The Journal of Systems and Software 241 (2026) 113006

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

ADEPT: A framework for automated data-driven understanding of design decisions in pattern-based microservices architectures☆,☆☆

J. Andres Diaz-Pace [a,*], Catia Trubiani [b,*], David Garlan [c]

[a] ISISTAN Research Institute, CONICET and UNICEN University, Tandil, Buenos Aires, Argentina
[b] Gran Sasso Science Institute, Via Michele Iacobucci 2, 67100, L'Aquila, Italy
[c] Software and Societal Systems Department, Carnegie Mellon University, Pittsburgh, PA, USA

ARTICLE INFO

Dataset link: https://github.com/andresdp/patterns-sa/

Keywords:
Data-driven sensitivity analysis
Design decisions
Architectural patterns
Microservices
Performance metrics

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 improve the system response time. However, for a given performance requirement, such as service latency or resource utilization, the benefits of choosing an architectural pattern and its design decisions are not guaranteed. Therefore, it is crucial to collect data on patterns' parameters, thereby providing insights to software architects on the relationships between design decisions and performance requirements. In this work, we propose a data-driven approach to assess the quantitative impact of a given pattern and its design decisions, thus understanding the achievement of performance tradeoffs. Our approach seeks to help architects understand and control the pattern parameters that cause variations, i.e. the sensitivity of performance tradeoffs, through scenario discovery techniques. To this end, we rely on machine learning algorithms (i.e., PRIM and CART) to infer constraints on the parameter values. The approach is embodied in a framework called ADEPT, which provides a workflow for generating a pattern design space along with the tradeoffs of interest, identifying key parameters, and then applying scenario discovery algorithms to these tradeoffs. We perform an extensive evaluation using a dataset collected from models of five microservice patterns and their performance characteristics. The experimental results show that ADEPT is effective in identifying key pattern parameters and inferring parameter constraints through PRIM and CART, thus understanding tradeoff variability. Thus, this research supports software architects in making informed design decisions by providing quantitative insights into the performance behavior of microservice patterns.

1. Introduction

Microservice-based applications have gained the attention of researchers and practitioners (Zhang et al., 2024; Di et al., 2024), and the software architecture community has also been attracted to pursue methodologies to support their successful development (Ahmad et al., 2024; Quéval and Zdun, 2023). Although the adoption of patterns has a long history (Martin, 2000), their effective use and impact on quality attributes are a more recent research trend (Sousa et al., 2022; Tundo et al., 2024). 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 (Vale et al., 2022). For instance, focusing on performance-related characteristics such as system response time or resource utilization (Li et al., 2021), it is important to understand how they lead to different tradeoffs depending on specific pattern decisions and related design assumptions.

To enable a quantitative analysis, a recent work (Pinciroli et al., 2023) has shown that well-known microservices patterns can be modeled and evaluated through software performance engineering (SPE) techniques, e.g., queueing networks (Lazowska et al., 1984). Interestingly, the SPE models provide evidence that the performance behavior of the patterns is sensitive to both design decisions and input parameters that are difficult to assess. Furthermore, the variability in the values of such parameters can significantly affect the potential performance tradeoffs. The parameters of a pattern capture its characteristics, including environmental conditions (e.g., incoming heterogeneous requests and varying frequencies), whereas design decisions

☆ This article is part of a Special issue entitled: 'Impactful-Software-Architecture' published in The Journal of Systems & Software.
☆☆ Editor: Jacopo Soldani.

* Corresponding authors.
E-mail addresses: andres.diazpace@isistan.unicen.edu.ar (J.A. Diaz-Pace), catia.trubiani@gssi.it (C. Trubiani), garlan@cs.cmu.edu (D. Garlan).

https://doi.org/10.1016/j.jss.2026.113006
Received 20 February 2026; Received in revised form 29 May 2026; Accepted 10 June 2026
Available online 16 June 2026
0164-1212/© 2026 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

determine the actual implementation to solve a pattern (e.g., a software or hardware isolation mechanism to segregate a few operations).

Given that both the parameters and the design decisions contribute to large variations in the performance behavior of a pattern, it becomes of key relevance to leverage the notion of *sensitivity* of a design decision with respect to achieving desired performance properties, thereby understanding how to define parameter ranges in that regard (Zhang et al., 2023a). For instance, in a microservices pattern, the heterogeneity of requests and their frequencies may influence the design decision to achieve a good tradeoff targeting low utilization and fast response time (Pinciroli et al., 2023). Motivated by the previous context, we investigate the variability of the parameters that characterize microservice-based architectural patterns, so that controlling these parameters can make the pattern (or design model) less prone to performance variations. The goal of this work is two-fold: (i) to understand the performance variation of patterns' design decisions, and (ii) to reduce the performance sensitivity of the decisions through inferring parameter constraints. Our approach provides knowledge to software architects on the performance sensitivity due to the identification of key parameters of a pattern model, thus supporting them in making informed decisions.

In prior work (Diaz-Pace et al., 2025), we proposed a data-driven approach to profile the space of design alternatives for a given model (i.e., 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 relied on machine learning (ML) techniques for scenario discovery (SD), such as PRIM (Friedman and Fisher, 1999) and CART (Breiman et al., 1984), to infer constraints on the parameter values, thereby ensuring that the desired tradeoff is achieved. We performed an initial evaluation with three microservice patterns (Vale et al., 2022): (i) *Anti-corruption Layer*, (ii) *Command Query Responsibility Segregation*, and (iii) *Gateway Offloading*, in which we studied the effects of input parameter variations on two performance metrics of interest, i.e., response time and utilization. The analysis of these metrics, before and after applying parameter constraints, showed that SD techniques were effective in reducing performance variations of the pattern design alternatives.

In this paper, we generalize the previous approach (Diaz-Pace et al., 2025) into a framework, namely ADEPT (Architectural Decision Exploration and Pattern Tradeoffs), which provides a workflow of phases and activities that an architect can apply to explore a pattern design space and gain insights into its performance sensitivity. Moreover, two additional microservices patterns (Vale et al., 2022) have been evaluated: (iv) *Gateway Aggregation*, and (v) *Pipes and Filters*. The evaluation has been redesigned for all five considered patterns, as the newly developed framework provides additional features. Specifically, these features include: the introduction of multiple strategies to define tradeoffs; the specification of novel metrics for assessing the parameter constraints inferred by PRIM/CART; a variety of visualization techniques to inspect the pattern design space and parameter constraints; a plug-in mechanism for incorporating new patterns. To demonstrate the usefulness of ADEPT, we performed experiments to quantitatively compare the PRIM and CART algorithms and investigate the role of the tradeoff definition strategy on the inferred parameter constraints.

The remainder of the paper is organized as follows. Section 2 provides background knowledge on the adopted architectural patterns, the SD framework along with a motivating example that clarifies the main use case for ADEPT, and the PRIM and CART algorithms as alternative SD implementations. Section 3 describes the ADEPT approach and the steps of its workflow. Section 4 briefly introduces the experimental setup, i.e., research questions and experiments' design choices. Section 5 presents the evaluation and answers the raised questions. Section 6 discusses the threats to the validity of the approach. Related work is presented in Section 7, thus positioning our contribution with respect to the state-of-the-art approaches. Section 8 outlines the concluding remarks and future work. Replication data are publicly available; see Section "Data Availability".

2. Background

This section provides background knowledge on the selected microservices patterns, and the role of two Machine Learning (ML) algorithms used for scenario discovery within the framework.

2.1. Architectural patterns

The selected microservices patterns have been modeled and evaluated in Pinciroli et al. (2023). Besides, the theoretical performance models presented in Pinciroli et al. (2023) and validated in Meijer et al. (2024), in which real measurements of resource utilization and system response time align with model-based performance predictions. Given the validity of the performance models and results, we build the dataset for this work by extracting the values of input parameters and output metrics from Pinciroli et al. (2023).

Table 1 lists the patterns and their input parameters that we consider and analyze in this work. Note that the top rows of Table 1 show 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 five considered patterns. The architectural designs of the patterns are taken from the Microsoft's reference catalog.[1]

Gateway Offloading (GO). The problem addressed by this pattern occurs when multiple services require the same functionality (e.g., encryption), thus causing a backlog of requests. The solution prescribed by this pattern involves offloading shared or specialized service functionalities to a gateway proxy that manages them more efficiently, thus preventing service slowdowns. The architecture design consists of four main components: three services (i.e., `S1`, `S2`, and `S3`) and the gateway (i.e., `GW`) that hosts common services. Table 1 indicates that the performance (e.g., response time, utilization, throughput) of this pattern is affected by the service times of components, i.e., S_{GW}, S_{S1}, S_{S2}, and S_{S3}.

Command and Query Responsibility Segregation (CQRS). The problem addressed by this pattern originates from architectures that query and update the same (software or hardware) resource. In the solution offered by this pattern, read operations are segregated from update operations. 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 design differs in the 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`), which requires 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 addressed by this pattern arises when an application leverages different systems for its operation, e.g., a (recently) migrated application needs to interact with a legacy system that makes use of technologies not compatible with the primary application. From a domain-driven design perspective, this pattern applies whenever it is necessary to isolate domain models, not only to set apart legacy systems. The pattern solution introduces an adaptation layer that mediates the communication between incompatible systems. The architectural design consists of three main components: two subsystems (i.e., `SS1` and `SS2`) and the anti-corruption layer (i.e., `ACL`) that mediates the communications between the two subsystems. As shown in Table 1, the pattern's performance is affected by the probability of invoking the ACL component ($Prob$), and by the service times of the three components involved, i.e., S_{SS1}, S_{ACL}, and S_{SS2}.

[1] https://learn.microsoft.com/en-us/azure/architecture/patterns/

Table 1
Architectural patterns and their performance-related parameters. Parameters N and Z apply to all patterns.

	Parameters	Description
	N	Total number of requests in the system
	Z	Thinking time
Gateway Offloading(GO)	S_{GW}	Service time of GW 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
Command and Query Responsibility Segregation (CQRS)	S_{DB}	Service time of DB component
	$S_{\text{DB-read}}$	Service time of DB component to manage *read* requests
	$S_{\text{DB-write}}$	Service time of DB component to manage *write* requests
	$R \leftrightarrow W$	Time needed to synchronize read and write requests
Anti-corruption Layer (ACL)	$Prob$	Probability of invoking the ACL component
	S_{SS1}	Service time of subsystem S_1
	S_{ACL}	Service time of ACL component
	S_{SS2}	Service time of subsystem S_2
Gateway Aggregation (GA)	$Prob$	Probability of invoking the GW component
	S_{GW}	Service time of GW 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
Pipes and Filters (PAF)	S_{T1}	Service time of $T1$ task
	S_{T2}	Service time of $T2$ task
	S_{T3}	Service time of $T3$ task
	S_{T4}	Service time of $T4$ task

Gateway Aggregation (GA). The problem addressed by this pattern manifests itself when an application must interact with multiple services before completing the request of the end user. The solution consists of introducing a gateway component that manages interactions between the application and required services, returning aggregated results while optimizing communication with microservices. Similarly to Gateway Offloading, the architectural design includes the gateway receiving a request and the microservices involved in processing, i.e., S1, S2, and S3. Table 1 shows that the relevant input parameters affecting the performance of this pattern relate to the service times of all involved components, i.e., S_{GW}, S_{S1}, S_{S2}, and S_{S3}.

Pipes and Filters (PAF). The problem addressed by this pattern occurs when applications need to execute tasks associated with requests from different sources, e.g., freemium and premium users. The pattern solution suggests code reuse, thereby facilitating the management of requests and maximizing the efficiency of the application. The architectural design presents separate services, i.e., all tasks are replicated, and requests are forwarded based on their source. Pinciroli et al. (2023) propose a different implementation that replicates *Task 1* and *Task 2*, used to process both types of requests, while *Task 3* and *Task 4* are only used for Freemium and Premium requests, respectively. Table 1 indicates that the input parameters affecting the performance characteristics of this pattern relate to the service times of all involved tasks, i.e., S_{T1}, S_{T2}, S_{T2}, and S_{T4}.

2.2. Exploration of design spaces

A conceptualization of a *design space* for a pattern typically involves two related spaces: (i) the space of architectural configurations, and (ii) the quality-attribute value space (Diaz-Pace and Garlan, 2024). The former refers to all feasible configurations for a pattern that can be derived from a collection of design decisions and parameters. The latter, in turn, contains the values of the performance metrics (e.g., latency, utilization, throughput) that result from evaluating (or simulating) the configurations of the first space.

Exploring these two spaces to assess pattern decisions and their effects on particular performance tradeoffs is challenging for human architects, due to both the size and number of dimensions of each space, and also because the correspondences between instances of the two spaces are not always straightforward. This complexity is further complicated by the uncertainty in some parameters of a pattern model (e.g., see Table 1), which can cause a pattern configuration to drift from a desired performance tradeoff (e.g., low response time and moderate utilization) to another.

In this context, *exploratory modeling* (Kwakkel, 2017) is a methodology that aims to investigate the implications of the inputs to a given model for decision making by conducting a series of experiments. The results of these experiments are analyzed to identify regions of the space that are of interest to the decision maker, which are usually referred to as scenarios. This technique is called *scenario discovery* (SD) (Bryant and Lempert, 2010) and often relies on ML algorithms to infer scenarios from the available data. Fig. 1 shows the ADEPT framework, in which exploratory modeling and scenario discovery are adapted to the domain of microservices patterns.

This work uses two SD algorithms (Bryant and Lempert, 2010), specifically PRIM (Friedman and Fisher, 1999) and CART (Breiman et al., 1984). The architect can apply either one algorithm or the other. For both algorithms, the inputs correspond to pattern parameters related to performance metrics, while the outputs are rules to constrain these parameters. Further scenario discovery algorithms can be adopted, and we plan to consider additional procedures as part of our future research.

2.3. Motivating example

As an example of how ADEPT works, let us consider the *Gateway Offloading (GO)* pattern (Pinciroli et al., 2023). A *GO* instantiation for three services and two pipelines is illustrated in Fig. 2. In this example, let us assume a model with two heterogeneous requests: *request A* requires being executed by the first pipeline (*service1*); *request B* needs to be executed by the second pipeline (*service2* and *service3*). We also assume that the three microservices require requests to be decrypted before executing them. When using the *GO* pattern, the encryption operation is deployed in the *gateway* component.

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 Pinciroli et al. (2023), let us consider three alternatives for this decision, namely: *no offloading*, *offloading short operations* (e.g., those with execution times up to 5 ms), or *offloading long operations* (e.g., execution times between 5 ms and 10 ms). As performance metrics, let us assume

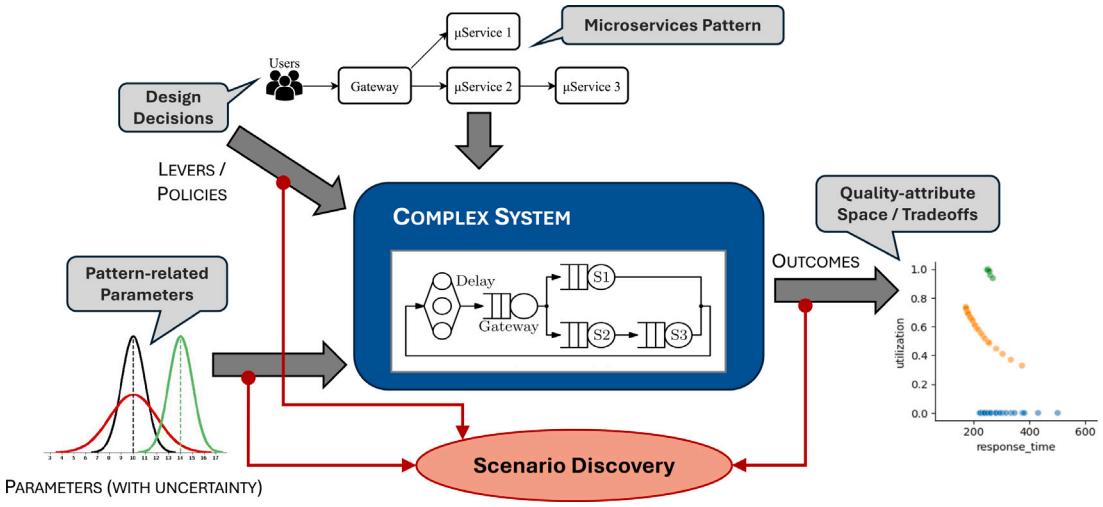
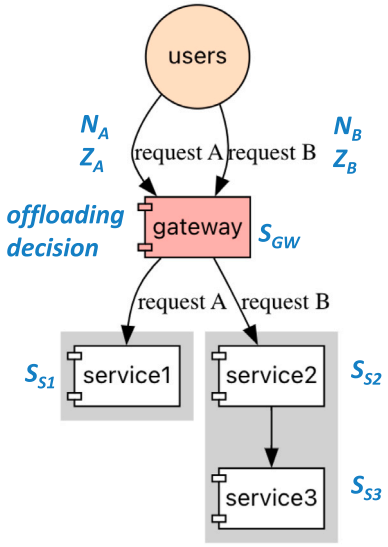


Fig. 1. Adaptive exploratory modeling and scenario discovery to the analysis of design spaces for microservices patterns.



Gateway Offloading		
Req. Type	request A	request B
N	$[0, 25]$	$25 - N_{\mathrm{A}}$
Z	100	100
S_{GW}	$[0, 10]$	$S_{\mathrm{GW,requestA}}$
S_{S1}	$20 - S_{\mathrm{GW,requestA}}$	–
S_{S2}	–	$12 - S_{\mathrm{GW,requestB}}$
S_{S3}	–	$15 - S_{\mathrm{GW,requestB}}$

Fig. 2. Example of a *Gateway Offloading* pattern with performance parameters (these parameters are refined from Table 1).

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) (Lazowska et al., 1984) can be applied to estimate performance metrics. Using the numerical values provided in Pinciroli et al. (2023), certain tradeoffs between utilization and response time emerge, as shown in Fig. 3. For this example, we have discretized the metric values into four bins (or quadrants), each exposing a tradeoff category.

Let us consider that the architect is interested in *GO* configurations with average utilization (*M*) and fast response time (*XS*). In the quality space in Fig. 3 (left), the performance variations across configurations are due to design decisions and parameter assumptions. In this context, a design investigation for the architect is whether constraints can be imposed on the parameters (and also on the decisions) to reduce performance variability and thus increase the chances that a pattern configuration fulfills a given tradeoff (e.g., $\langle XS, M\rangle$). This effort focuses on the *insensitivity* of a set of configurations that share the same decision. In other words, our approach aims to understand how sensitive the performance of a pattern configuration is to variations in its parameters or design decisions. For instance, in the *GO* space in Fig. 3, the decision to offload short services seems to be mostly insensitive to the $\langle XS, M\rangle$ tradeoff, although it might deviate towards configurations with slower response time (*XS* or *M*) and lower utilization (*S*). The no-offloading decision shows a considerable sensitivity in response time for the $\langle XS, XS\rangle$ tradeoff, while off-loading long services is mostly insensitive around the $\langle XS, L\rangle$ and $\langle S, L\rangle$ tradeoffs.

To ensure the $\langle XS, M\rangle$ tradeoff (i.e., fast response time and average utilization), the architect 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 key parameters, such as N_A, N_B, and S_{GW}, in such a way that the *GO* configurations fall within the $\langle XS, M\rangle$ region. This effect can be achieved with $N_B = [2.5, 23.0]$ and $S_{GW} = [2.5, 7.5]$, as indicated by the box in Fig. 3 (right side). These constraints express rules for controlling the parameter values while ensuring a good number of configurations satisfying the target tradeoff.

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 to

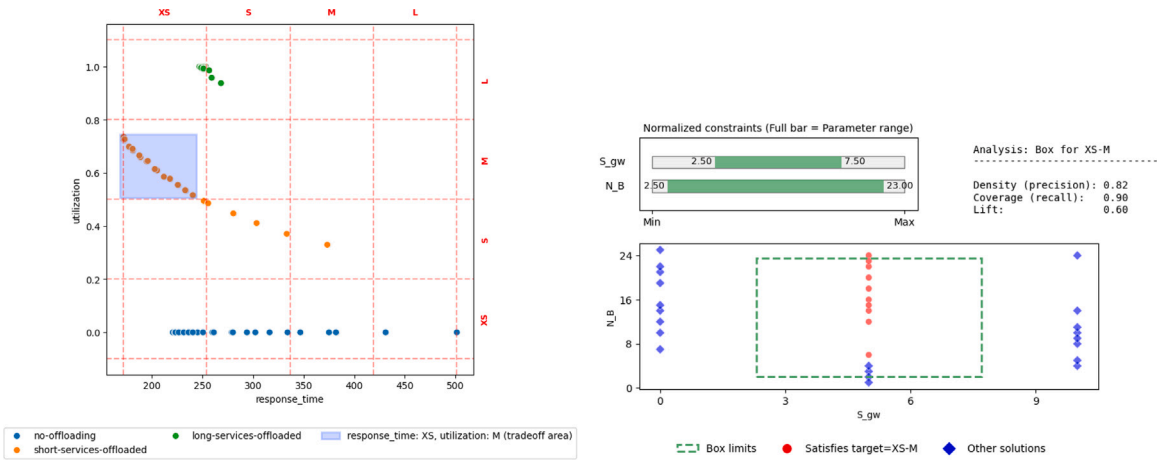


Fig. 3. On the left, tradeoffs between response time and utilization for different configurations of the Gateway Offloading (GO) pattern. An example of constraints imposed (via PRIM) for reducing the sensitivity of achieving a target tradeoff $\langle XS, M \rangle$ are displayed on the right.

the pattern performance, but also environmental factors play a role, e.g., the rate of *request B* (N_B) incoming requests influences the pattern response. In general, finding a set of constraints that yield insensitive configurations with respect to a tradeoff is not straightforward. Here is where SD algorithms become useful. In our example, the constraints for S_{GW} and N_B were derived with PRIM. We can notice in Fig. 3 (right side) that restricting the parameter values creates a 2D "box" or *scenario* that encloses most desired configurations. However, this solution is not perfect, since a few desired configurations fall outside of the box, and some unwanted configurations (i.e., those that do not satisfy the target tradeoff) get into the box. This situation is reflected by the density and coverage metrics of the box. The lift metric (60%) indicates the improvement in density (of configurations) with respect to the original unconstrained space for the target tradeoff.

2.4. ML algorithms for scenario discovery

In the following, we briefly describe the PRIM and CART algorithms. These strategies were chosen because they produce interpretable, rule-based constraints that intuitively support software architects in understanding when to control pattern parameters. Furthermore, both methods are non-parametric and make no strong assumptions about the distribution of pattern configurations in the design space. Nonetheless, ADEPT is not tied to these two alternatives, and other algorithms such as Bayesian rule lists (Wang et al., 2017) and explainable boosting machines (with rule extraction) (Lou et al., 2013) can be applied.

PRIM (Patient Rule Induction Method). This method is a bump-hunting algorithm (Friedman and Fisher, 1999) that searches for regions (bumps) in the input space with relatively high (or low) values of a target variable. In our dataset, this target variable takes a Boolean value indicating whether a performance requirement, typically a tradeoff between performance metrics, is met. PRIM describes the regions by simple, interpretable rules, as they are rectangles called *boxes* in the input space. In our case, the input space refers to the space of pattern configurations and their configuration parameters. PRIM works by iteratively reducing the data size by small amounts. First, candidate boxes are generated. Each box removes a portion of the data based on the levels of a single input variable (parameter). 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 replaces 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 when the stopping criteria are met (e.g., the current box is too small). PRIM seeks regions that have both a high density of positive instances (i.e., those satisfying the target property, or good precision) and good coverage of the space (i.e., good recall) being analyzed.

CART (Classification And Regression Trees). This method is a decision tree algorithm (Breiman et al., 1984) that recursively splits the data into subsets based on the values of input variables, building a tree-like structure to predict a target variable. Similarly to PRIM, we assume that the target variable is a Boolean property referring to a performance requirement, and then use CART to address a classification problem. Unlike PRIM, which takes one property (tradeoff) at a time, CART works as a multi-class classifier and can target several tradeoffs simultaneously. 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 the initial set and is split into two subsets based on the best input and threshold values. These subsets are subsequently 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 (Breiman et al., 1984), which measures the probability of misclassifying a random instance from a subset labeled according to the majority class. The lower the Gini impurity, the purer 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 the accuracy of the classifier. In general, PRIM and CART do not identify the same boxes for a given dataset.

3. Approach

ADEPT (Architectural Decision Exploration and Pattern Tradeoffs) is a data-driven framework to support an architect's understanding of the effects of design decisions and assumptions for a microservices pattern on its performance properties and tradeoffs. Starting from a pattern model (and its corresponding dataset), the main goal is to help the architect identify the decisions and parameters that contribute the most to performance variability and to infer parameter constraints to reduce performance sensitivity with respect to specific tradeoffs. In this way, the architect obtains an envelope of (desired) pattern behavior for her tradeoff of interest. The framework generalizes the techniques presented in our previous work (Diaz-Pace et al., 2025), enabling quantitative evaluation and exploration of the design space associated with a pattern through computational experimentation, sensitivity analysis and scenario discovery (SD). In addition, a variety of visualization techniques are provided for the architect to inspect the pattern design space and the inferred constraints.

Table 2 summarizes the key concepts developed in ADEPT. A *pattern* is modeled in terms of predefined (discrete) design decisions and

Table 2
Main concepts of ADEPT.

Concept	Description
Pattern	The microservice pattern to be analyzed.
Parameter	A variable of the pattern design whose values are affected by uncertainty, often beyond the architect's control (e.g., environment factors). A parameter takes numeric values in a predefined range.
Decision	A variation point within the pattern design, often mapped to an discrete implementation choice (policy).
Configuration space	The collection of all feasible instances of the pattern design, including the combinations of decisions and parameter values.
Quality metric	A quality objective to quantitatively measure (or evaluate) a pattern configuration (e.g., latency, utilization, and throughput for performance).
Quality space	The collection of all performance metric values for the pattern instances (points in the configuration space).
Tradeoff	A well-defined region of the quality space, which encompasses a subset of evaluated configurations.
Box (constraints)	A collection of rules, each one defining the range of allowed values for a pattern parameter.

numeric parameters. A *parameter* captures characteristics of patterns that may vary, e.g., the workload is specified by the number of requests a system is expected to receive. Within a single pattern, there might be different decisions that express how the solutions are implemented. A *decision* represents a policy (in SD terms) that a software architect can test; e.g., when offloading services, there are several opportunities to decide which services to offload. A particular combination of decisions and parameters defines a pattern configuration (or pattern instance). The set of all configurations constitutes the *configuration space*. A pattern configuration is analyzed against one or more quality objectives, each evaluated using a specific *quality metric*, e.g., latency, utilization, and throughput are representative performance metrics. The set of all evaluations constitutes the *quality space*. Within the quality space, the architect can define tradeoffs as regions with specific ranges of values for the quality metrics. A *tradeoff* represents a set of reference values that an architect cares about, i.e., the numerical values of metrics that are significant for the system. Such tradeoffs are intended to guide the architect in understanding the fulfillment of system requirements. Note that certain tradeoff regions might be infeasible (i.e., include none of the evaluated configurations), depending on the metric ranges. A *box* is introduced to define constraints with allowed ranges of values for patterns' parameters in order to reduce performance variability. These ranges are expressed as simple rules for a software architect.

Fig. 4 shows the processing pipeline that comprises four phases: pattern modeling, data exploration, sensitivity assessment of pattern parameters, and constraint inference (using PRIM and/or CART).

3.1. Phase 1: Pattern modeling

The process begins with the specification (or model) of a pattern M, which will be the main target of the analysis. Our framework does not prescribe a specification formalism. The requirements for the pattern modeling are: an initial architecture (i.e., a run-time view of components and connectors), and the specification of design decisions (e.g., a scheduling policy for a server) and parameters (e.g., the variability in the service time of a specific system resource). Furthermore, the performance metrics of interest (e.g., latency, utilization, throughput) should be specified. Design decisions and parameters are inputs for the analysis of M, while performance metrics are treated as outputs. To facilitate the analysis of new patterns, the pattern model is captured in JSON format.

Once M is specified, a sample of pattern configurations involving different values for the decisions, parameters and performance metrics must be collected. The number of pattern instances to sample (or generate) is determined by the architect. Each instance is evaluated by a performance model, which characterizes the behavior of a pattern configuration under given parameter values (e.g., different workload variations) and computes the performance metrics (e.g., response time, utilization). The performance model can be made concrete by a simulator, as in Pinciroli et al. (2023), or come directly from collected system traces (Compagnucci et al., 2026). The instances along with their performance metrics are organized in a dataset. ADEPT is agnostic as to how this dataset is created, as long as its columns and values are identified and mapped in the JSON specification of the pattern.

3.2. Phase 2: Data exploration

Treating the design space of a pattern M as a dataset enables us to perform an exploratory data analysis. In particular, for the quality space, the architect should define the tradeoffs she is interested in. We take an operational perspective on these tradeoffs as regions of the quality space where a property holds. For instance, the architect can define an acceptable tradeoff (or property) for those configurations that achieve a latency lower than a threshold value, and have an average utilization (e.g., computed as the average of utilization values across all the available resources).

To facilitate the definition of tradeoffs, ADEPT provides a number of predefined strategies, as listed in Table 3. These strategies partition quality objectives (metrics) into predefined categories that lead to rectangular regions (or quadrants) in the quality space. The architect can decide to implement her own strategy with a custom partition of the space. A strategy can be static or dynamic, depending on whether it adapts to the characteristics of the quality space. A static strategy involves discretizing values for each performance metric into ordinal (or Likert) categories. For instance, if we use a 2-point scale for latency and utilization and label the resulting ranges of objective values with the categories $\langle S, L \rangle$ (small, large), then we obtain four possible quadrants in the quality space for the pattern configurations. In general, the tradeoff definition strategy depends on the problem and the architect's context.

Once the tradeoff regions are defined, the architect can assess how certain decisions or parameters cause some configurations to fall into a specific region. Furthermore, she can assess whether configurations can drift from one region to another due to a particular decision, but more importantly, due to variations in parameter values. A pattern configuration from M based on a predefined decision D is denoted as M_D. Given a set of configurations for a chosen M_D, variations in their parameter values lead to variations in the quality space values, i.e., the system configurations move across different quadrants, thus determining which values affect the performance metrics under analysis. This situation is illustrated in Fig. 3 (left) with three decisions in the configuration space.

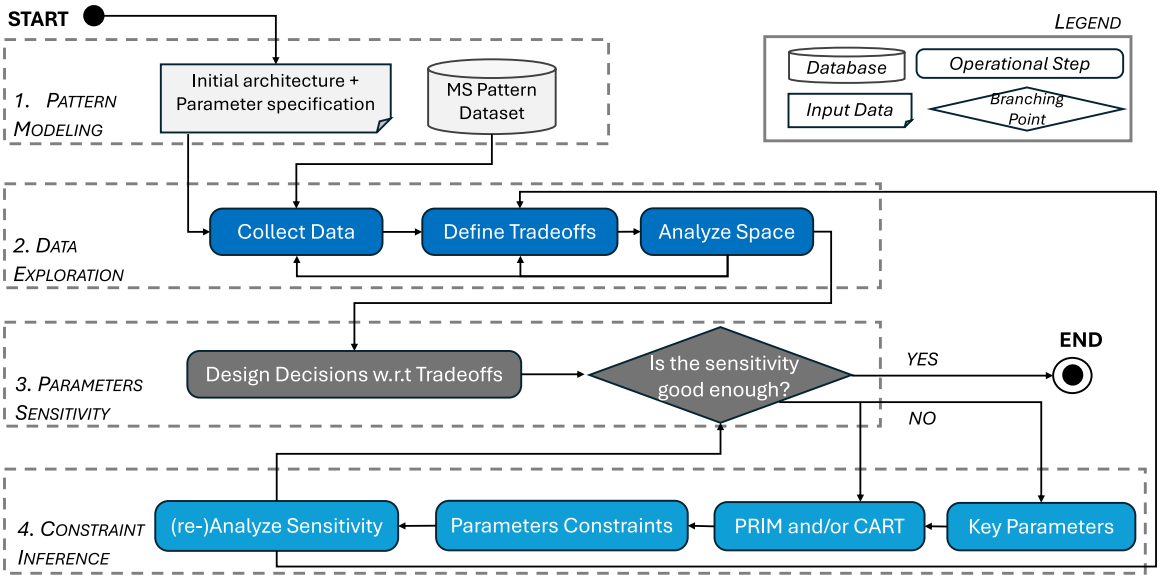


Fig. 4. Main phases of the ADEPT framework for analyzing and acting on the performance sensitivity of microservices patterns. The dotted boxes denote the phases, while the boxes connected by arrows show the suggested execution paths for pattern analysis.

Table 3
Tradeoff definition strategies supported by ADEPT.

Strategy	Description	Categories (per objective)	Static?
Fixed bins	It divides the range of values for each objective into n bins of equal size. The quality space becomes a grid with n^2 quadrants.	$n = 2, \ldots$	Yes
Pareto	It computes the Pareto front (non-dominated solutions) for the quality space. Then, it partitions each objective range into satisfying and non-satisfying solutions. A solution is regarded as satisfying when it lies within an ϵ distance of the Pareto front, and is non-satisfying otherwise. The quality space is then partitioned into 4 quadrants	2	No
Clustering	It applies k-means clustering on each objective range. The best k (number of clusters) is chosen for each objective. Let k_1 and k_2 be the best numbers of clusters for two objectives, then the quality space becomes a grid with $k_1 * k_2$ quadrants.	$k = 2, 3, 4, 5$	No

3.3. Phase 3: Parameters sensitivity

In the third phase, the software architect chooses one or more target tradeoffs and determines whether the performance variability across the M configurations, with unconstrained decisions and parameters, is good enough for her preferences. This assessment can be made, for instance, by visually inspecting the quality space. For example, the decision to offload long services in Fig. 3 shows that the resulting architectural configurations move between the $\langle XS, L\rangle$ and $\langle S, L\rangle$ quadrants, that is, utilization remains high but response time varies from very low to low. The hypothesis is that this variability is due to changes in the numerical values of the pattern parameters. If this performance variability is undesirable (for the decision), then the architect might want to know whether imposing parameter constraints can reduce M variability to one single tradeoff (e.g., the XS, L quadrant). She can also analyze performance variability for other decisions, such as whether to offload short services or not offload at all for the *GO* pattern.

In general, a pattern M_D is said to be insensitive to parameter changes for a given tradeoff T, if all (or most) pattern instances sharing the same decision D and different parameter values appear within the corresponding quadrant for T. In ADEPT, the architect can employ a metric called *density*, which computes the number of configurations within each tradeoff region (quadrant) with respect to the total number of configurations in the quality space. This density can be calculated per pattern decision to get insights into the most frequent tradeoffs that are feasible under that decision. If $density(M_D, T)$ is close to 1.0 and only a small fraction of configurations go to another (but closer) quadrant, the architect can conclude that the performance variability is tolerable for D and T, and no parameter constraints are needed. A low value of $density(M_D, T)$ indicates that constraints should be applied.

3.4. Phase 4: Constraint inference

In this phase, we aim to control parameter uncertainty and thereby reduce the variability of performance metrics within specific tradeoff regions. 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 M_D to satisfy a given property with bound variability. Furthermore, a property corresponds to a tradeoff T with an associated tradeoff region defined by quality-attribute labels.

To reduce the number of pattern parameters that SD algorithms must process, a filter can be applied to process only the key parameters for the performance metrics. This means that the constraints resulting from CART or PRIM will be expressed only in terms of these key parameters. Identifying correlated features (i.e., parameters) in a dataset

and performing feature selection is a common but not straightforward ML task. Features with a greater diversity of values tend to be more predictive, while those with the least missing data tend to be more useful. In ADEPT, we rely on a procedure called *smart correlation*,[2] which looks for groups of highly-correlated (numerical) parameters and uses a heuristic to select the most informative ones. Initially, the correlation matrix for all features (e.g., using methods such as Pearson or Spearman) is calculated, and the features that are highly correlated with each other are clustered. For each cluster, the feature with the highest variance is kept and the other features are dropped. This filtering procedure is optional in the ADEPT pipeline, although recommended when a pattern involves a large number of parameters.

The chosen SD algorithm (either PRIM or CART) is executed with the key parameters and target tradeoffs as input, and it returns the parameter constraints (boxes) that the architect can inspect and apply to the pattern configurations. Note that the SD algorithms are not guaranteed to find boxes for all target tradeoffs, as they depend on both the provided data and the algorithms' heuristic nature.

To ensure that the inferred constraints are generalizable, an ML experiment is performed on the pattern dataset. For this reason, it is good practice to split the original dataset into two parts: a training set and a test set. In our context, the SD algorithm is computed on the training set and evaluated using the test set. We re-compute the density for each box (on the test set) after applying the parameter constraints. Ideally, when the constraints for a given tradeoff are effective, the density should increase. This means that the box limits are tight enough to enclose a good number of satisfying configurations, while keeping non-satisfying ones outside the box. However, an additional metric of interest is the box *coverage*, which refers to the number of enclosed satisfying configurations with respect to the total number of satisfying solutions. If many satisfying solutions lie outside the box, coverage is low, i.e., the box is too narrow for insensitive solutions. In general, the architect should strike a balance between density and coverage.

4. Experimental setup

We evaluate our approach by applying it to five microservice patterns and computing performance metrics for the pattern configurations, using the PRIM and CART algorithms when appropriate. The experiments involve these patterns: *Gateway Offloading (GO)*, *Command Query Responsibility Seggregation (CRQS)*, *Anticorruption Layer*(ACL), *Gateway Aggregation* (GA), and *Pipes and Filters* (PAF), as modeled in Pinciroli et al. (2023).

We aim to address the following research questions.

- **RQ#1**: For a given pattern, which parameters and design decisions are more effective for satisfying a performance tradeoff?
- **RQ#2**: For a given design decision and parameters, how do PRIM and CART help to reduce sensitivity with respect to a performance tradeoff?
- **RQ#3**: How does the initial definition of tradeoffs affect the results of PRIM and CART?

For **RQ#1** and **RQ#2** we use a 4-bin discretization of the quality space considering response time and utilization (or throughput) as performance metrics, as shown in Fig. 5. The distribution of configurations for the *GO* is given in Fig. 3 (left). There are three alternative decisions for the *GO* and *PAF* patterns, two decisions for the *CQRS* pattern, and one decision for the *ACL* and *GA* patterns. Each quadrant refers to a distinct performance tradeoff. Not all tradeoffs were feasible in our sampled design spaces for the patterns. For **RQ#3**, we extend the tradeoff definition with a 2-bin discretization, a Pareto-based and a clustering-based strategies. For consistency, all labels are named using the scale $[XS, S, M, L, XL]$ to denote increasing objective values.

[2] https://feature-engine.trainindata.com/en/1.8.x/user_guide/selection/SmartCorrelatedSelection.html

According to the processing pipeline of Fig. 4, 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.

To identify key parameters and evaluate PRIM and CART, we split the pattern datasets into training and test sets (70/30) using stratification on tradeoff labels, as typically done in ML. We remove outliers from the datasets using the z-score method ($z = 3$). Pearson correlation was used for feature analysis to retain the key pattern parameters, which 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 constraints (boxes). In CART, we consider each design decision separately, as the resulting tree generates the parameter constraints for all the tradeoff labels at once.

For answering **RQ#2** and **RQ#3**, all (non-empty) boxes obtained from both algorithms are used for evaluation on the test set. To quantify the effects of the parameter constraints, we compute the density of configurations for a target tradeoff before and after applying the boxes (lift). Note that, using common ML terminology, density can be seen as precision and coverage as recall for a tradeoff region. We also analyze differences in average values of lift, precision, and recall for the boxes generated by each algorithm to assess sensitivity improvements under different tradeoff definition strategies.

5. Evaluation

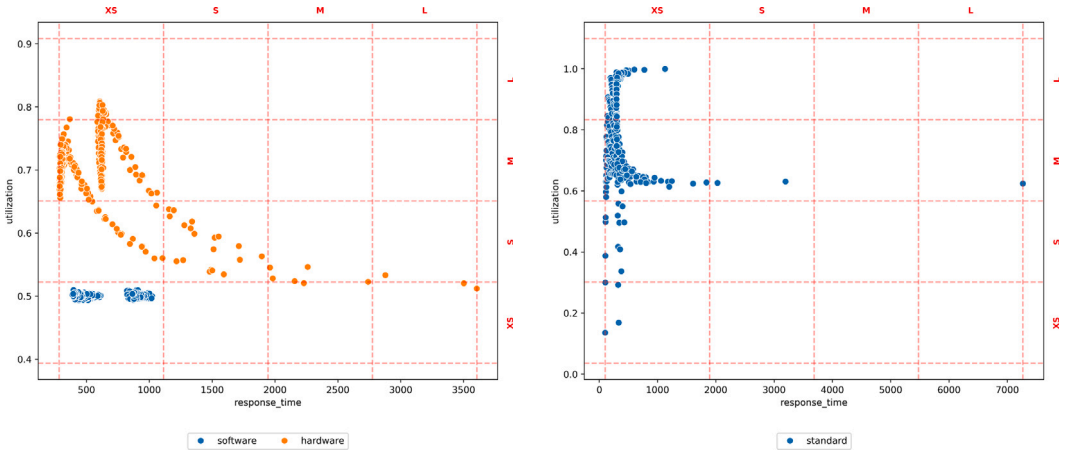
5.1. RQ#1: Role of pattern decision and parameters in performance tradeoffs

In this research question, we assess the effects of the pattern decisions on the tradeoffs between the performance metrics, and then discuss whether parameter constraints need to be imposed.

Tradeoff analysis of GO pattern. Fig. 3(a) shows that the decision of no-offloading always results in a low system utilization (XS), and the response time can fluctuate in the range fast–slow ($[XS, L]$) depending on the parameter values. This scenario indicates that the architect should apply parameter constraints to ensure a desired response time. In turn, the decision of offloading long services always leads to high utilization (L) and fast response time ($[XS, S]$), and thus seems to be an adequate decision for a reasonable performance (if high utilization is not problematic for the system). Parameter constraints do not seem to be needed in this case (unless the architect would prefer XS or S for the response time range). 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 response time. Also, parameter constraints might help the architect choose between S or M for utilization.

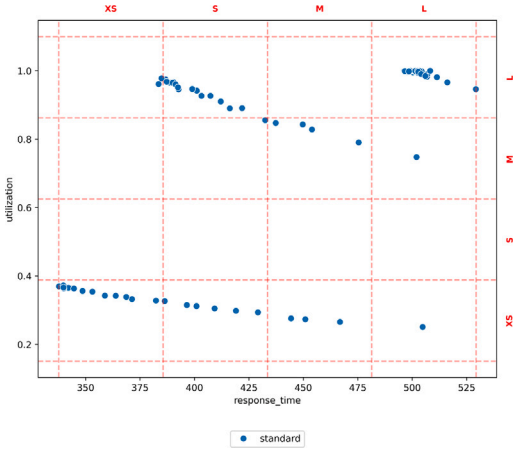
Tradeoff analysis of CQRS pattern. Fig. 5(a) exhibits scenarios of trading off average-to-low utilization and average-to-fast response time. If the software decision is chosen, response time will always be fast but system utilization will be low, without the need for parameter constraints. The hardware decision is preferred for obtaining a better (i.e., higher) utilization, but the response time might degrade for some parameter values in the configurations. In this case, the architect should control the parameter values to keep the response time in the XS and S levels.

Tradeoff analysis of ACL pattern. Fig. 5(b) differs from the previous ones, since all performance variations are due to parameter values. The best configurations have average-to-high utilization ($[M, L]$) with mostly fast response time (XS). However, deflections towards low utilization or slower response times are observed. To avoid those cases, the architect should apply appropriate parameter constraints to favor the $\langle XS, M \rangle$ quadrant, especially on the probability of dispatching requests to the ACL component.

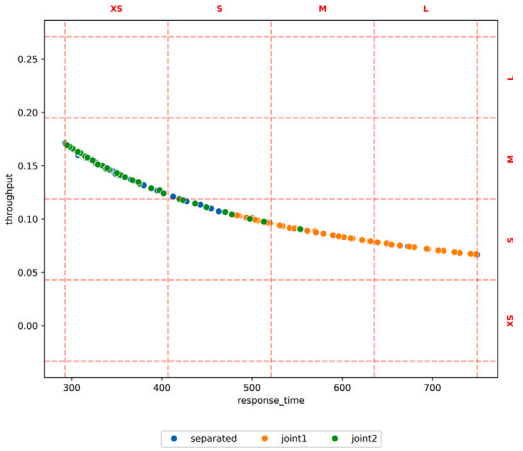


(a) Command Query Responsibility Segregation (CQRS)

(b) Anti Corruption Layer (ACL)



(c) Gateway Aggregation (GA)



(d) Pipes and Filters (PAF)

Fig. 5. Distribution of pattern configurations with respect to the performance tradeoffs (discretization into 4 bins).

Tradeoff analysis of GA pattern. Fig. 5(c) displays three well-defined shapes of tradeoffs due to parameter variations. There is an undesirable scenario (top-right quadrant), in which response time is slow (L) and utilization is high (L). In the other two scenarios, we notice that slower response times are correlated with lower utilization. Thus, the architect might decide between a fast-to-average response time with low utilization ($\langle XS, SX\rangle$ or $\langle S, XS\rangle$) or average response time and utilization ($\langle M, M\rangle$). In both cases, parameter constraints need to be imposed to avoid deflections to unwanted tradeoff quadrants.

Tradeoff analysis of PAF pattern. Fig. 5(d) shows a clear correlation between response time and throughput: the faster the response time, the higher the throughput. Decisions involve configurations with separated services (i.e., services processing different requests are deployed multiple times) or joint services. Except for very few configurations, the allocation of 2 CPUs to the joint services performs similarly as having separated services, which sets apart from the case of joint services with 1 allocated CPU. The options for separated services and 2-CPU joint services achieve both fast-to-average response time and average throughput, which seem desirable scenarios. Parameter constraints might be imposed depending on whether a fast response time or average throughput is preferred. In the option for 1-CPU joint services, in turn, the response time increases significantly but throughput goes down ($\langle M, S\rangle$).

Summarizing, the analyses of the quality spaces expose the pros and cons of each microservice pattern, and how decisions can be made to achieve specific performance tradeoffs, thus answering **RQ#1**. Furthermore, the analyses expose the sensitivity of certain pattern configurations based on their parameter values, depending on the tradeoff targeted by the software architect.

5.2. RQ#2: Effectiveness of PRIM and CART to address target tradeoffs

Based on the pattern 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 in terms of density scores are summarized in Tables 4–8. Performance tradeoffs are represented by response time and utilization (or throughput for *PAF*) with four bins (XS, X, M and L). Note that test sets represent the baselines, before applying any algorithm. The density values are in bold (and green), underlined (and yellow) or framed (and red) to highlight whether the change (lift) was positive, marginal or negative, respectively, in terms of configurations meeting the target tradeoff after imposing the constraints. All the information about constraints is provided in the reproducibility package.

Metrics and Constraints for GO pattern.

Out of 8 possible parameters, N_B and S_{S2B} turned out to be the most relevant ones for the pattern performance sensitivity. We recall that all patterns are subject to heterogeneous requests, namely A and B. This means that the number of B requests (N_B) is recognized as to

Table 4
Density metrics for the *GO* pattern considering 4 bins. Bins are labeled as follows: extra-small (XS), small (S), medium (M), and large (L). Decisions are: no-offloading (NO), short-services-offloaded (SSO), long-services-offloaded (LSO).

GO	Decision	Performance tradeoff	M-XS	S-S	S-XS	XS-L	XS-M	XS-XS
Test set	NO		**0.11**		**0.33**			**0.56**
	LSO					**1.00**		
	SSO			**0.29**			**0.71**	
PRIM	NO	S-XS	0.11		0.33			0.56
		XS-XS			0.29			**0.71**
	LSO	XS-L				1.00		
	SSO	XS-M		0.18			**0.82**	
CART	NO	S-XS			**1.00**			
		XS-XS			0.17			**0.83**
	LSO	XS-L				1.00		
	SSO	XS-M					**1.00**	

Table 5
Density metrics for the *CQRS* pattern considering 4 bins. Bins are labeled as follows: extra-small (XS), small (S), medium (M), and large (L). Decisions are: hardware separation (HW) and software separation (SW).

CQRS	Decision	Performance tradeoff	S-S	XS-L	XS-M	XS-S	XS-XS
Test set	HW		**0.06**	**0.09**	**0.79**	**0.06**	
	SW						**1.00**
PRIM	HW	S-S	**0.50**				
		XS-L		**0.65**	0.35		
		XS-M			**1.00**		
		XS-S	0.08		0.33	**0.58**	
		XS-XS	0.08	0.18	0.74		
	SW	S-S					1.00
		XS-L					1.00
		XS-M					1.00
		XS-S					1.00
		XS-XS					1.00
CART	HW	XS-M			**1.00**		
	SW	XS-M					1.00

Table 6
Density metrics for the *ACL* considering 4 bins. Bins are labeled as follows: extra-small (XS), small (S), medium (M), and large (L). The decision is about the frequency of invoking the ACL component.

ACL	Decision	Performance tradeoff	S-M	XS-L	XS-M	XS-S	XS-XS
Test set	ACL		**0.01**	**0.26**	**0.71**	**0.01**	**0.01**
PRIM	ACL	XS-L		**0.79**	0.21		
		XS-M	0.03		**0.97**		
CART	ACL	XS-L		**0.88**	0.12		
		XS-M	0.03		**0.97**		

have a large impact on the system response time, whereas the service time when invoking the S_2 service (S_{S2B}) is responsible for variations in utilization values.

In Table 4 , the first block of rows (test set) shows the initial scores for different tradeoffs, and we can notice that these tradeoffs depend on the decision implemented at the gateway. For example, the first row indicates that with no-offloading 56% of solutions belong to the $\langle XS, XS\rangle$ tradeoff, 33% falls into the $\langle S, XS\rangle$ quadrant, while 11% relates to the $\langle M, XS\rangle$ tradeoff. When PRIM is applied, for instance, for the *SSO* decision and $\langle XS, M\rangle$ as the target tradeoff, 82% of the constrained configurations belong to that tradeoff, which gives a lift of 11% with respect to the initial score (71%). As a side-effect of the constraints, the $\langle S, S\rangle$ quadrant is affected and a lower percentage of configurations remain in that tradeoff (from 29% to 18%). Both algorithms are able to effectively constraint certain parameters for ensuring the desirable tradeoffs of Fig. 3, namely: $\langle XS, XS\rangle$ (no-offloading), and $\langle XS, M\rangle$ (short-services offloading). On the one hand, PRIM was not effective in improving a portion of the tradeoffs. On the other hand, the tradeoffs addressed by CART improved significantly. CART outperformed PRIM for $\langle S, XS\rangle$ (1.00 versus 0.33 with respect to an initial score of 0.33) and $\langle XS, M\rangle$ (1.00 versus 0.82 for an initial score of 0.71), under the decision of short-services offloading; and for $\langle XS, XS\rangle$ (0.83 versus 0.71 for an initial score 0.56), under the decision of no offloading. The differences can be attributed to the internal working of the algorithms, which returned different parameter ranges for the same targets. Although the CART boxes were more effective than those of PRIM, they tended to have a lower coverage than in PRIM.

Metrics and Constraints for CQRS pattern. The most prevalent tradeoffs are $\langle XS, M\rangle$ for the hardware decision and $\langle XS, XS\rangle$ for the software alternative, as shown in Fig. 5(a) and corroborated by Table 5 (first block of rows). From a list of 8 parameters, the frequency of write requests in the database (Z_write) is identified as the key parameter for pattern's performance sensitivity, along with the service time of the database to manage read requests ($S_{DB-read}$). Besides, the analysis reveals a lower importance for the parameter related to the service time when writing requests ($S_{DB-write}$).

Table 7
Density metrics for the *GA* pattern considering 4 bins. Bins are labeled as follows: extra-small (XS), small (S), medium (M), and large (L). The decision involves aggregating the results at the gateway component.

GA	Decision	Performance tradeoff	L-L	M-M	M-XS	S-L	S-M	S-XS	XS-XS
Test set	GA		**0.41**	**0.06**	**0.03**	**0.19**	**0.03**	**0.06**	**0.22**
PRIM	GA	L-L	**1.00**						
		M-M		**0.22**		0.67	0.11		
		M-XS			**0.10**			0.20	0.70
		S-L		0.13		**0.75**	0.13		
		S-XS						**0.22**	0.78
		XS-XS							**1.00**
CART	GA	L-L	**1.00**						
		M-M		**1.00**					
		S-L				**1.00**			
		S-XS						**1.00**	
		XS-XS							**1.00**

Table 8
Density metrics for the *PAF* pattern considering 4 bins. Bins are labeled as follows: extra-small (XS), small (S), medium (M), and large (L). Decisions are: all services deployed with 1 CPU (Joint1), all services deployed with 2 CPUs (Joint2), and services separately deployed (Sep).

PAF	Decision	Performance tradeoff	L-S	M-S	S-M	S-S	XS-M
Test set	Joint1		**0.33**	**0.39**		**0.28**	
	Joint2			**0.04**		**0.24**	**0.72**
	Sep		**0.05**	**0.05**	**0.05**	**0.11**	**0.74**
PRIM	Joint1	L-S	**0.83**	0.17			
		M-S	0.33	**0.67**			
		S-S	0.33	0.39		0.28	
		XS-M	0.15	0.46		**0.38**	
	Joint2	XS-M				0.14	**0.86**
	Sep	XS-M			0.07	0.07	**0.86**
CART	Joint1	L-S	**0.75**	0.25			
		M-S	0.50	**0.50**			
	Sep	XS-M				**0.25**	0.75

Table 9
Precision (density), recall (coverage) and lift metrics for the parameter constraints (boxes) by pattern and algorithm.

Pattern	Algorithm	Precision (avg)	Recall (avg)	Lift (avg)
GO	PRIM	0.48	**0.65**	0.31
	CART	**0.64**	0.49	**0.47**
CQRS	PRIM	**0.36**	**0.63**	**0.31**
	CART	0.05	0.01	−0.15
ACL	PRIM	0.35	**0.30**	0.15
	CART	**0.37**	0.29	**0.17**
GA	PRIM	0.47	**0.86**	0.33
	CART	**0.71**	0.48	**0.57**
PAF	PRIM	**0.47**	**0.46**	**0.27**
	CART	0.40	0.13	0.20

As noted from the metrics in Table 5, both PRIM and CART obtain maximal effectiveness at improving the $\langle XS, M\rangle$ target, which is a desirable tradeoff under the hardware decision. CART is very limited for this dataset and could not address other tradeoffs. Instead, PRIM is able to improve $\langle S, S\rangle$ (0.5 with respect to an original score of 0.06), $\langle XS, L\rangle$ (0.65 with respect to 0.09) and $\langle XS, S\rangle$ (0.58 with respect to 0.06), under the hardware decision. Note that, despite the PRIM improvement, the $\langle XS, L\rangle$ quadrant is very unlikely to be targeted, because utilization is too high. The defined tradeoff strategy involves a larger number of boxes, and this choice highlights an interesting feature of PRIM when compared to CART.

Metrics and Constraints for ACL pattern. The most prevalent tradeoffs are $\langle XS, M\rangle$ and $\langle XS, L\rangle$, from which the first quadrant is a desirable target, according to Fig. 5(b). The constraints computed by both algorithms yield gains in the density scores for the $\langle XS, M\rangle$ tradeoff, with a slightly higher lift of CART over PRIM, as evidenced in Table 6. Out of 14 parameters, N_A is determined to be key for performance sensitivity, since it relates to the workload of requests, thus affecting the performance metrics. A further parameter marked as relevant for both response time and utilization is the probability of invoking subsystem S_1. Specifically, the B requests affect the performance characteristics of the pattern.

Metrics and Constraints for GA pattern. Among the key parameters, our experiments show that the number of requests (N) affects the system response time, particularly for the B requests (N_B). The same type of requests is associated with a probability of invoking the gateway component (*Prob*), which has a large impact on the performance metrics under analysis. In total, 12 parameters were analyzed by the feature selection algorithm. Both PRIM and CART address most of the target tradeoffs satisfactorily, particularly the more prevalent quadrants ($\langle M, M\rangle$, $\langle S, XS\rangle$ and $\langle XS, XS\rangle$). Table 7 shows that CART achieves a higher lift than PRIM. Nonetheless, CART is unable to find boxes for the $\langle M, XS\rangle$ quadrant. Furthermore, PRIM outperforms CART in terms of coverage (for every target tradeoff).

Metrics and Constraints for PAF pattern. From a list of 14 parameters, our analysis indicates that the number of requests (N) affects both the system response time and utilization, particularly for A requests (N_A). Moreover, the service time of the T_1 task is recognized as the most relevant parameter for this pattern. This is expected, since the T_1

Table 10
Precision (density), recall (coverage) and lift metrics for the parameter constraints (boxes) for different strategies for tradeoff definition. The chosen patterns (*GO*, *CQRS* and *PAF*) involve design decisions.

Pattern	Strategy	Algorithm	Precision (avg)	Recall (avg)	Lift (avg)
GO	2 x 2 bins	PRIM	0.67	0.64	0.33
		CART	0.65	0.50	0.31
	4 x 4 bins	PRIM	0.48	0.65	0.31
		CART	0.64	0.49	0.47
	Pareto (2 x 2 bins)	PRIM	0.60	0.50	0.35
		CART	0.66	0.48	0.41
	clustering (2 x 4 bins)	PRIM	**0.83**	**0.89**	**0,67**
		CART	**0.96**	**0.81**	**0.80**
CQRS	2 x 2 bins	PRIM	0.50	0.36	0.00
		CART	0.50	0.04	0.00
	4 x 4 bins	PRIM	0.36	0.63	0.16
		CART	0.05	0.01	−0.15
	Pareto (2 x 2 bins)	PRIM	0.64	0.43	0.39
		CART	0.45	0.07	0.20
	clustering (4 x 2 bins)	PRIM	0.31	0.51	0.15
		CART	0.14	0.00	−0.03
PAF	2 bins	PRIM	0.54	0.69	0.21
		CART	0.58	0.22	0.25
	4 bins	PRIM	0.47	0.46	0.27
		CART	0.40	0.13	0.20
	Pareto (2 bins)	PRIM	1.00	0.69	0.00
		CART	1.00	0.55	0.00
	clustering (3 x 3 bins)	PRIM	**0.77**	**0.86**	**0.44**
		CART	**0.89**	0.32	**0.56**

task plays a central role in the computation of all requests; therefore, restricting it helps to achieve results with less variability. According to Table 8, CART is unable to address certain tradeoffs, particularly under the decision of having a service deployment with 2 CPUs (joint2). PRIM achieves a better lift for the most prevalent tradeoffs, $\langle M, S\rangle$ and $\langle XS, M\rangle$, the latter being a desirable target for the architect, under the decisions of having services deployed with 2 CPUs or separately.

Table 9 summarizes the density (precision), coverage (recall) and lift metrics for PRIM and CART averaged over all (feasible) tradeoffs, and the values in bold show the highest values. For the *GO* and *GA* patterns we notice similar results, i.e., highest recall achieved by PRIM, and largest precision and lift obtained by CART. In the *ACL* pattern, both algorithms achieve similar, relatively low metrics. The *CQRS* and *PAF* patterns show, in turn, a different trend, in which PRIM outperforms CART regarding density and coverage, although with moderate scores.

> The analysis results show a similar performance for PRIM and CART in terms of reducing the sensitivity of target tradeoffs, with the former algorithm often inferring more boxes than the latter. CART achieves a better precision while PRIM gets a better coverage (with a reasonable precision), which answers **RQ#2**. In general, the improvements for all the patterns are most noticeable for those tradeoffs encompassing many configurations (i.e., dense quadrants) in the dataset. In practice, not all tradeoffs (whether addressed or not by the algorithms) are desirable targets for an architect.

5.3. RQ#3: Alternative strategies for tradeoff definition

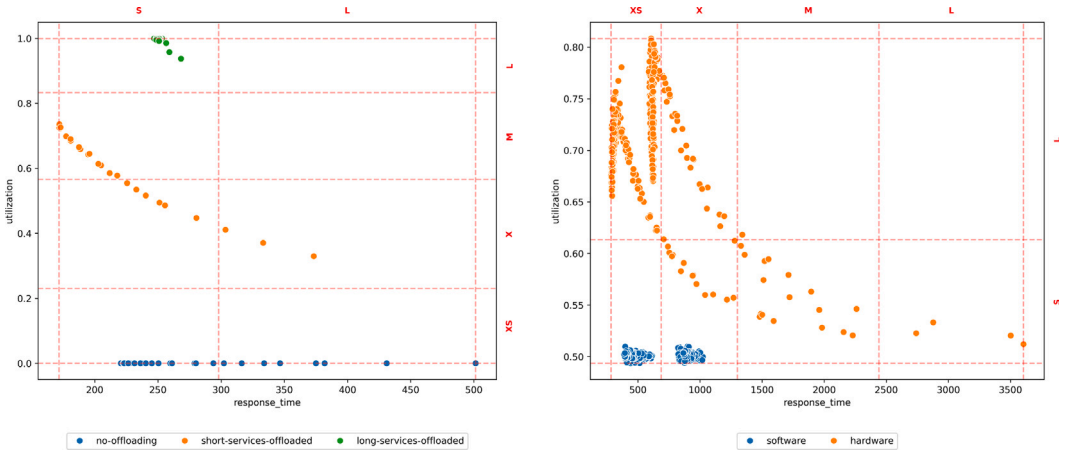
Since the previous experiments were carried out with a discretization of the performance metrics in four fixed bins, in this research question we test the approach with alternative binning strategies, as described in Table 3. For each alternative, we make PRIM and CART infer the corresponding parameter constraints and compute density (precision) and coverage (recall) metrics. The results for the *GO*, *CQRS* and *PAF* patterns, which are the ones involving multiple decisions, are summarized in Table 10.

Like in Table 9, we observe tradeoffs between precision (in favor of CART) and coverage (in favor of PRIM), but they are modulated by the bounds of the bins. In (static) strategies with fixed bins ($2x2$ or $4x4$), they influence how similar (or dissimilar) configurations are grouped, and thus how the parameter constraints are derived. The Pareto and clustering strategies can adapt to the distribution of solutions in the quality space. For instance, Fig. 6 shows the partitioning of the quality space for the three patterns using k-means clustering on each performance objective, and we notice a varying number of bins per objective. When it comes to the improvements attributed to the parameter constraints, we found that clustering performed better than the other strategies for the *GO* and *PAF* patterns (marked in bold in Table 10). The *CQRS* case was an exception, because CART performed poorly, but we still noticed variations in the density and coverage of PRIM when using alternative definition strategies.

> We corroborate that using alternative definition strategies for the tradeoffs changes the way in which the evaluated configurations are partitioned, which can make constraint inference harder or easier for the PRIM and CART algorithms. Thus, we answer **RQ#3** by saying that the density and coverage of the resulting boxes seems to be dependent on the tradeoff definitions. Ultimately, the definition of the acceptable tradeoffs for the system should be provided by the architect.

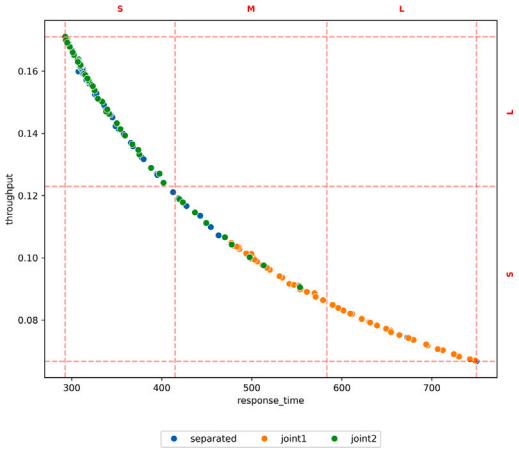
6. Threats to validity

Internal validity. The pattern datasets are generated from performance models (Queuing Networks) that were validated in prior work (Pinciroli et al., 2023; Meijer et al., 2024). These models make certain assumptions that might not capture real-world dynamics or might miss parameters that influence system performance. Furthermore, the



(a) Gateway Offloading (GO)

(b) Command Query Responsibility Segregation (CQRS)



(c) Pipes and Filters (PAF)

Fig. 6. Distribution of pattern configurations with respect to the performance tradeoffs (k-means clustering). Tradeoff regions adapt their shapes to the points of each quality space.

datasets are static, i.e. they provide predetermined system snapshots (configurations) and might not include other relevant configurations. The causal relationship between the parameters and tradeoffs tackled by ADEPT is affected by these factors. Since both PRIM and CART are heuristic algorithms, their outputs are not guaranteed to be optimal and also depend on the way the datasets are split. In the case of CART, the conversion of a classification tree into boxes can affect the accuracy of the parameter constraints. The filtering of key parameters (feature selection) is beneficial for the analysis and resulting constraints, but it might discard parameters having complex effects on the performance metrics (e.g., non-linear relationships). Performing a parameter filtering before SD is currently optional in ADEPT. Besides, more experiments are needed to derive guidelines on when to use parameter filtering. In the experiments, we assume that tradeoff sensitivity is caused by a combination of (discrete) decisions and (numeric) parameters, but our SD analysis does not currently distinguish their individual effects. Thus, there is a risk of attributing performance sensitivity to the wrong element. Addressing this confounding effect through a refined analysis is the subject of future work.

External validity. Our evaluation is performed on five microservices patterns. While these patterns are representative, they do not cover the whole spectrum of pattern-based microservices systems. Furthermore, we assume one pattern per system and did not analyze systems composed of multiple patterns. The SD algorithms proved to be effective in inferring constraints for the considered datasets, although studies with other microservices patterns and systems need to be carried out in order to generalize our findings. The datasets were generated by sampling the parameters of the performance models, which might have biased the resulting parameter space, when compared to the usage of actual system traces as input to the approach. The results of using alternative tradeoff definitions showed variations in the performance of the algorithms, suggesting further investigation of this aspect. Finally, our analyses were based on the predictions of the performance models, and we did not compare the results with the (actual) runtime behavior of the services or the metric values using each pattern.

Construct validity. The concept of tradeoff is key in the software architecture domain, as well as in the ADEPT framework, since we focus on tradeoff sensitivity due to pattern decisions and parameters. Our framework provides an operational view of a tradeoff as a region (quadrant) of evaluated configurations in the quality space, allowing automated definition strategies (e.g., fixed bins, Pareto, clustering). Although we intend to keep tradeoff definitions simple, having a grid of rectangular shapes in the quality space might not reflect the architect's view of what a tradeoff is for her problem (e.g., a tradeoff might not necessarily be a quadrant). The boundaries created by the quadrants can lead to the loss of information for certain configurations. Note that this is not a limitation of SD algorithms, as they can infer constraints for any boolean property, regardless of its shape in the quality space. Our experiments rely on pattern models and performance metrics (e.g., response time, utilization, throughput) as proxies for pattern behavior, which together provide the basis for the tradeoff definition

and sensitivity analyses. Thus, the validity of our results relies on how well these models and quality metrics reflect the architect's concerns.

7. Related work

Microservices represent a new development strategy intended to guarantee agility, scalability, maintainability, and performance (Jamshidi et al., 2018), hence, even major vendors, e.g., Netflix (Thönes, 2015), have been attracted by these premises. In the literature, recent approaches have been developed to study the performance characteristics of microservices, e.g., Zhang et al. (2024), Di et al. (2024), Ahmad et al. (2024). Hereafter, we present the approaches mostly related to the aspects tackled in this manuscript, acknowledging that our selection of related work is not exhaustive.

Performance evaluation. Our work focuses on interpreting the performance characteristics of architectural patterns for microservices. This is motivated by a few survey articles that analyze various quality attributes and identify performance as one of the most prominent. For instance, Di Francesco et al. (2019) study 103 papers on microservice architectures and observe that performance is growing in popularity and relevance with respect to other quality properties. A further endorsement is provided by Li et al. (2021) that review 72 studies and indicate performance as one of the most critical attributes when designing a microservices application. Wijerathna et al. (2022) raise attention to the bad practice of delaying the performance evaluation until the latest stages of the development process. As a result, fixing issues becomes expensive, whereas it is preferable to incorporate performance-based knowledge from the architectural phase. Rodrigues et al. (2025) review 784 papers and select 26 primary studies on the performance and scalability assessment of microservice architectures. Interestingly, architectural patterns are associated to positive and negative impacts, depending on the pattern application, as also demonstrated by our work. In summary, all these studies support our research effort to focus on the interpretation of performance predictions, since they do affect the quality of microservice architectures.

Architectural patterns. Our work targets the architectural patterns that have been defined in the literature as relevant for the performance of microservice-based applications. This is motivated by a few methodological articles that analyze various architectural patterns and recognize that they do affect the quality of the corresponding software architectures. For instance, Khomh and Abtahizadeh (2018) study the influence of six design patterns on a cloud-based application, specifically focusing on performance and energy consumption metrics that exhibit large variations. However, there is no sensitivity analysis in the derived findings, as we do in this paper. Akbulut and Perros (2019) present a performance evaluation of three design patterns, and their measurements showcase that their performance metrics of interest improve or worsen depending on the application scenarios. Amiri et al. (2021) explore performance and reliability trade-offs as architectural patterns are varied. Ma et al. (2022) identify anomalous performance behaviors by proposing a correlation mechanism that detects the occurrence of cloud design patterns. Ngo et al. (2022) analyze the performance of a serverless application, and improvements are attributed to the adoption of a design pattern that balances incoming request load. Liu et al. (2025) present a collection of architectural patterns for foundation model-based agents, i.e., AI components that reason on the context and collaboratively make decisions. This concept can be applied to our approach: we can introduce agents to assess whether parameter sensitivity is sufficient and study their actions. In summary, all these approaches support the need to evaluate architectural patterns, but they do not perform a sensitivity analysis, as we pursue in this manuscript.

Behavioral robustness.

Our research is motivated by understanding possible deviations in the values of design pattern parameters with respect to performance properties; this is achieved through PRIM and CART, which reduce parameters' variability. This approach is closely related to the notion of sensitivity and robustness, as recently renewed by Zhang et al. (2020). Robustness is defined as the largest set of deviations under which the system still guarantees a desired property. This concept has recently been redefined as a robustification objective (Zhang et al., 2023b), i.e., improving the system's design robustness (while minimizing the cost of architectural changes) against potential deviations. Zhang et al. (2023a) present a tool-based approach that solves a multi-objective problem to minimize changes in behavior and cost of different design modifications. Anevlavis et al. (2022) propose a robust linear-time temporal logic to reason about the different ways in which assumptions and guarantees can be violated in the system design space. Wei and Liu (2025) take a step further in connecting robustness to AI systems, defining it as the ability to cope with errors. The correspondence with the system design can be formulated as follows: an architecture is robust if the prediction of its quality is consistently accurate, even if one or more input variables or assumptions are changed due to unforeseen circumstances. This is highly relevant to our work, as we assess the sensitivity of the parameters to determine which variations in input values affect the performance characteristics of the patterns under analysis.

Overall, to the best of our knowledge, there exist several approaches that investigate the performance of pattern-based microservices architectures. However, our approach relies on a novel strategy that infers rules to support software architects in understanding and controlling pattern variability while targeting the preferred performance tradeoffs.

8. Conclusion and future work

This paper proposes a data-driven analysis framework, namely ADEPT, to understand the performance characteristics of microservice architectural patterns. Our framework provides a flexible pipeline that combines concepts from software architecture exploration, scenario discovery, and ML. This research contributes to the quantitative evaluation of pattern-based microservices. Software architects can leverage the features provided by ADEPT to gain insights into patterns' design space, thereby understanding the key decisions and parameters that affect the adoption of a given pattern, i.e., its performance tradeoffs, and how to control the variability of these tradeoffs.

We rely on Queueing Networks as analytical models to construct datasets for pattern performance. Our empirical results using PRIM and CART include different tradeoff definitions and parameter ranges, and report interesting performance variations. Both PRIM and CART seem to be effective in inferring parameter constraints. However, we recall that our approach is not tied to a performance modeling formalism or a particular definition of tradeoffs. The primary goal of ADEPT is to collect data about patterns and their performance indicators, including tradeoffs of interest, as the basis for sensitivity analysis and subsequent constraint inference. In this context, other performance models or even system traces can be used for the datasets and downstream analyses.

In future work, we plan to address the threats to the approach's validity, prioritizing the analysis of a larger set of architectural patterns and their combinations, including other application domains (e.g., federated learning systems (Compagnucci et al., 2026)), and to consider other quality indicators (e.g., reliability and security). We are interested in deriving design decisions with associated tradeoffs so that architects have a wider understanding of the alternative decisions at their disposal. Beyond *PRIM* and *CART*, we plan to evaluate additional techniques, such as Bayesian rule lists and explainable boosting machines (Wang et al., 2017; Lou et al., 2013), that can be useful for generating alternative explanations. The former can produce lists of if-then rules along with probabilities, allowing architects to assess the uncertainty of a parameter constraint; while the latter can derive rules capturing non-linear parameters' interactions. Besides, we plan to extend the stages of the ADEPT pipeline with textual explanations that can bridge the gap between the architect's technical vocabulary and the

numeric results from the SD and ML techniques currently used in the framework. To this end, we will explore the potential of Large Language Models. Finally, we aim to assess the actual adoption of architectural patterns in industrial case studies, thereby collecting further data to deepen our understanding of the variability of patterns' parameters and decisions.

CRediT authorship contribution statement

J. Andres Diaz-Pace: Writing – original draft, Validation, Software, Data curation, Conceptualization. **Catia Trubiani:** Writing – original draft, Validation, Methodology, Conceptualization. **David Garlan:** Writing – review & editing, Validation, Conceptualization.

Declaration of competing interest

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

Acknowledgments

This work has been partially funded by 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.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.jss.2026.113006.

Data availability

We provide a reproducibility package with the scripts, datasets, charts and metrics for all the patterns used in our experiments at: https://github.com/andresdp/patterns-sa/.

References

Ahmad, H., Treude, C., Wagner, M., Szabo, C., 2024. Smart HPA: A Resource-Efficient Horizontal Pod Auto-scaler for Microservice Architectures. In: 2024 IEEE 21st International Conference on Software Architecture (ICSA). pp. 46–57. http://dx.doi.org/10.1109/ICSA59870.2024.00013.

Akbulut, A., Perros, H.G., 2019. Performance Analysis of Microservice Design Patterns. IEEE Internet Comput. 23 (6), 19–27.

Amiri, A., Zdun, U., Hoorn, A.V., 2021. Modeling and Empirical Validation of Reliability and Performance Trade-Offs of Dynamic Routing in Service- and Cloud-Based Architectures. IEEE Trans. Serv. Comput. 15 (6), 3372–3386.

Anevlavis, T., Philippe, M., Neider, D., Tabuada, P., 2022. Being correct is not enough: efficient verification using robust linear temporal logic. ACM Trans. Comput. Log. 23 (2), 1–39.

Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J., 1984. Classification and regression trees.

Bryant, B.P., Lempert, R.J., 2010. Thinking inside the box: A participatory, computer-assisted approach to scenario discovery. Technol. Forecast. Soc. Change 77 (1), 34–49.

Compagnucci, I., Pinciroli, R., Trubiani, C., 2026. Experimenting architectural patterns in federated learning systems. J. Syst. Softw. 232, 112655. http://dx.doi.org/10.1016/j.jss.2025.112655, URL: https://www.sciencedirect.com/science/article/pii/S0164121225003243.

Di, P., Liu, B., Gao, Y., 2024. MicroFuzz: An Efficient Fuzzing Framework for Microservices. In: Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice. In: ICSE-SEIP '24, Association for Computing Machinery, New York, NY, USA, pp. 216–227. http://dx.doi.org/10.1145/3639477.3639723.

Di Francesco, P., Lago, P., Malavolta, I., 2019. Architecting with microservices: A systematic mapping study. J. Syst. Softw. 150, 77–97. http://dx.doi.org/10.1016/j.jss.2019.01.001, URL: https://www.sciencedirect.com/science/article/pii/S0164121219300019.

Diaz-Pace, J.A., Garlan, D., 2024. The architect in the maze: On the effective usage of automated design exploration. In: Proceedings of the 1st International Workshop on Designing Software. Designing '24, Association for Computing Machinery, New York, NY, USA, pp. 9–14. http://dx.doi.org/10.1145/3643660.3643947.

Diaz-Pace, J.A., Trubiani, C., Garlan, D., 2025. Data-driven understanding of design decisions in pattern-based microservices architecture. In: Software Architecture: 19th European Conference, ECSA 2025, Limassol, Cyprus, September 15–19, 2025, Proceedings. Springer-Verlag, Berlin, Heidelberg, pp. 276–293. http://dx.doi.org/10.1007/978-3-032-02138-0_18.

Friedman, J.H., Fisher, N.I., 1999. Bump hunting in high-dimensional data. Stat. Comput. 9 (2), 123–143.

Jamshidi, P., Pahl, C., Mendonça, N.C., Lewis, J., Tilkov, S., 2018. Microservices: The Journey So Far and Challenges Ahead. IEEE Softw. 35 (3), 24–35.

Khomh, F., Abtahizadeh, S.A., 2018. Understanding the impact of cloud patterns on performance and energy consumption. J. Syst. Softw. 141, 151–170.

Kwakkel, J.H., 2017. The exploratory modeling workbench: An open source toolkit for exploratory modeling, scenario discovery, and (multi-objective) robust decision making. Environ. Model. Softw. 96, 239–250. http://dx.doi.org/10.1016/j.envsoft.2017.06.054, URL: https://www.sciencedirect.com/science/article/pii/S1364815217301251.

Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C., 1984. Quantitative System Performance - Computer System Analysis Using Queueing Network Models. Prentice Hall.

Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J., Babar, M.A., 2021. Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. Inf. Softw. Technol. 131, 106449.

Liu, Y., Lo, S.K., Lu, Q., Zhu, L., Zhao, D., Xu, X., Harrer, S., Whittle, J., 2025. Agent design pattern catalogue: A collection of architectural patterns for foundation model based agents. J. Syst. Softw. 220, 112278.

Lou, Y., Caruana, R., Gehrke, J., Hooker, G., 2013. Accurate intelligible models with pairwise interactions. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '13, Association for Computing Machinery, New York, NY, USA, pp. 623–631. http://dx.doi.org/10.1145/2487575.2487579.

Ma, M., Lin, W., Pan, D., Wang, P., 2022. ServiceRank: Root Cause Identification of Anomaly in Large-Scale Microservice Architectures. IEEE Transanctions Dependable Secur. Comput. 19 (5), 3087–3100.

Martin, R.C., 2000. Design Principles and Design Patterns. Object Mentor 1 (34), 597.

Meijer, W., Trubiani, C., Aleti, A., 2024. Experimental evaluation of architectural software performance design patterns in microservices. J. Syst. Softw. 218, 112183.

Ngo, K.L., Mukherjee, J., Jiang, Z.M., Litoiu, M., 2022. Evaluating the Scalability and Elasticity of Function as a Service Platform. In: Proc. of the International Conference on Performance Engineering (ICPE). pp. 117–124.

Pinciroli, R., Aleti, A., Trubiani, C., 2023. Performance Modeling and Analysis of Design Patterns for Microservice Systems. In: Proc. of the International Conference on Software Architecture (ICSA). pp. 35–46.

Quéval, P.-J., Zdun, U., 2023. Extracting the Architecture of Microservices: An Approach for Explainability and Traceability. In: Proc. of the European Conference on Software Architecture (ECSA). pp. 346–353.

Rodrigues, H., Silva, A.R., Avritzer, A., 2025. Assessment of performance and its scalability in microservice architectures: Systematic literature review. J. Syst. Softw. 112500.

Sousa, T.B., Ferreira, H.S., Correia, F.F., 2022. A Survey on the Adoption of Patterns for Engineering Software for the Cloud. IEEE Trans. Softw. Eng. 48 (6), 2128–2140.

Thönes, J., 2015. Microservices. IEEE Softw. 32 (1), 116–116.

Tundo, A., Mobilio, M., Riganelli, O., Mariani, L., 2024. Monitoring Probe Deployment Patterns for Cloud-Native Applications: Definition and Empirical Assessment. IEEE Trans. Serv. Comput..

Vale, G., Correia, F.F., Guerra, E.M., Rosa, T.d., Fritzsch, J., Bogner, J., 2022. 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.

Wang, T., Rudin, C., Doshi-Velez, F., Liu, Y., Klampfl, E., MacNeille, P., 2017. A Bayesian framework for learning rule sets for interpretable classification. J. Mach. Learn. Res. 18 (1), 2357–2393.

Wei, W., Liu, L., 2025. Trustworthy distributed ai systems: Robustness, privacy, and governance. ACM Comput. Surv. 57 (6), 1–42.

Wijerathna, L., Aleti, A., Bi, T., Tang, A., 2022. Mining and relating design contexts and design patterns from Stack Overflow. Empir. Softw. Eng. 27 (1), 8:1–8:53.

Zhang, C., Dardik, I., Meira-Góes, R., Garlan, D., Kang, E., 2023a. 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.

Zhang, C., Dong, Z., Peng, X., Zhang, B., Chen, M., 2024. 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.

Zhang, C., Garlan, D., Kang, E., 2020. 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.

Zhang, C., Saluja, T., Meira-Góes, R., Bolton, M., Garlan, D., Kang, E., 2023b. Robustification of behavioral designs against environmental deviations. In: Proc. of the International Conference on Software Engineering (ICSE). pp. 423–434.