always insensitive around the  $\langle \text{fast}, \text{high} \rangle$  tradeoff.

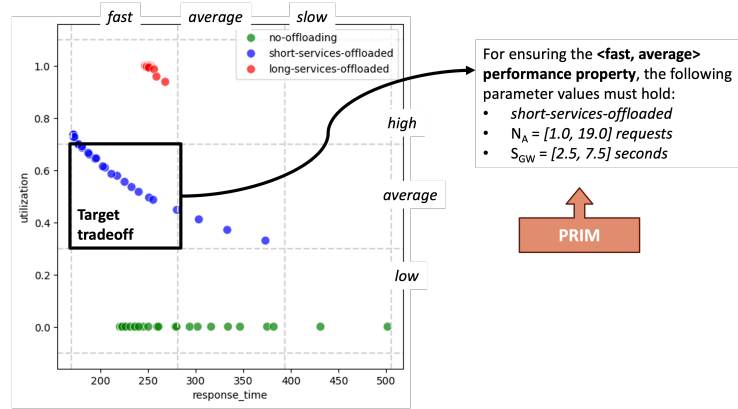


Fig. 3: Tradeoffs between response time and utilization for different configurations of the *Gateway Offloading* pattern. On the right, the constraints imposed (via PRIM) for reducing the sensitivity of achieving a target tradeoff are shown.



Let us consider that the architect wants to ensure both average utilization and fast response time as her target tradeoff. Thus, she needs to select an appropriate decision (e.g., off-loading short services) and also avoid deflections due to parameter variability. In particular, she can try to bound relevant parameters, such as  $N_A$ ,  $N_B$ ,  $S_{GW}$ , in such a way the *GO* configurations fall within the  $< fast, average >$  region. This effect can be achieved with  $N_A = [1.0, 19.0]$  and  $S_{GW} = [2.5, 7.5]$ . These constraints express rules for controlling the parameter values. The architect can rely on these rules to understand the design decisions related to the behavior of the *GO* pattern. For instance, she can realize that the gateway service time  $S_{GW}$  is instrumental in the pattern performance, but also environmental factors play a role, e.g., the rate of incoming requests of *request A* ( $N_A$ ) influences the pattern response. However, finding a set of constraints that ensure insensitive configurations with respect to a quality-attribute property is not straightforward. Here is where SD algorithms become useful.

## 4 Experimental Setup

We evaluate our approach by exercising it on three microservices patterns and computing different performance metrics for the pattern configurations. The experiments involve the patterns *Gateway Offloading (GO)*, *Command Query Responsibility Segregation (CRQS)* and *Anticorruption Layer (ACL)*, as modeled in [15]. We aim at addressing the following research questions:

- **RQ#1:** Which design decisions are more effective for satisfying a performance tradeoff?
- **RQ#2:** For a given design decision, how do PRIM and CART help to reduce sensitivity with respect to a performance tradeoff?

According to the processing pipeline of Fig. 1, we initially generate a dataset per pattern by executing its corresponding QN model and then visually inspect the distribution of the pattern configurations with respect to the tradeoffs. Two performance metrics are discretized into three equally-spaced ranges of values, leading to these tradeoff categories: *fast*, *average*, and *slow* for response time, and *high*, *average* and *low* for utilization. These results mainly target **RQ#1**.

For assessing the PRIM and CART algorithms, we split the pattern datasets into training and test sets (70/30) using stratification on the tradeoff labels, as typically done in ML. We remove outliers from the datasets via the z-score method ( $z = 3$ ). We additionally perform a feature analysis via the F-statistic to retain the most relevant parameters (features) with respect to utilization and response time (targets). These parameters were later fed into the algorithms. In PRIM, we apply the algorithm on the training set for every combination of (feasible) tradeoff label and design decision, and record the parameter rules (boxes). In CART, we consider each design decision, as the resulting tree generates the parameter rules for all the tradeoff labels at once.

For answering **RQ#2**, all the boxes obtained from both algorithms are used for evaluation on the test set. To quantify the effects of imposing the parameter

rules, we compute the percentage of configurations that satisfy each performance property (i.e., tradeoff label) before and after applying the boxes. We refer to this percentage as *density score*. We also analyze differences in the scores for the boxes generated by each algorithm to assess performance sensitivity improvements.

## 5 Evaluation

Fig. 4 shows the distributions of quality-attribute tradeoffs for the different configurations. There are three alternative decisions for the *GO* pattern, two decisions for the *CQRS* pattern, and one decision for the *ACL* pattern. Each quadrant refers to a different performance requirement (for the corresponding pattern) involving a tradeoff between utilization and response time. As depicted in Fig. 4, not all the tradeoffs were feasible in our sampled architectural space.

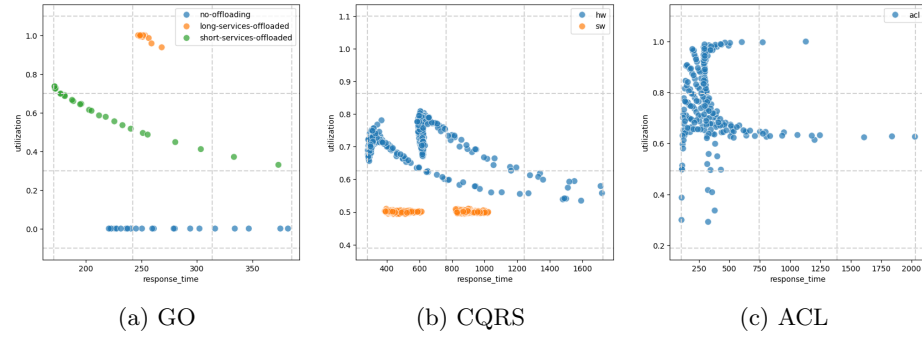


Fig. 4: Distribution of configurations with respect to the performance tradeoffs.

**Tradeoff analysis of *GO* pattern.** Fig. 4a shows that the decision of no-offloading always results in a low system utilization, and the response time can fluctuate between fast and slow depending on the parameter values. This scenario indicates that the architect should apply parameter constraints to ensure a good response time. In turn, the decision of offloading only long services always leads to a high utilization and an average response time, and thus seems to be an adequate decision for a reasonable performance (if a high utilization is not problematic for the system). When the goal is to optimize response time, the decision of offloading short services gives average-to-fast response time with an average utilization. Nonetheless, parameter constraints are necessary to avoid deflections towards configurations with slow latency.

**Tradeoff analysis of *CQRS* pattern.** Figure 4b exhibits a similar trend as the *GO* pattern with respect to trading off average-to-low utilization and average-to-fast response time. If the software decision is chosen, the system utilization will be always low, and the architect could apply additional constraints to opt between an average or fast response time. The hardware decision is preferred for obtaining a high utilization, but the response time might degrade

for some parameter values in the configurations. In this scenario, the architect should control for the parameter values to keep the response time fast or average (i.e., avoid the bottom-right quadrant).

**Tradeoff analysis of *ACL* pattern.** Fig. 4c differs from the previous ones, since here all performance variations are due to parameter values. The best configurations have average-to-high utilization with mostly fast response time. However, deflections towards low utilization or slower response times are observed. To avoid those scenarios, the architect should apply appropriate constraints on the parameter values, especially on the frequency of despatching requests to the ACL component.

Summarizing, Fig. 4 visualizes the pros and cons of each microservice pattern, and how decisions could be taken to achieve specific performance tradeoffs, thus answering **RQ#1**. Furthermore, the analyses expose the sensitivity of certain pattern configurations, depending on the tradeoff targeted by the architect.

Based on the previous datasets, we ran PRIM and CART for the feasible configurations, design decisions, and tradeoffs, in order to obtain a spectrum of boxes and density scores. The results are summarized in Tables 2, 3, and 4. Performance tradeoff is represented by *response time* (fast, average, slow), and *utilization* (high, average, low). Please note that test sets represent the baselines, i.e., without applying any algorithm. PRIM and CART produce density score values after applying the boxes for different performance tradeoffs. The values are in bold (and green), underlined (and yellow) or framed (and red) to highlight whether the change was positive, marginal or negative, respectively, in terms of configurations meeting the target tradeoff after imposing the constraints.

**Constraints for *GO* pattern.** After splitting the dataset (78 instances), we build the PRIM and CART models using the training set ( $\approx 45$  instances). Out of 8 possible parameters,  $N_A$ ,  $N_B$ , and  $S_{GW}$  turned out to be the most relevant ones for performance sensitivity of the pattern in both algorithms. Table 2 summarizes the density metrics obtained from the experiments. The first block of rows (test set) shows the initial scores for different tradeoffs, in which the most prevalent tradeoffs ( $\langle fast, low \rangle$ ,  $\langle average, high \rangle$ ,  $\langle fast, average \rangle$ ) depend on the design decision implemented at the gateway. For instance, the first row in the table indicates that with no-offloading 50% of solutions belong to the  $\langle fast, low \rangle$  tradeoff, 30% falls in the  $\langle average, low \rangle$  quadrant, whereas 20% relates to the  $\langle slow, low \rangle$  tradeoff. When PRIM is applied, for instance, in the sixth row for the *SSO* decision and  $\langle fast, high \rangle$  as the target tradeoff, 100% of the constrained configurations belong to that tradeoff quadrant. The parameter constraints for this case are  $N_A = [8.0, 12.0]$  and  $S_{GW} = [2.5, 7.5]$ . The constraints for all the cases are provided in the reproducibility package.

Both algorithms were able to effectively constraint certain parameters for ensuring the desirable tradeoffs of Fig. 4a, namely:  $\langle fast, low \rangle$  (no-offloading),  $\langle fast, average \rangle$  and  $\langle fast, high \rangle$  (short-services offloading). On one hand, PRIM was able to address more cases than CART, although it was not effective at improving all of them. On the other hand, the cases addressed by CART were limited but all of them were improved. CART outperformed PRIM for

GO	Decision	Performance tradeoff	fast, low	fast, avg	fast, high	avg, low	avg, avg	avg, high	slow, low	slow, avg	slow, high
Test set	NO		<b>0.50</b>			<b>0.30</b>			<b>0.20</b>		
	LSO							<b>1.00</b>			
	SSO			<b>0.67</b>	<b>0.11</b>		<b>0.11</b>			<b>0.11</b>	
PRIM	SSO	avg, avg		0.75	0.12		<b>0.13</b>				
		fast, avg		<b>0.83</b>	0.17						
		fast, high			<b>1.00</b>						
		slow, avg					1.00			<b>0.00</b>	
	LSO	avg, high						<b>1.00</b>			
		avg, low	0.83			<b>0.17</b>					
	NO	fast, low	<b>1.00</b>								
		slow, low	0.50			0.30			<b>0.20</b>		
		fast, high		0.50	<b>0.50</b>						
CART	SSO	avg, avg		0.33			<b>0.33</b>			0.33	
		fast, avg		<b>1.00</b>							
		fast, low	<b>1.00</b>								
	NO	fast, low	<b>1.00</b>								

Table 2: GO pattern’s results. Decisions are: no-offloading (NO), short-services-offloaded (SSO), long-services-offloaded (LSO). Average is abbreviated with avg.

the  $\langle \text{fast}, \text{average} \rangle$  (1.00 versus 0.83 w.r.t. 0.67 as the initial score) and  $\langle \text{average}, \text{average} \rangle$  (0.33 versus 0.13 w.r.t. 0.11 as the initial score) targets under the decision of short-services offloading. PRIM did better than CART for the  $\langle \text{fast}, \text{high} \rangle$  target (1.0 versus 0.5 w.r.t. 0.11 as the initial score). The differences can be attributed to the internal working of the algorithms, which returned different parameter ranges even for the same targets. In CART, we observed that improving the  $\langle \text{average}, \text{average} \rangle$  and  $\langle \text{fast}, \text{high} \rangle$  targets (with short-services offloading) had a positive side-effect on the  $\langle \text{fast}, \text{average} \rangle$  target, which is the most prevalent tradeoff in the space.

**Constraints for CQRS pattern.** The PRIM and CART models are built from the training set (382 instances) and their rules are applied to the test set (256 instances). The most prevalent tradeoffs are  $\langle \text{fast}, \text{average} \rangle$  for the hardware decision and  $\langle \text{fast}, \text{low} \rangle$  and  $\langle \text{average}, \text{low} \rangle$  for the software alternative. Coincidentally, both models identify the frequency of write requests in the database ( $Z_{write}$ ) as the key parameter for performance sensitivity of the pattern. Furthermore, both models are in agreement suggesting a lower frequency for the parameter that relates to the writing requests.

As noted from the density metrics in Table 3, both PRIM and CART obtain maximal effectiveness at improving the  $\langle \text{fast}, \text{average} \rangle$  target, which is a desirable target with the hardware decision, according to Fig. 4b. CART is very limited and could not address other tradeoffs. However, PRIM successfully addresses the  $\langle \text{average}, \text{average} \rangle$  (0.2 w.r.t. 0.05 as the initial score) and  $\langle \text{slow}, \text{low} \rangle$  (0.56 w.r.t. 0.06 as the initial score) targets for the hardware decision, even when those tradeoffs are not prevalent in the dataset. This result

<b>CQRS</b>	<i>Decision</i>	<i>Performance tradeoff</i>	<i>fast, low</i>	<i>fast, avg</i>	<i>fast, high</i>	<i>avg, low</i>	<i>avg, avg</i>	<i>avg, high</i>	<i>slow, low</i>	<i>slow, avg</i>	<i>slow, high</i>
Test set	hw		0.04	<b>0.84</b>		0.02	<b>0.05</b>		<b>0.06</b>		
	sw		<b>0.44</b>			<b>0.56</b>					
PRIM	hw	<i>avg, avg</i>	0.17	0.43		0.03	<b>0.20</b>		0.17		
		<i>avg, low</i>	0.05	0.81		<b>0.02</b>	0.06		0.07		
		<i>fast, avg</i>		<b>1.00</b>							
		<i>slow, low</i>	0.22			0.11	0.11		<b>0.56</b>		
	sw	<i>avg, avg</i>	0.52			0.48	<b>0.00</b>				
		<i>avg, low</i>	0.46			<b>0.54</b>					
		<i>fast, avg</i>	0.41	<b>0.00</b>		0.59					
		<i>slow, low</i>	0.43			0.57			<b>0.00</b>		
CART	hw	<i>fast, avg</i>		<b>1.00</b>							
	sw	<i>fast, avg</i>	0.45	<b>0.00</b>		0.55					

Table 3: CQRS pattern’s results. Decisions are: hardware separation (hw) and software separation (sw). Average is abbreviated with avg.

<b>ACL</b>	<i>Decision</i>	<i>Performance tradeoff</i>	<i>fast, low</i>	<i>fast, avg</i>	<i>fast, high</i>	<i>avg, low</i>	<i>avg, avg</i>	<i>avg, high</i>	<i>slow, low</i>	<i>slow, avg</i>	<i>slow, high</i>
Test set	acl		0.02	<b>0.48</b>	<b>0.39</b>		<b>0.09</b>	0.01		0.01	
PRIM	acl	<i>fast, avg</i>		<b>1.00</b>							
	acl	<i>fast, high</i>		0.04	<b>0.96</b>						
CART	acl	<i>fast, avg</i>		<b>0.73</b>			0.20			0.07	
	acl	<i>fast, high</i>		0.10	<b>0.87</b>			0.03			

Table 4: ACL pattern’s results. Decision consists of the frequency of invoking the ACL component. Average is abbreviated with avg.

highlights an interesting feature of PRIM that is not observed in CART, as it mostly detects prevalent tradeoffs.

**Constraints for ACL pattern.** The pattern dataset was split (234 instances) in order to build the PRIM and CART models on the training set ( $\approx 138$  instances). The most prevalent tradeoffs are  $\langle \text{fast}, \text{average} \rangle$  and  $\langle \text{average}, \text{high} \rangle$ , which are also desirable targets, according to Fig. 4c.

The corresponding rules are evaluated on the test set, and both yield gains in the density scores for the target tradeoffs above, with a slight score increment for PRIM over CART. The parameters  $N_A$  and  $N_B$  are shared by the two algorithms as being relevant for performance sensitivity, with more or less the same ranges of values. The reason is that these parameters relate to the workload of requests, thus affecting the performance metrics. However, PRIM also returned  $p_A$  as a relevant parameter, while CART returned  $1-p_B$  for the same role. These two parameters relate to the probability of invoking the ACL component that plays a key role for this pattern. In PRIM, the suggested ranges for  $p_A$  were below 0.5 for  $\langle \text{fast}, \text{average} \rangle$  and above 0.7 for  $\langle \text{fast}, \text{high} \rangle$ . In CART instead the value range for  $1-p_B$  was below 0.5 for both tradeoffs.

Summarizing, these analysis results answer **RQ#2**. In general, the score improvements for all the patterns are most noticeable in those quality-attribute tradeoffs encompassing many configurations in the dataset, i.e., tradeoffs with a reasonable coverage of solutions in the space. Both PRIM and CART work well for certain under-represented tradeoffs for the *GO* and *CQRS* patterns.

## 6 Related work

Microservices are increasingly popular due to their promises of agility, scalability, maintainability, and performance [8], which have even attracted major vendors, e.g., Netflix [20]. Recent studies have remarked on the importance of evaluating the performance of microservices, e.g., [27,6,1] to mention a few recent ones. In the following, we discuss the main approaches related to the aspects tackled in this paper, acknowledging that our selection of works is not exhaustive.

*Performance evaluation.* Di Francesco et al. [17] collect 103 studies on microservice architectures, and performance is observed to grow in popularity and relevance among microservice quality attributes. Li et al. [11] review 72 studies and outline performance as one of critical attributes when designing a microservices application. Wijerathna et al. [23] show that performance is often evaluated during the latest stages of the development cycles, hence fixing issues becomes expensive, so it is instead preferable to incorporate performance-based knowledge from the architectural phase. All these studies highlight the importance of performance analysis in microservices architectures, thus endorsing our research.

*Architectural patterns.* Khomh et al. [9] study a cloud-based application along with six design patterns that influence the system performance and energy consumption, but there is no sensitivity analysis in the derived findings. Akbulut et al. [2] evaluate the performance of three design patterns, and measurements reveal that the patterns perform better or worse depending on the different scenarios they are applied to. Amiri et al. [3] leverage the adoption of architectural patterns to explore performance and reliability tradeoffs. Ma et al. [12] present a correlation mechanism to detect cloud design patterns showing an anomalous performance behavior. Long et al. [14] assess the usage of one design pattern that aims to balance the incoming load of requests, thus showing a positive impact on the performance of a serverless application. Overall, these studies evaluate the impact of applying architectural patterns, but they do not quantitatively compare the design decisions along with the correlated sensitivity analysis, as we pursue in this paper.

*Behavioral robustness.* The notion of sensitivity is related to robustness, as recently renewed by Zhang et al. [25]. Robustness is defined as the largest set of deviating environmental behaviors under which the system is capable of guaranteeing a desired property. This concept evolved into a robustification objective [26], i.e., how to improve the robustness of a design (while minimizing the cost of architectural changes) against potential deviations. A tool-based approach is proposed in [24] where repairs are synthesized from solving a multi-objective

problem that minimizes the amount of behaviors and the costs of design modifications. Our research is motivated by controlling the possible deviations in design patterns’ parameters with respect to performance properties. This is achieved through PRIM and CART that reduce the variability of parameters.

In summary, to the best of our knowledge, there are several approaches that investigated the performance behavior of pattern-based microservices architectures. Our novelty relies on inferring rules to support architects in understanding and controlling pattern variability while achieving performance tradeoffs.

## 7 Conclusion and future work

This paper proposes a data-driven analysis of architectural patterns applied in the context of microservices applications. Our results provide insights to software architects since they are informed on the key parameters that affect the adoption of architectural patterns and their impact on performance tradeoffs, thus contributing to the quantitative evaluation of microservices applications. In this work, we rely on Queuing Networks as analytical models for performance, since we exploit the dataset provided by Pincioli et al. [15] that includes different parameter ranges and reports interesting performance insights. However, our approach is not tied to any performance modeling formalism, its primary goal is to collect data about patterns and their performance indicators as the basis for sensitivity analysis and constraint inference.

As future work, we plan to analyze a larger set of architectural patterns and their combinations, and to consider other quality indicators, such as reliability and security. We are interested in deriving design decisions with associated tradeoffs so that software architects have a wider understanding of the alternative decisions at their disposal. We also plan to assess the actual adoption of architectural patterns in industrial case studies, thus collecting further data to study the variability of pattern parameters.

## 8 Data Availability

We provide a reproducibility package with the scripts and datasets used in our experiments at: <https://github.com/andresdp/performance-sensitivity-patterns>

## 9 Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback. This work has been partially funded by the MUR-PRIN project 20228FT78M DREAM, MUR Department of Excellence 2023 - 2027 for GSSI, and PNRR ECS00000041 VITALITY. Also, the work has been partially supported by PICT-2021-00757 project, Argentina.

## References

1. Ahmad, H., Treude, C., Wagner, M., Szabo, C.: Smart HPA: A Resource-Efficient Horizontal Pod Auto-scaler for Microservice Architectures. In: Proc. of the International Conference on Software Architectures (ICSA) (2024), to appear
2. Akbulut, A., Perros, H.G.: Performance Analysis of Microservice Design Patterns. *IEEE Internet Computing* **23**(6), 19–27 (2019)
3. Amiri, A., Zdun, U., Hoorn, A.V.: Modeling and Empirical Validation of Reliability and Performance Trade-Offs of Dynamic Routing in Service- and Cloud-Based Architectures. *IEEE Transactions on Services Computing* **15**(6), 3372–3386 (2021)
4. Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and regression trees (1984)
5. Bryant, B.P., Lempert, R.J.: Thinking inside the box: A participatory, computer-assisted approach to scenario discovery. *Technological Forecasting and Social Change* **77**(1), 34–49 (2010)
6. Di, P., Liu, B., Gao, Y.: MicroFuzz: An Efficient Fuzzing Framework for Microservices. In: Proc. of the International Conference on Software Engineering (ICSE) – Software Engineering in Practice Track (2024), to appear
7. Friedman, J.H., Fisher, N.I.: Bump hunting in high-dimensional data. *Statistics and computing* **9**(2), 123–143 (1999)
8. Jamshidi, P., Pahl, C., Mendonça, N.C., Lewis, J., Tilkov, S.: Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* **35**(3), 24–35 (2018)
9. Khomh, F., Abtahizadeh, S.A.: Understanding the impact of cloud patterns on performance and energy consumption. *Journal of Systems and Software* **141**, 151–170 (2018)
10. Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C.: Quantitative system performance - computer system analysis using queueing network models. Prentice Hall (1984)
11. Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J., Babar, M.A.: Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. *Information and Software Technology* **131**, 106449 (2021)
12. Ma, M., Lin, W., Pan, D., Wang, P.: ServiceRank: Root Cause Identification of Anomaly in Large-Scale Microservice Architectures. *IEEE Transactions on Dependable and Secure Computing* **19**(5), 3087–3100 (2022)
13. Martin, R.C.: Design Principles and Design Patterns. *Object Mentor* **1**(34), 597 (2000)
14. Ngo, K.L., Mukherjee, J., Jiang, Z.M., Litoiu, M.: Evaluating the Scalability and Elasticity of Function as a Service Platform. In: Proc. of the International Conference on Performance Engineering (ICPE). pp. 117–124 (2022)
15. Pinciroli, R., Aleti, A., Trubiani, C.: Performance Modeling and Analysis of Design Patterns for Microservice Systems. In: Proc. of the International Conference on Software Architecture (ICSA). pp. 35–46 (2023)
16. Quéval, P.J., Zdun, U.: Extracting the Architecture of Microservices: An Approach for Explainability and Traceability. In: Proc. of the European Conference on Software Architecture (ECSA). pp. 346–353 (2023)
17. Di Francesco, P., Lago, P., Malavolta, I.: Architecting with microservices: A systematic mapping study. *Journal of Systems and Software* **150**, 77–97 (2019)
18. Microsoft Learn: Cloud Design Patterns. <https://learn.microsoft.com/en-us/azure/architecture/patterns/> (2022), accessed on April 2024



19. Sousa, T.B., Ferreira, H.S., Correia, F.F.: A Survey on the Adoption of Patterns for Engineering Software for the Cloud. *IEEE Transactions on Software Engineering* **48**(6), 2128–2140 (2022)
20. Thönes, J.: Microservices. *IEEE software* **32**(1), 116–116 (2015)
21. Tundo, A., Mobilio, M., Riganelli, O., Mariani, L.: Monitoring Probe Deployment Patterns for Cloud-Native Applications: Definition and Empirical Assessment. *IEEE Transactions on Services Computing* (2024)
22. Vale, G., Correia, F.F., Guerra, E.M., de Oliveira Rosa, T., Fritzsche, J., Bogner, J.: Designing Microservice Systems Using Patterns: An Empirical Study on Quality Trade-Offs. In: *Proc. of the International Conference on Software Architecture (ICSA)*. pp. 69–79 (2022)
23. Wijerathna, L., Aleti, A., Bi, T., Tang, A.: Mining and relating design contexts and design patterns from Stack Overflow. *Empirical Software Engineering* **27**(1), 8:1–8:53 (2022)
24. Zhang, C., Dardik, I., Meira-Góes, R., Garlan, D., Kang, E.: Fortis: A Tool for Analysis and Repair of Robust Software Systems. In: *Proc. of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. pp. 1–9 (2023)
25. Zhang, C., Garlan, D., Kang, E.: A behavioral notion of robustness for software systems. In: *Proc. of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. pp. 1–12 (2020)
26. Zhang, C., Saluja, T., Meira-Góes, R., Bolton, M., Garlan, D., Kang, E.: Robustification of behavioral designs against environmental deviations. In: *Proc. of the International Conference on Software Engineering (ICSE)*. pp. 423–434 (2023)
27. Zhang, C., Dong, Z., Peng, X., Zhang, B., Chen, M.: Trace-based Multi-Dimensional Root Cause Localization of Performance Issues in Microservice Systems. In: *Proc. of the International Conference on Software Engineering (ICSE)*. pp. 894–894 (2024)