Actor-Driven Decomposition of Microservices through Multi-level Scalability Assessment

MATTEO CAMILLI, Politecnico di Milano, Italy
CARMINE COLARUSSO, University of Sannio, Italy
BARBARA RUSSO, Free University of Bozen-Bolzano, Italy
EUGENIO ZIMEO, University of Sannio, Italy

The microservices architectural style has gained widespread acceptance. However, designing applications according to this style is still challenging. Common difficulties concern finding clear boundaries that guide decomposition while ensuring performance and scalability. With the aim of providing software architects and engineers with a systematic methodology, we introduce a novel actor-driven decomposition strategy to complement the domain-driven design and overcome some of its limitations by reaching a finer modularization yet enforcing performance and scalability improvements. The methodology uses a multi-level scalability assessment framework that supports decision-making over iterative steps. At each iteration, architecture alternatives are quantitatively evaluated at multiple granularity levels. The assessment helps architects to understand the extent to which architecture alternatives increase or decrease performance and scalability. We applied the methodology to drive further decomposition of the core microservices of a real data-intensive smart mobility application and an existing open-source benchmark in the e-commerce domain. The results of an in-depth evaluation show that the approach can effectively support engineers in (*i*) decomposing monoliths or coarse-grained microservices into more scalable microservices and (*ii*) comparing among alternative architectures to guide decision-making for their deployment in modern infrastructures that orchestrate lightweight virtualized execution units.

CCS Concepts: • **Software and its engineering → Designing software**; **Software architectures**; **Software performance**; **Distributed systems organizing principles**; **Cloud computing**;

Additional Key Words and Phrases: Microservices, decomposition process, architectural patterns, performance analysis, scalability assessment

This work has been partly supported by the SARDECH project funded by the Free University of Bozen, Italy, and by the SISMA (Solutions for Engineering Microservices Architectures) project funded by the Italian Ministry of Research, PRIN no. 201752ENYB_002.

Authors' addresses: M. Camilli, Politecnico di Milano, Via Camillo Golgi 42, 20133, Milano, Italy; email: matteo.camilli@polimi.it; C. Colarusso and E. Zimeo, University of Sannio, Via Traiano 1, 82100, Benevento, Italy; emails: {ccolarusso, zimeo}@unisannio.it; B. Russo, Free University of Bozen-Bolzano, Piazza Università, 1, 39100, Bolzano, Italy; email: barbara.russo@unibz.it.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2023 Copyright held by the owner/author(s).

1049-331X/2023/07-ART117 $15.00
https://doi.org/10.1145/3583563

ACM Reference format:
Matteo Camilli, Carmine Colarusso, Barbara Russo, and Eugenio Zimeo. 2023. Actor-Driven Decomposition of Microservices through Multi-level Scalability Assessment. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 117 (July 2023), 46 pages.
https://doi.org/10.1145/3583563

1 INTRODUCTION

The microservices architectural style [1] is based on the so-called "share as little as possible" principle by emphasizing the concept of *bounded context* [2], that is, the identification of logical boundaries where particular terms, definitions, and rules apply in a consistent way. The adoption of this style leads to a higher decentralization (in terms of control and development) and to the adoption of continuous integration and deployment practices during the engineering life-cycle. This decentralization is an important enabling factor to increase the scale of operation. Each piece of functionality can be replicated on demand depending on the workload (i.e., the amount of concurrent users) and the relative frequency of invocation.

As reported by several leading companies (e.g., Amazon, Netflix, and Uber), the decomposition of monolithic systems into microservices is anything but trivial [3]. Enterprises that undergo this process often have the aspiration of increasing both performance and scalability in addition to reducing maintenance costs. Nevertheless, deriving a good decomposition into microservices is challenging. This often leads to nontrivial alternative architectural choices that need to be assessed in a quantitative way to ensure target quality attributes, such as performance and scalability, which represent key concerns in engineering microservices systems.

This is motivating a growing scientific interest in microservices from academia. Recent research is focusing on tackling challenges from both technical and organizational perspectives to guide developers, architects, and technical managers in making decomposition decisions. These decisions are often guided by design patterns to organize the data and access to services, while determining the proper granularity of the microservices and managing the distributed resources (either physical or virtual) are still severe issues [1, 4]. Here, wrong design choices during the decomposition process may affect important quality attributes of the system. For instance, improper decomposition might increase networking activity, leading to non-negligible communication overhead and subsequent performance degradation [1, 5].

Domain-Driven Design (DDD) represents the current mainstream approach to design microservices applications [6]. However, as described in [7], deducing microservices from domain models is challenging, especially when domain models are underspecified. DDD may lead to coarse-grained bounded contexts that might limit scalability under a constrained set of resources. This issue is even exacerbated in data-intensive applications[1] that usually operate on large and shared datasets, as discussed in [8]. Overall, the fundamental lack we address in this article is that *DDD ignores information about user behavior at runtime, thus preventing finer replication criteria from being applied.*

The state-of-the-art and practice in this area are still immature, as reported in [9], even if systematic assessment of performance and scalability is recognized as a painful activity by practitioners in the field [4, 10], especially for large-scale systems. To address these issues, we introduce a novel decomposition refinement strategy, henceforth referred to as *Actor-Driven Decomposition* (ADD). ADD complements existing approaches commonly used to derive the bounded contexts, such as DDD. Starting from an initial decomposition, our approach takes into account the actors of the

[1]Applications that are I/O bound or that process large volumes of data.

target application to split the service interfaces according to their role. The concept of actor (and role) received less attention than the notion of bounded context in decomposition approaches for microservices systems. Furthermore, there is a lack of knowledge on how actors should be integrated in established decomposition frameworks and whether the two approaches can be combined to increase the scale of operation.

To evaluate the ADD strategy, we propose a systematic and quantitative assessment approach able to guide engineers towards informed architectural decisions that enforce performance and scalability improvements within the expected production setting. According to [11, 12], historical data can be used to extract information about users thanks to the availability of online monitoring tools, such as OpenAPM.[2] The quantitative assessment makes use of load testing [12] in order to stress the system of interest under target operational settings synthesized from the actors interacting with the system in production. Evidence gathered at runtime is then analyzed using a multi-level approach, taking into consideration multiple granularity levels: *system*, *component*, and *operation*.

We adopted engineering research [13] to evaluate the ADD approach and multi-level scalability assessment by conducting an in-depth, detailed examination of two case studies[3] developed by leveraging modern technology stacks and deployment infrastructures: (1) a real smart mobility data-intensive system [14] and (2) an existing microservices e-commerce benchmark, called TrainTicket [15]. By using the proposed methodology, we were able to compare the ADD strategy with the results of the DDD one and to guide the tuning of resource allocation by exploiting feedback at different levels (system, component, and operation). We ultimately show that ADD improves the decomposition obtained with DDD and, therefore, leads to finer and more-scalable modularization of bounded contexts.

The key contributions of the article can be summarized as follows:

- Methodology for designing, deploying, and scalability tuning of microservices applications;
- Actor-driven decomposition of monoliths or bounded contexts into scalable microservices
- Approach for automated, multi-level assessment of scalability
- Application of the proposed methodology to a real application and an existing benchmark both deployed in modern infrastructures for containers orchestration

The remainder of this article is structured as follows. In Section 2, we introduce an overview of our methodology, preliminaries, and research questions. In Section 3, we describe the ADD strategy. In Section 4, we introduce our multi-level scalability assessment framework. In Section 5, we present our first case study, in which we applied our methodology to a real system in the domain of smart mobility. In Section 6, we present our second case study, in which we applied our methodology to a well-known microservices e-commerce system benchmark. In Section 7, we answer the research questions and we discuss threats to validity. In Section 8, we present the related work. In Section 9, we present our conclusions and future research directions.

2 METHODOLOGY AND PRELIMINARIES

In this section, we introduce an overview of our methodology (Section 2.1), the research questions we aim to answer through an experimental campaign (Section 2.2), and the background notions used in the rest of the article (Section 2.3). We briefly review Domain Driven Design; the

[2] https://openapm.io/.

[3] The dataset and the scripts that can be used to replicate the experiments are publicly available at https://github.com/zerodayshack/Microservices-Performance-Assessment.

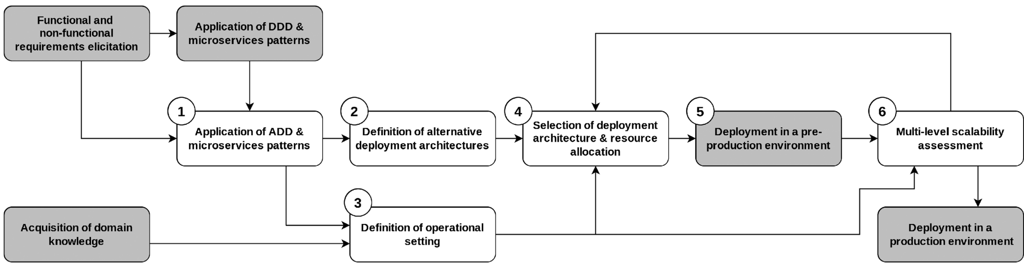


Fig. 1. Overview of the methodology.

notion of architecture, component, and deployment; the notion of scalability requirement; scaling dimensions; the notion of actor; and the operational profile.

2.1 Methodology Overview

Figure 1 illustrates the high-level building blocks of the methodology as well as the dataflow among them. White boxes represent the main activities that we present as part of our contribution. Gray boxes refer to existing practices often applied along the software life-cycle, especially in the context of microservices systems.

As shown in Figure 1, we assume the existence of functional and nonfunctional requirements, elicited for the purpose of designing and implementing the application of interest, and the domain knowledge about the actors and their role in terms of used functions provided by the application. Among the possible nonfunctional requirements (e.g., scalability, reliability, resilience, security), our methodology mainly focuses on performance and scalability.

We assume that existing methodologies, such as DDD, as well as microservices design patterns (e.g., aggregate, and subdomains), have been applied to identify a possible initial decomposition of the application into a set of coherent bounded contexts that separate the application functions according to the entity model characterizing the domain of interest. At this point, we refine[4] the bounded contexts by ① applying the ADD strategy. We still apply well-known microservices patterns [6] (e.g., application programming interface (API) gateway, shared database, database per service) and we possibly complement them with additional patterns (e.g., command query responsibility segregation, saga, events sourcing) to take into account the actors and their role in the target system.

As outcome of ADD, engineers ② define a set of alternative *deployment architectures* specifying the components, their relations, and the mapping to execution units. These deployment architectures further decompose the services previously identified through DDD, with the ultimate goal of maintaining or even increasing the scale of operation. The evaluation of the deployment architectures starts with the ③ definition of the *operational profile* of the application to characterize the expected workload in the production environment. This profile is defined by engineers according to the set of actors identified by ADD, the operations invoked by the actors, and the expected workload intensity for each actor. This latter piece of information can be either manually defined by engineers according to an existing Service Level Agreement (SLA) or mechanically derived from execution traces collected by observing similar systems or previous versions of the system in a given observation period.

[4]Note that starting from a monolithic system rather than refining the results of DDD is possible but undesirable, especially if additional quality properties other than scalability are important (e.g., maintainability). Indeed, without the boundaries identified by domain entities, the reuse of (micro)services could be hard.

For each deployment architecture, an initial ④ resource allocation must be defined. The allocation maps the available resources to execution units taking into account the computational complexity of the functions running on such units. At this stage, we aim at ensuring an acceptable[5] response time for the operations exposed by microservices under a baseline workload. After tuning the resource allocation, the system is ⑤ deployed in a staging (or pre-production) environment to test the application using the ⑥ multi-level scalability assessment. This latter stage automatically executes testing sessions replicating the operational setting. Then, it guides engineers in understanding and improving the scalability of the selected deployment architecture(s) at different levels of abstraction: *system*, *component*, and *operation* levels. The outcome of the multi-level assessment is then used to select the alternative(s) that better fits the expected operational condition. The resource allocation of the selected architecture(s) can be further tuned according to the outcome of the assessment stage. This triggers a new assessment iteration, as shown in Figure 1. The final outcome of the methodology is a deployment architecture that, at the end of the iterations, yields better scalability. With this architecture, the system is then deployed into the production environment.

2.2 Research Questions

To demonstrate the applicability and effectiveness of the proposed methodology, we identify the following Research Questions (RQs), which we answer in the next sections.

RQ1: *To what extent can our scalability assessment workflow support decision-making over alternative microservices architectures at the system level?*
We use a high-level scalability metric to compare architecture alternatives and evaluate the performance of the whole system under increasing loads. In doing so, we aim to understand whether this metric and our scalability assessment can be integrated in a decision-making process, such as decision gates before the deployment of a release build.

RQ2: *To what extent can our scalability assessment workflow support resource allocation improvement at the microservice (or component) level?*
We aim to understand whether our scalability assessment can spot issues at each microservice (i.e., component level) and whether the localization of scalability issues can be used to improve the components' performance by reallocating the available resources.

RQ3: *Is the scalability assessment workflow able to spot scalability and performance issues at the operational level?*
We aim to understand whether our scalability assessment can identify critical microservice operations that, according to their performance behavior, prevent the system from increasing the scale of operation.

RQ4: *Can ADD practically drive further decomposition of bounded contexts yet enforce performance and scalability improvements?*
We aim to understand whether ADD is a practical method able to improve the decomposition obtained through DDD in terms of scalability. Thus, ADD can effectively complement DDD to obtain microservices with finer granularity yet enforce performance and scalability improvements.

2.3 Preliminaries

2.3.1 Domain Driven Design. The DDD approach [6] is the de-facto standard to design microservices applications. For this approach, a service must be small enough to be developed by a small team and to be easily tested. The main idea behind this kind of decomposition is to limit the impact

[5] Generally, "acceptable" is defined according to usability engineering practices and depends on the application domain [16].

of changed requirements to a single service (*Single Responsibility Principle*), thus also reducing the overhead of team management that we can observe in the case of cross-service impacts. To this end, the domain is initially decomposed in sub-domains by identifying the *business capabilities* characterizing the domain, and their boundaries insulate the so-called *bounded contexts* [6, 17].

To design bounded contexts and materialize them into microservices, existing tactics may be adopted (e.g., aggregate, entity, root entity, value object). These tactics mainly aim to ensure the identification of transactional invariants to be used as a non-distributed data model, thus avoiding the complexity and inefficiency of distributed transactions.

Due to their cohesive structure, the microservices originating from bounded contexts are often used by a variety of actors that, according to their role, execute different sets of operations, causing a non-uniform workload directed to the exported operations. This may cause hot spots in the API of a bounded context, often forcing the replication of the whole context to satisfy the performance requirements. Bounded contexts improve software maintenance, yet they may limit the scale of operation when the actors are not considered in the decomposition process.

2.3.2 Architecture, Components, and Deployment. In this work, we stick to the definition of software *architecture* (or simply architecture) introduced in [18]. An architecture α is defined as a set of *components* C and a set of *relationships* R among them and with the environment, where components are domain-specific software elements. Software components mapping to the execution units is henceforth referred to as *deployment*. In our context, we also use this term to describe the mapping from component containers to operating system containers (e.g., Docker containers) or virtual/physical machines.

While software components and relationships among them are defined according to the functional requirements, the deployment allocates resources and consequently influences non-functional properties, such as performance and scalability. In modern infrastructures, deployments can be complex due to the presence of different virtualization layers: application components are typically hosted by frameworks accessible as servers —for example, application servers, database management systems (DBMSs), and the like — whereas servers are hosted by lightweight containers or/and virtual machines. Often, some lightweight containers are deployed as a whole in a logical unit called a *pod*. In the rest of the article, we use the term $deployment_{sp}$ to indicate the deployment of servers to pods and the term $deployment_{pm}$ for any deployment of pods to physical or virtual machines. Finally, we refer to *deployment architecture* to indicate a software architecture including both $deployment_{sp}$ and $deployment_{pm}$. The set of alternative deployed architectures is henceforth denoted by DA.

Microservices applications can be considered as a specialization of component-based software applications in which each component is hosted and managed by a dedicated executing unit, typically exposing CRUD[6] operations by means of RESTful APIs. This approach often yields advantages such as fine-grained replication for scalability, independent fault management, resilience, and selective versioning at runtime. More formally, we refer to a microservice application as set S of independently deployable components (microservices) operating through a set of operations O exposed by the API.

2.3.3 The Scaling Dimensions. The Scale Cube conceptual model [19, 20] defines three main dimensions over which system scalability improvements can be achieved. The three dimensions of the cube describe a method for scaling systems. Single instance (or monolithic) applications are located at $(0, 0, 0)$, which is the point that represents the lowest scaling capability. Scalability improvements can be achieved by moving an application along the three axes towards the

[6] CRUD stands for "Create, Read, Update, and Delete," the four basic operations applied to a persistent storage.

point $(1, 1, 1)$, which represents an ideal cloned, decomposed, and partitioned solution. The three dimensions represent actions that one may perform taking into account complementary aspects: horizontal replication (x-axis); functional decomposition (y-axis); and data partitioning (z-axis).

x-Axis scaling consists of instantiating multiple copies of services or applications behind a load balancer; this is typically performed with containerization. y-Axis scaling requires substantial (re)design effort since it decomposes the business logic into different services by deploying each one into different units of execution responsible for only a subset of the data. The z-axis scaling takes into account the database(s) and is typically used to partition the data across the services. Each service replica deals with only a subset of the data improving transaction scalability. The design and implementation of a partitioning scheme is anything but trivial, especially when the decomposition of the business logic is coarse and services are responsible for many operations related to one or more domain entities.

In this article, we aim to improve the effect of cloning (x-axis) by introducing an additional decomposition strategy on the y-axis, which allows for a more selective replication of microservices, and related patterns that enforce data separation (z-axis), taking into account the expected load generated by different actors of the system.

2.3.4 The Scalability Requirement. The *scalability requirement* quantifies the capability of the system to handle loads while maintaining performance within a desired range [21]. The requirement is defined as a pass/fail *scalability threshold*, which allows failing and successful *operation*s to be detected at runtime according to their average response time.

To determine the threshold, we first select an operational setting as described in Section 2.3.5. Then, we identify a reference load λ_0 and a deployment architecture α_0 for which the system is expected to be responsive (baseline setting). Under this configuration, the System Under Test (SUT) is executed to compute the mean μ_j^0 and standard deviation σ_j^0 of the response time of each operation o_j. We then define the *scalability threshold* Γ_j^0 for each operation o_j as

$$\Gamma_j^0 = \mu_j^0 + 3 \cdot \sigma_j^0. \tag{1}$$

During any other test session with target setting (λ, α), we compute the mean response time μ_j for each operation o_j. We say that the operation o_j *fails* if

$$\mu_j > \Gamma_j^0 \tag{2}$$

and it *succeeds*, otherwise. Equation (2) can be explained by means of the Chebyshev inequality [22, 23], a version for non-parametric distributions of the well-known $3 \cdot \sigma$ empirical rule. Such an inequality states that more than 88.8% of values in a univariate distribution lie within three standard deviations of the mean. This heuristic allows us to spot whether the average response time of o_j, under the architecture α, is an outlier of the distribution observed using the baseline setting.

During a test session with target setting (λ, α), we measure the *scalability share* of an operation o_j by leveraging the pass/fail criterion in Equation (2):

$$s_j = \nu_j \cdot \delta_j, \tag{3}$$

where δ_j is the Dirac function [24] that denotes whether o_j has passed ($\delta_j = 1$) or failed ($\delta_j = 0$) the test and ν_j is the frequency of invocation of operation o_j over all invocations to all operations of the SUT. Equation (3) estimates the conditional probability that operation o_j succeeds given deployment architecture α and load λ.

The sensitivity of scalability metrics based on the pass/fail criterion in Equation (2) has been studied in [12]. The study demonstrates the adequacy of the criterion in detecting scalability

Table 1. Example of Actors and Usage Profile

Actor	Behavior model	Behavior mix
a$_1$	$bm_1 = \{ o_1 \xrightarrow{t} o_2 \ldots \xrightarrow{t} o_l \}$	ω_1
	$bm_2 = \{ o_1 \xrightarrow{t} o_3 \ldots \xrightarrow{t} o_m \}$	ω_2
a$_2$	$bm_3 = \{ o_5 \xrightarrow{t} o_8 \ldots \xrightarrow{t} o_n \}$	ω_3

degradation at the fine-grained operational level. If the system is not under stress (relatively low load), the metric is not sensitive to the threshold value. If the system is under stress, the metric is sensitive to the threshold because of the degradation in performance observed for these relatively high load levels.

2.3.5 Operational Setting. The operational setting includes the operating conditions used to test a given SUT, taking into account the semantic constraints imposed by each actor to the sequences of operations that can be invoked. Let $\mathcal{A}$ be a set of actors[7] given a set of operations $O = \{o_1, o_2, \ldots, o_n\}$ and let the *covering* of O as $C(O) = \cup_{a \in \mathcal{A}} O_a$, where O_a is the set of operations allowed for actor $a \in \mathcal{A}$; we then define the operational setting (extending the one defined in [26]) as the following set of specifications.

- The *workload specification model*: Given a set of operations $O = \{o_1, o_2, \ldots, o_n\}$ exposed by the SUT API, a set of allowed sequences of operations and their invocations through the APIs along with the specification of valid requests (relative paths of the requests, parameters, and constraints).
- The *actor workload specification model*: A subset O_a of the workload specification model O invoked by a specific actor $a \in \mathcal{A}$.
- A set of *behavior models*: A set of user behaviors $\{bm_1, \ldots, bm_n\}$, each one specified by a sequence in the actor workload specification model, along with a pseudo-random thinking time t between subsequent invocations.
- The *behavior mix*: The set of frequencies $\{\omega_1, \ldots, \omega_n\}$, each approximating the probability of drawing the corresponding behavior model during the generation of a *load*, λ, (i.e., the number of concurrent users interacting with the SUT).
- The *usage profile*: The set of behavior models with their behavior mix:

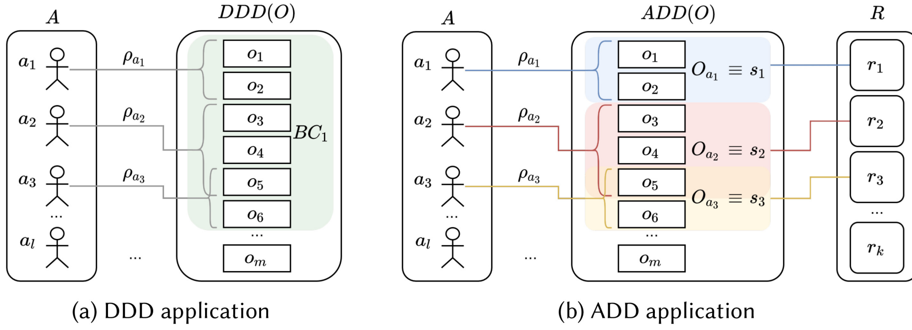
$$\Omega = \{\langle bm_1, \omega_1 \rangle, \ldots, \langle bm_h, \omega_h \rangle\}. \tag{4}$$

- The *operational profile*: The probability distribution of load. This distribution can be discretized by *data binning*, which groups continuous values into a number of bins to obtain a discretized distribution f of selected loads $\Lambda = \{\lambda_1, \ldots, \lambda_s\}$ and their frequencies:

$$f(\Lambda) = \{f(\lambda_1), \ldots, f(\lambda_z)\}. \tag{5}$$

Similarly, as the *workload intensity* ρ (i.e., the total number of requests directed to the system during a time interval T) depends on the number of concurrent users and their behavior, we can write $\rho = \sum_{a \in \mathcal{A}} \rho_a$, where ρ_a is the workload intensity generated by the concurrent users who are instances of actor $a \in \mathcal{A}$. Therefore, each actor interacting with the system is specified by one or more elements in the *usage profile* as exemplified in Table 1. The usage profile can be either defined by operators and developers based on their prior knowledge or mechanically derived by analyzing the historical data of the system using off-the-shelf monitoring tools [11, 27].

[7] According to the Unified Modeling Language (UML), an actor "specifies a role played by a user or any other system that interacts with the subject" [25].



(a) DDD application (b) ADD application

Fig. 2. DDD refinement through Actor-Driven Decomposition applied to operations O.

3 ACTOR-DRIVEN DECOMPOSITION OF MICROSERVICES

In this section, we first present the ADD strategy (Section 3.1) and then introduce some architectural patterns we identified as useful guidelines to follow when data partitioning is not possible to break down a bounded context into smaller pieces (Section 3.2).

3.1 Actor-Driven Decomposition Strategy

As anticipated in the previous sections, ADD complements and refines the results of DDD with the goal of improving scalability. According to the Scale Cube model in Section 2, ADD drives the design of a target application towards a more selective horizontal replication (x-axis scaling) through a refinement of functional partitioning and data partitioning (y-axis and z-axis scaling, respectively). While DDD usually decomposes by noun-preserving cohesion of domain entities, considering the actors (and their roles) takes a complementary perspective and identifies possible finer splitting according to usage. ADD does not prevent engineers from applying recommended practices, such as the single responsibility principle or aggregate roots. It starts from a definition of bounded contexts and considers possible decoupling based on how end-users interact with their operations. According to [28], we can see ADD as an additional tactical pattern to identify further boundaries (that do not violate the existing ones) considering non-functional requirements.

This approach has the potential of increasing the operation scale by selectively replicating finer units according to actors' needs. The idea is to create a logical separation of the operations belonging to the same bounded context based on how the actors are going to use them. In this case, we can selectively increase/decrease computational resources to different groups of operations according to the load generated by the actors.

Figure 2 shows a high-level schema of DDD refinement through ADD. Given the bounded context BC_1 in Figure 2(a), which corresponds to a coarse-grained microservice (derived from a bounded context) having its own API exposing $o_1, \ldots, o_6$, each actor in $\mathcal{A} = \{a_1, \ldots, a_4\}$ generates a different workload whose intensity is ρ_{a_i} for all i. Each workload intensity is directed to a subset of the operations depending on the behavior model of the corresponding actor. For instance, ρ_{a_2} is directed to $\{o_3, o_4, o_5\}$, while ρ_{a_3} is directed to $\{o_5, o_6\}$. Since different workloads are handled by the same service, the whole service needs to be replicated even in cases in which only one of these workloads is heavy.

As shown in Figure 2(b), the behavior model of the actors yields the covering $C(O) = \{O_{a_1}, O_{a_2}, O_{a_3}\}$, that identifies a logical decomposition into the services $S = \{s_1, s_2, s_3\}$. This refinement of the original service can better exploit the available resources $r_1, \ldots, r_k$ since they can be assigned to more granular services according to the generated workloads. The identification of the covering $C(O)$ requires the following prior knowledge that architects typically have at design-time:

(1) operations exported by each service (e.g., Swagger OpenAPI documentation[8]) and (2) the actors using them. The logical decomposition S induced by $C(O)$ meets the DDD refinement guidelines suggested by tactical patterns introduced in [17, 28]:

- Each service has a single responsibility since it is a DDD service for a specific actor.
- There are no chatty calls between services since execution traces do not increase their complexity (they are essentially the same but possibly handled by different services according to the actors who issue the requests).
- Each ADD service is even smaller than the original one; therefore, it can be built by a small team working independently.
- There are no additional interdependencies that require two or more services to be deployed in lock-step.
- Services are not tightly coupled and can evolve independently according to the evolution of the actors.

As shown in Figure 2(b), a covering $C(O)$ may lead to partially overlapping sets, such as O_{a_2} and O_{a_3}. In this case, alternative strategies to define services are possible. In particular, the operations in the intersection:

- can be replicated in different microservices;
- can be included in a larger microservice merging the partially overlapping sets of operations;
- can be implemented by a dedicated microservice that is then used by more than one actor.

Binding the final decision to a single choice without proper understanding of the corresponding impact on quality attributes is dangerous and could lead to violation of the scalability requirement (Equation (1)) in production. For this reason, whenever the architects have a set of possible alternative architectures, proper and systematic assessment of them shall be carried out.

Let $\hat{r}_i$ be the amount of resources allocated to service s_i that ensures the satisfaction of the scalability requirement in the case of reference load λ_0. Then, let us consider a generic load $\lambda > \lambda_0$ such that λ generates the workload intensity $\rho = \sum_i \rho_{a_i}$. According to the logical decomposition described earlier, the amount of resources allocated to each service s_i shall be proportional to the workload intensity ρ_{a_i} of the corresponding actor. The mapping from available resources to services is denoted by $\mathcal{M}$[9] and defined as follows:

$$\mathcal{M} = \{\hat{r}_1 \cdot \lceil \Delta(\rho_{a_1}) \rceil, \hat{r}_2 \cdot \lceil \Delta(\rho_{a_2}) \rceil, \ldots, \hat{r}_l \cdot \lceil \Delta(\rho_{a_l}) \rceil\}, \tag{6}$$

where $\Delta(\cdot)$ represents a scaling factor expressed as a function (either linear or nonlinear) of the relative workload intensity, which depends on the number of concurrent users.

The scaling factor function Δ is usually implemented by means of autoscaling components that automatically adjust the amount of resources per service at runtime. Let us consider an upper-bound r in the total amount of available resources and the workload intensity $\bar{\rho} = \sum_i \bar{\rho}_{a_i}$ such that all of the resources are saturated by the function Δ. Under this condition, the autoscaling component usually follows the distribution of the workload intensity, that is, the amount of resources for each service s_i is proportional to the workload intensity $\bar{\rho}_{a_i}$. Thus, the mapping is approximately as follows:

$$\mathcal{M} = r \cdot \left\{ \left\lceil \frac{\bar{\rho}_{a_1}}{\bar{\rho}} \right\rceil, \left\lceil \frac{\bar{\rho}_{a_2}}{\bar{\rho}} \right\rceil, \ldots, \left\lceil \frac{\bar{\rho}_{a_l}}{\bar{\rho}} \right\rceil \right\}, \tag{7}$$

[8] https://swagger.io/.

[9] Automatically generated by an autoscaler or manually assigned by an operator.

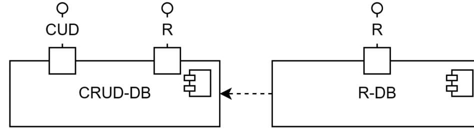


Fig. 3. Actor-driven CRUD decomposition and database replication.

taking into account the constraints imposed by the available resources on each virtual or physical machine. These constraints could prevent the ideal mapping from being applied due to the presence of nondivisible resources.

This condition represents an important scenario that pushes microservice performance to its limit. Testing the target system under these circumstances is required to study the total amount as well as the distribution of resources that guarantee the satisfaction of the expected maximum load defined, for instance, by an SLA. Our methodology systematically replicates these circumstances in controlled experiments to study the extent to which alternative architectural options (driven by ADD) improve the scalability of the target system under different distributions of the workload intensity $\bar{\rho}_{a_i}$.

Even though the focus of our work is on performance and scalability, it is worth noting that the increased granularity level obtained through ADD may lead to better fault localization and higher reliability. For instance, let us consider BC_1 in Figure 2(a) and $O_{a_1}, O_{a_2}, O_{a_3}$ in Figure 2(b) obtained through DDD and ADD, respectively. While ADD yields better distribution of the workload, in the DDD version the workload of three actors ($\bar{\rho}_{a_1}$, $\bar{\rho}_{a_2}$, and $\bar{\rho}_{a_3}$) may generate a substantial amount of traffic directed to a unique service. In this case, BC_1 may become a scalability bottleneck, prone to server errors due to saturation of resources, thus reducing the reliability of the system [29].

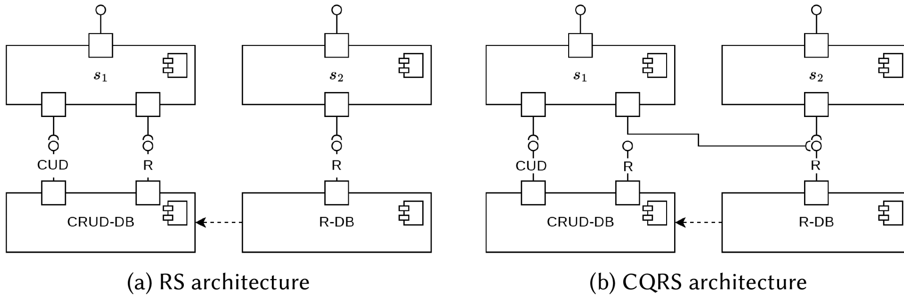
3.2 Databases and Stateful Services Decomposition

The decomposition of a bounded context into smaller pieces may lead to difficulty in splitting the database due to the high cohesion of the related schema, leading to a shared database solution that may limit scalability. Here, we propose two different approaches that can be applied in different situations according to the chracteristics of the services identified by ADD.

3.2.1 CRUD Decomposition and Database Replication. The first option we consider works at the functional level by separating write-and-read operations. A database can always be separated in two types of instances as shown in Figure 3:

- `CRUD-DB`, with a full feature database that exposes the `CUD` (Create, Update, and Delete) and `R` (Read) interfaces to manage all operations (or only CUD ones);
- `R-DB`, optimized for read-only operations. It exposes the `R` interface that manages only read operations performed on instances of the database obtained by periodically synchronizing `CRUD-DB` transactions.

As illustrated in Figure 3, `R-DB` depends on `CRUD-DB` since each successful CUD transaction must be propagated to all of the read-only instances to enforce data consistency. This is typically supported by modern DBMSs or can be implemented through a publish/subscribe connector between the instances. `CRUD-DB` is used by microservices that expose both write and read operations, whereas `R-DB` is used by microservices that expose read operations only. As an example, let us consider the set of operations O exposed by a microservice (bounded context) of a smart mobility application. The microservice handles road networks and is used by the administrator actor a_1, who can both read available roads and insert new roads, and the end user a_2, who can read only the existing roads to calculate shortest paths. According to the ADD strategy, the service can be



(a) RS architecture (b) CQRS architecture

Fig. 4. Microservices architectures with CRUD decomposition.

decomposed according to the covering $O = \{O_{a_1}, O_{a_2}\}$. Since O_{a_2} contains read-only operations, the corresponding microservice s_2 will expose a subset of the original API and will communicate only with the R-DB instances.

Figure 4 illustrates two alternative architectural patterns to decompose a microservice whose operations are used by different actors. The description of the patterns follows:

- *Role Separation* (RS). As shown in Figure 4(a), both read and write operations of actor a_1 are directed to the CRUD-DB; thus, this actor does not act on R-DB. The two microservices are separated at each level, with a weak interaction at the DB level due to the need for propagating changes from CRUD-DB to R-DB.
- *Command and Query Responsibility Segregation* (CQRS). Figure 4(b) shows this pattern, inspired by the one originally introduced in [6]. Following this approach, we change the architecture at the DB level by forwarding the *CUD* operations towards CRUD-DB and the *R* ones towards R-DB. Even if there is a link between s_1 and the R-DB used by actor a_2, this architecture does not violate separation since a dedicated R-DB can be assigned to each instance of s_1. This architecture requires careful management of data propagation to avoid data inconsistency issues. For example, a write operation performed by an instance of actor a_1 could not be seen by a short time subsequent read operation from the same client. All of these considerations apply not only to databases but also to any stateful service.

3.2.2 Database Partitioning. Another option we propose is to split the data of a single database into different database instances having the same schema. This approach can be applied when each actor accesses a disjoint subset of the data contained in the original database. As an example, consider an e-commerce application used to sell train tickets of different types, such as high-speed train tickets and other tickets. Users buying other tickets typically generate a heavier workload than buyers of first-class tickets. In this case, rather than maintaining a single database for all of the tickets, we may split the data into two instances. This separation, again, enables a differentiated replication or resources allocation depending on the expected or actual load generated by high-speed train ticket buyers and other buyers.

As illustrated in Figure 5, the workload generated by $a_1, \ldots, a_n$ is directed to different services exposing the same API. Each one of these services works on its own database instance. The databases $p_1, \ldots, p_n$ have the same schema and their disjoint union gives the data contained in the original database.

4 MULTI-LEVEL SCALABILITY ASSESSMENT

To support decision-making while applying the ADD approach, we present a methodology for scalability assessment of alternative architectures that takes into account both logical and deployment relationships. In the following, we first introduce a high-level overview of the workflow

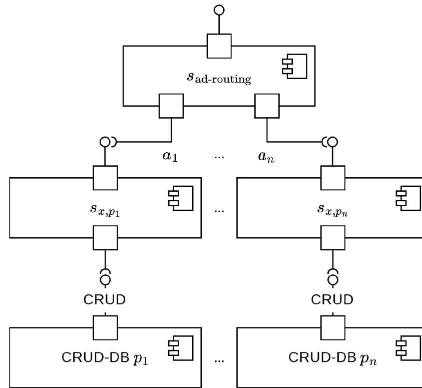


Fig. 5. Actor-driven database partitioning.

to guide the decisions of architects (Section 4.1). Then, we dive deeper into the definition of the operational setting (Section 4.2), the measurement workflow (Section 4.4), and the analysis workflow (Section 4.5).

4.1 Overview

The objective of our assessment approach is to guide architects on deployment architecture selection, in the set of all available alternatives *DA*, that better fits the operational setting in terms of performance and scalability.

Figure 6 shows an overview of the main stages: ① *derivation of the operational setting*, ② *load testing*, ③ *measurement framework*, and ④ *analysis workflow*.

The operational setting derived in stage ① describes the system usage and is typically obtained from operational data generated during system execution and SUT specification [11, 30]. In stage ②, load tests are designed taking into account a target operational setting and alternative deployment architectures. The tests are then automatically executed to collect response times of invocations to each SUT operation. In stage ③, raw data are automatically processed to compute a set of metrics according to our measurement framework. In stage ④, the analysis workflow guides engineers to informed decisions by means of a cockpit that displays plots and measures tailored to the questions of interest.

4.2 Derivation of the Operational Setting

In this stage, the operational setting (Section 2.3.5) is determined by combining the information on SUT specification (e.g., the allowed sequences of operations and invocations from the RESTful APIs for the workload specification model) with the historical data generated by the SUT during its operations (e.g., the frequency of occurrence of a certain behavior model). The usage profile can be manually defined or automatically derived from operational data. The manual definition of the usage profile is typically conducted by deriving user-application interaction patterns of relevant scenarios. The automatic extraction is performed on historical data and is carried out using process mining [31], or clustering algorithms, such as the WESSBAS approach [27]. For instance, for a cloud provider delivering an application as a service, the load and behavior mix represent the precondition for defining the guarantee terms of an SLA, which specifies Service Level Objects (SLOs), such as response time, against the declared preconditions [32].

The operational profile is typically determined by observing the target system during operations in a certain period of time. The load is periodically recorded and its frequency is computed. The load values in a given interval (e.g., 0–300 in Figure 7(a)) are binned to obtain the discretized

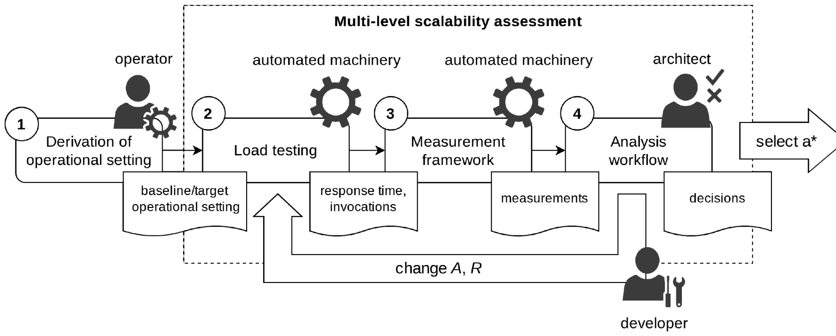
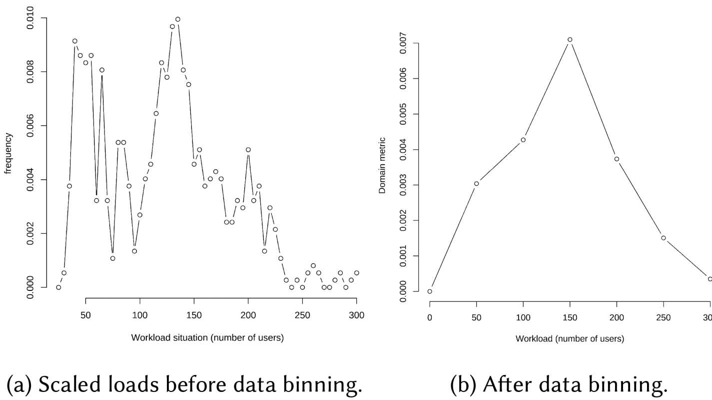


Fig. 6. High-level overview of the scalability assessment.



(a) Scaled loads before data binning. (b) After data binning.

Fig. 7. Operational profile.

distribution over a set of equally distant load values $\lambda_1, \ldots, \lambda_k \in \Lambda$. The cumulative frequency between each two consecutive loads is then assigned to the smaller one. For instance, the result of the binning process on the data in Figure 7(a) is the polygon illustrated in Figure 7(b).

4.3 Load Testing

As shown in Figure 8, this stage includes two load testing sessions to collect the response time of the invocation to each SUT operation. For both sessions, the usage profile is the same. The former is referred to as the *baseline test session* and the SUT is executed under a baseline deployment architecture α_0 and load λ_0 for which the SUT is expected to operate with acceptable performance as described in Section 2.3.4. During this session, the response time of each operation is collected and the *scalability threshold* in Equation (1) is computed. The latter session is referred to as *target test session* and it tests the SUT under a target operational profile defined by the selected loads Λ and the alternative deployment architectures DA. For each pair $(\lambda, \alpha) \in \Lambda \times DA$, a test is executed.[10] During each test, all invocations to SUT operations and corresponding response times are collected. The outcome of each test is the mean response time and the invocation frequency for each SUT operation.

According to our definition of architecture in Section 2, α also includes deployment aspects: deployment of servers to pods ($deployment_{sp}$) and deployment of pods to physical or virtual machines

[10]Multiple parallel tests are desirable (to speed up the process) and feasible as long as each test can replicate the same amount of resources available in the production environment without interfering with each other.

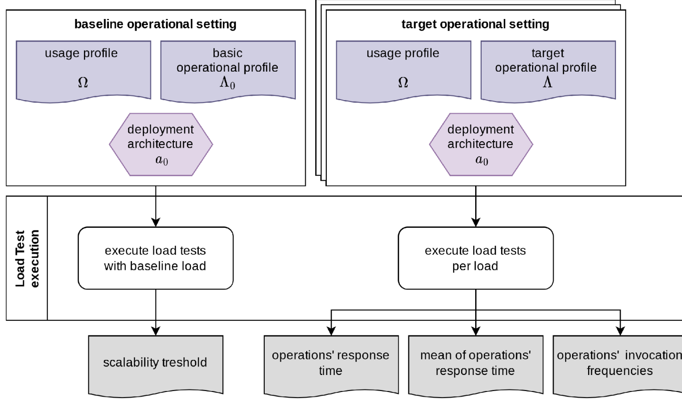


Fig. 8. Load testing.

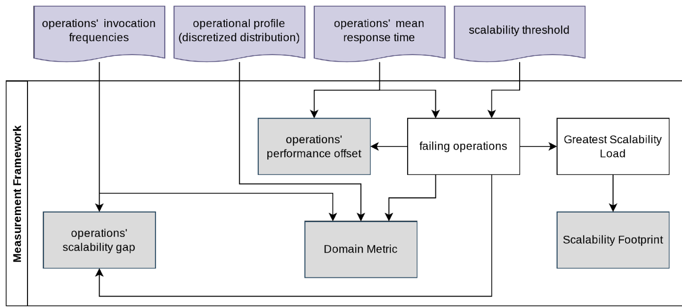


Fig. 9. Measurement framework.

($deployment_{pm}$). This means that the results of the tests also depend on the adopted infrastructure, that is, scalability requires the identification of proper separation boundaries in the software architecture to enable an effective exploitation of the underlying (scalable) infrastructure.

4.4 Measurement Framework

This stage is fully automated and follows the *measurement framework* illustrated in Figure 9. It starts with the high-level goal of *assessing the SUT scalability in terms of its capability of meeting the performance requirement under increasing load*. It processes the response time and invocation frequency of all operations to provide four metrics for the analysis stage: the *(relative) Domain Metric*, *scalability footprint*, *scalability gap*, and *performance offset* described in the following.

Relative Domain Metric. The relative *Domain Metric* measures the overall scalability of a deployment architecture at a given load, as described in [12]. It represents the probability that the SUT with deployment architecture α does not fail under a given load $\lambda \in \Lambda$ and is computed as follows:

$$\mathcal{DM}^{\alpha}(\lambda) = f(\lambda) \cdot \sum_{j=1}^{n} s_j^{\alpha}(\lambda), \tag{8}$$

where $f(\cdot)$ is the discretized distribution of loads (Section 2.3.5), n is the number of operations, and $s_j^{\alpha}(\lambda)$ is the scalability share of o_j (Equation (3)). When no operation fails with load λ, $\mathcal{DM}^{\alpha}(\lambda)$ is equal to $f(\lambda)$ and is less than $f(\lambda)$ otherwise. This difference measures the *scalability degradation*

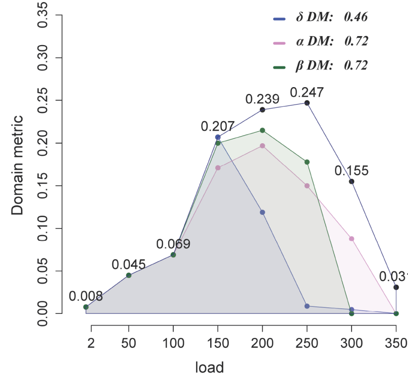


Fig. 10. Plots of relative $\mathcal{DM}$ over loads and value of the total DM for three deployment architectures. The difference between α ■ and β ■ cannot be distinguished using the quantitative (DM value) or qualitative comparison (plots).

due to the failed operations with load λ:

$$\mathcal{SD}^{\alpha}(\lambda) = f(\lambda) - \mathcal{DM}^{\alpha}(\lambda) = f(\lambda) \cdot \sum_{j \in \mathcal{S}} (1 - s_j^{\alpha}(\lambda)) \tag{9}$$

where $\mathcal{S}$ is the set of indices of the operations that fail. Thus, a failing operation contributes with its scalability share (Equation (3)) to the overall scalability degradation of the SUT. At each load, performance degradation can be visualized as the gap between the plot of the relative Domain Metric (outermost polygon) and the one of the discretized distribution (inner polygon), as shown in Figure 10. Internal polygons approaching the outermost line yield scalability closer to optimal from a system-level perspective.

By applying the Bayesian rule, we can also compute the total *Domain Metric*, as follows:

$$\mathcal{DM}^{\alpha} = \sum_{\lambda \in \Lambda} f(\lambda) \cdot \sum_{j=1}^{n} s_j^{\alpha}(\lambda). \tag{10}$$

$\mathcal{DM}^{\alpha}$ provides engineers with a single value that measures the overall SUT scalability with the deployment architecture α [33].

Even though the total Domain Metric represents an effective instrument to decide over different deployment architectures, in some cases it may not explain subtle differences; thus, further investigation might be necessary. Figure 10 illustrates an example of such a situation. The two deployment architectures α and β have the same total Domain Metric (0.72) but different scalability behavior over loads. To understand what is better in these cases, we need to analyze the system at a lower abstraction level. To this end, we introduce the Scalability Footprint, which represents the scalability capability per individual operation.

Scalability footprint. The scalability footprint measures the scalability level of each operation exposed by the SUT. To obtain it, we first define the *Greatest Successful Load* (GSL) $\hat{\lambda}_j$ for an operation o_j as the greatest load in Λ for which o_j succeeds. This implies that o_j fails for all $\lambda > \hat{\lambda}_j$. When $\hat{\lambda}_j$ equals the maximum load in Λ, o_j exhibits optimal scalability. The set of GSLs for all operations is referred to as *Scalability Footprint* $\hat{\Lambda}^{\alpha}$ of the SUT with deployment architecture α. In some cases, the footprint represents a strong boundary: when the average response time μ_j increases monotonically with the load, the operation o_j also succeeds for all $\lambda \leq \hat{\lambda}_j$. In this case, the Scalability

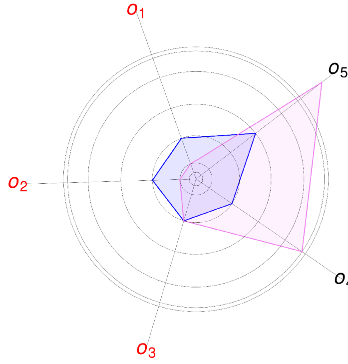


Fig. 11. Visualization of two scalability footprints for the deployment architectures α ■ and β ■ with components A (red operations) and B (black operations).

Footprint represents the boundary for which each operation always succeeds before and always fails after its GSL value.[11]

We use the scalability footprint to compare the alternative deployment architectures in both qualitative and quantitative manners. The qualitative comparison relies on a specific visualization method, that is, a radar plot as illustrated in Figure 11. Each circle in the grid of the radar represents a load $\lambda \in \Lambda$. The distance between two circles is the frequency of the discretized distribution f at the greater load. Each closed polygon in the radar represents the scalability footprint of a deployment architecture. Each vertex of a polygon is the GSL of the corresponding operation. For example, the blue and pink polygons in Figure 11 represent the footprints of α and β over the five operations $\{o_1, \ldots, o_5\}$. Operations exposed by the same architectural component have the same font color (e.g., red and black operations in Figure 11 belong to components A and B, respectively). The vertices either reaching or exceeding the outermost grid circle indicate optimal scalability for the corresponding operation (e.g., o_5).

The quantitative comparison between alternative deployment architectures relies on the Mann-Whitney U-test statistic [34] and the Cliff's delta effect size to classify its magnitude [35]. The Mann-Whitney statistic $U_{\alpha\,\beta}$ measures how many times the GSLs in $\hat{\Lambda}^{\alpha}$ are greater than the GSLs in $\hat{\Lambda}^{\beta}$ for the same operations. According to [34], the Mann-Whitney effect size $u_{\alpha\,\beta}$ is then computed as follows:

$$u_{\alpha\,\beta} = \frac{U_{\alpha\,\beta}}{|\hat{\Lambda}^{\alpha}||\hat{\Lambda}^{\beta}|}, \tag{11}$$

where $|\cdot|$ indicates the set cardinality. To classify the effect size, the Mann-Whitney effect size is first converted to the non-parametric *Cliff delta* effect size d:

$$d = 2 \cdot u_{\alpha\,\beta} - 1. \tag{12}$$

It is then classified according to the standard categorization introduced in [36]:

- negligible (N) effect size, if $|d| < 0.147$;
- small (S) effect size, if $|d| < 0.33$;
- large (L) effect size, otherwise.

[11] If μ_j does not increase monotonically, the scalability footprint only indicates that operations fail for all loads greater than their GS. In this case, the scalability footprint can be still used to compare alternative deployment architectures, but only in terms of their failing behavior after the GLS.

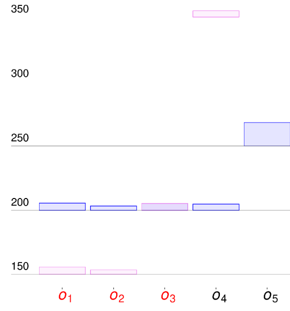


Fig. 12. Scalability gap of the two alternative architectures α and β with components A (red operations) and B (black operations).

The categories measure the strength of the difference between two footprints. For instance, the value $U_{\alpha\,\beta} = 13\%$ with large (L) effect size indicates that architecture α is largely (L) less scalable ($13\% < 50\%$) than β.

This quantitative evaluation can be applied considering all of the operations of the SUT or just those exposed by a specific component depending on the desired granularity level.

Scalability gap and performance offset. With the aim of providing engineers with additional information at the operation level, we define two additional metrics: the scalability gap and the performance offset of failing operations. The former measures the scalability loss and the latter the performance loss due to a failing operation. The *scalability gap SG_j* of a failing operation o_j is the scalability share (Equation (2)) at the minimum $\lambda \in \Lambda$ greater than the GLS of o_j:

$$SG_j^{\alpha}(\lambda) = v_j^{\alpha}(\lambda). \tag{13}$$

The *performance offset PO_j* of a failing operation o_j is the distance from the mean response time to the scalability threshold Γ_j^0 at the minimum $\lambda \in \Lambda$ greater than the GLS of o_j:

$$PO_j^{\alpha}(\lambda) = \frac{\mu_j^{\alpha}(\lambda) - \Gamma_j^0}{\Gamma_j^0}. \tag{14}$$

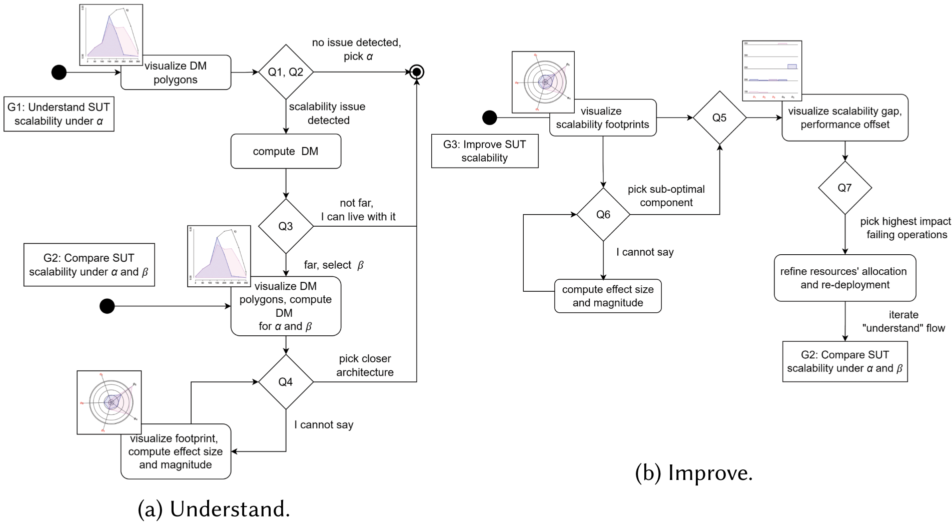
We visualize the scalability gaps and performance offsets of failed operations by means of histograms. Figure 12 shows the scalability gap under two alternative deployment architectures α and β. Horizontal lines represent the loads in Λ. A bar lying on an horizontal lines at l —say, $l = 250$ — represents the scalability gap of an operation (o_5) that has the GSL value equal to the maximum λ smaller than l ($\hat{\lambda}_5 = 200$). Thus, higher bars correspond to failing operations having higher negative impact on SUT scalability. We use the same representation to compare the performance offsets: higher bars correspond to failing operations having higher impact on the performance degradation of the SUT.

4.5 Analysis Workflow

The analysis workflow is tailored to understand (Figure 13(a)) and then improve (Figure 13(b)) the scalability of the system. The analysis starts from the following three goals:

- G1: Understand the scalability of the SUT with a given architecture α.
- G2: Compare the scalability of the SUT with alternative architectures α and β.
- G3: Improve the scalability of the SUT.

The decision process is then guided by questions derived from the goals. Our visualization and measurement framework is then used to answer the questions listed in Figure 13.



(a) Understand.

(b) Improve.

Facet	label	question
Understand	Q1	At which load scalability is an issue?
	Q2	Is a target deployment architecture close to optimal?
	Q3	How far from optimal is a target deployment architecture?
	Q4	Which target deployment architecture is closer to optimal?
Improve	Q5	What are the failing operations at a given load?
	Q6	Which component is far from optimal?
	Q7	What are the failing operations having highest negative impact on performance and scalability?

Fig. 13. Analysis workflow.

Understanding the target deployment architectures. Considering $G1$, the workflow starts by visualizing the $\mathcal{DM}$ polygon of a deployment architecture α to determine whether the architecture satisfies a scalability requirement. Specifically, the architect would like to answer *Q1* and *Q2* for α. If the corresponding $\mathcal{DM}$ polygon is very close to the outer one, the architect may consider α as optimal. If so, the analysis ends with the decision: *no change in the architecture is needed and* α *is recommended.* In case the architect observes scalability issues for some loads (e.g., $\lambda = 150$ and architecture α in Figure 10), the architect can compute the total $\mathcal{DM}$ value and answer *Q3*. If the total $\mathcal{DM}$ is far from optimal, the architect can then consider $G2$ and compare the polygon α and the total $\mathcal{DM}$ with alternative deployment architectures to answer *Q4*. For instance, β and δ are two alternative deployment architectures in Figure 10. Compared to α and β, δ has no scalability issues up to $\lambda = 150$, but the overall $\mathcal{DM}$ is lower. With this information, the architect can understand that δ represents a worse choice.

If the $\mathcal{DM}$ polygons and the total $\mathcal{DM}$ value are not sufficient to identify the deployment architecture closer to optimal, the architect uses the scalability footprints and applies a pairwise comparison of the alternative deployment architectures considering the effect size and magnitude. As an example, α and β in Figure 10 have the same total $\mathcal{DM}$ value but different $\mathcal{DM}$ polygons that may be equally good. In this case, differences may emerge analyzing the scalability footprints. In case multiple architectures still exhibit similar scalability, the architect can refine and improve the deployment configuration (e.g., allocation of resources to components or operations) of one or more architectures and then apply our approach to compare them, as follows.

It is worth noting that both α and β are initial assignments that improved over one or more iterative steps. These assignments are usually based on domain knowledge (coming from the analysis

of historical data [27]) or can be produced by using sampling techniques [37] driving the selection of the initial and subsequent candidate sets of architectures/configurations.

Improving the target deployment architectures. Starting from *G3*, the workflow considers the scalability footprints of one or more deployment architectures to answer *Q5* and *Q6*. The architect compares the effect size and magnitude of the scalability footprints of alternative deployment architectures and selects the components and their operations that require further investigation. The scalability footprints in the radar plot show the failing operations at each load level and the overall differences for components, as for components A and B in Figure 11. For instance, the footprint of β shows worse scalability for component A and better scalability for B. The effect size and magnitude calculated at component level may further indicate that B is largely more scalable with β and A is less scalable with β but by a negligible difference with α. In this case, the architect can choose β and change the deployment configuration for component A to improve the overall scalability of the SUT.

By following the workflow, the architect analyzes each operation by means of the scalability gap and performance offset. The main objective here is to identify failing operations whose impact is higher on performance and scalability. Operations associated with the largest impact yield a severe negative effect that can be mitigated by using suitable architectural choices and resource allocation (within the limits of the physical constraints of the underlying hardware). As an example, by inspecting Figure 12, we can observe that with β, the operations o_1, and o_2 exhibit a scalability gap at load 150, while o_3 yields a scalability gap at load 200. With this information, the architect can allocate more resources to B (operations o_1, o_2, o_3). Considering instead α, the operation o_5 fails at load 250 with a large scalability gap, whereas all other operations fail at load 200 but with a smaller gap. This means that the operation o_5 of B is the most critical one since its invocation frequency is higher compared with the other operations.

5 CASE STUDY 1: TRAFFIC SERVICE

The first case study used for our evaluation is a smart mobility application implemented as a data-intensive microservices system that monitors road networks traffic.

In this section, we briefly present the SUT in Section 5.1 and describe the application of ADD to decompose a bounded context of this application in Section 5.2. In Section 5.3, we discuss the operational setting that has been deployed in our testing environment, presented in Section 5.4, to carry out controlled experiments with multiple deployment architectures, presented in Section 5.5. We describe the execution's workflow of our experiments in Section 5.6 and analyze major results in Section 5.7.

5.1 System Under Test

The system is able to collect data through sensors installed on road networks (or by exploiting a dedicated emulator when data are only available offline for a deferred analysis) and to process those data in the cloud by exploiting technologies for distributed and high-performance processing of large amounts of data. It is an example of a large class of software systems in the context of the edge-to-cloud paradigm that today are very common due to the widespread adoption of Internet of Things (IoT) technologies for the development of smart solutions in city environments. We focus on an extract of the whole system, a specific service used for road network management. We refer the reader to [14, 38] for comprehensive descriptions.

The system handles the API requests by managing the road networks as graph-based data structures, where nodes represent road intersections, relationships (edges) between nodes represent streets, and their weights are the average travel times streamed by other platform components

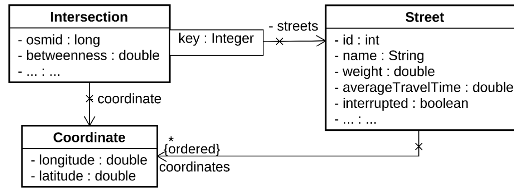


Fig. 14. Data entities within the bounded context of the DomainService.

Table 2. REST Operations Exposed by the TrafficService DDS

DDD	id	description	Action	URI (relative path)	actor	ADD
TrafficService	o_1	addIntersection	POST	`/intersections`	Admin	AdminService
	o_2	addStreet	POST	`/streets`	Admin	
	o_3	getIntersection	GET	`/intersections/{osmid}`	Admin	
	o_4	getStreet	GET	`/streets/{id}`	Admin	
	o_5	getStreets	GET	`/intersections/{id}/streets`	Admin	
	o_6	setStreetInterruption	PUT	`/streets/interruptions/{id}`	Admin	
	o_7	setStreetWeight	PUT	`/streets/{id}`	Admin	
	o_8	getStreetByIntersection	GET	`/streets?i1=x&i2=y`	EndUser	EndUserService
	o_9	getNearestIntersection	GET	`/intersections?lat=x&lon=y`	EndUser	
	o_{10}	getShortestPath	GET	`/paths?i1=x&i2=y&type=s`	EndUser	
	o_{11}	getTopCriticalNodes	GET	`/intersections?BCtop = x`	EndUser	

towards the service layer. Figure 14 shows the class diagram modeling a road network whose elements are the data entities characterizing the specific bounded context. The `Streets` are characterized by a series of coordinates, useful for managing the geometry during the visualization on a map. Every edge is directed. Thus, a two-way street is represented as two edges with opposite orientations. To test our system with real data, the road network of the metropolitan city of Rome was loaded into the database, including 42,753 intersections and 89,251 streets, respectively, graph nodes and edges.

5.2 Actor Driven Decomposition

The application exposes the REST operations listed in Table 2, which are used by two actors.

- `EndUser`: Obtains information via HTTP interactions through dynamic city maps in Web GUIs;
- `Admin`: Manages data with CRUD operations, exposed as RESTful services, that allow streets and intersections to be inserted and changed.

We observed that these two actors operate on the application in different ways, by invoking two different sets of operations as reported in Table 2. This way, from the initial bounded context materialized by TrafficService, we derived two smaller microservices, named *AdminService* and *EndUserService*. In the following, we will refer to services derived from DDD as Domain-Driven Services (DDS) whereas the services derived from ADD are referred as Actor-Driven Services (ADS).

Using the Scale Cube model, in this case study we exploit the ADD strategy to guide a further decomposition of bounded contexts along the y-axis. This enables a differentiated replication of the operations according to the expected load generated by different actors onto the system.

We then decompose the database according to the CRUD decomposition introduced in Section 3.2. The details of this decomposition and the related deployment architecture are discussed in Section 5.5.

Table 3. Behavior Models of the Two Actors Admin and EndUser

Actor	model ID	Operations' sequence
Admin	bm_1	$\{o_3 \xrightarrow{t_a} o_5 \xrightarrow{t_u} o_3 \xrightarrow{t_a} o_5 \xrightarrow{t_u} o_2\}$
	bm_2	$\{o_1 \xrightarrow{t_u} o_1 \xrightarrow{t_u} o_1 \xrightarrow{t_u} o_2 \xrightarrow{t_u} o_2\}$
	bm_3	$\{o_3 \xrightarrow{t_a} o_5 \xrightarrow{t_u} o_4 \xrightarrow{t_u} o_7 \xrightarrow{t_u} o_6 \xrightarrow{t_u} o_4\}$
EndUser	bm_4	$\{o_9 \xrightarrow{t_u} o_9 \xrightarrow{t_a} o_{10} \xrightarrow{t_u} o_9\}$
	bm_5	$\{o_{11} \xrightarrow{t_u} o_9 \xrightarrow{t_a} o_8 \xrightarrow{t_u} o_8\}$
	bm_6	$\{o_8 \xrightarrow{t_u} o_9 \xrightarrow{t_u} o_9\}$

Table 4. Two Behavior Mixes Specified by the Usage Profiles Ω and Ω'

	Ω		Ω'	
model ID	**behavior mix**	**#operations**	**behavior mix**	**#operations**
bm_1	0.166	$\sim 2.5k$	0.083	$\sim 1.25k$
bm_2	0.166	$\sim 2.5k$	0.083	$\sim 1.25k$
bm_3	0.166	$\sim 2.5k$	0.083	$\sim 1.25k$
bm_4	0.166	$\sim 2.5k$	0.250	$\sim 3.75k$
bm_5	0.166	$\sim 2.5k$	0.250	$\sim 3.75k$
bm_6	0.166	$\sim 2.5k$	0.250	$\sim 3.75k$

5.3 Operational Setting

5.3.1 Workload Specification and Behavior Models. Starting from the operations presented in Table 2, we defined six behavior models operating on two corresponding sets of REST operations. Each set corresponds to one actor (equivalently, component) of the system, according to our proposed decomposition strategy. Each behavior model is defined by sequences of operations and a *thinking time* for each service invocation sampled from two different uniform distributions: t_a between 1 and 200 msec to simulate automated elaboration of results, and t_u between 1,000 and 5,000 msec to simulate human interaction. Table 3 lists the six behavior models (with IDs $\{bm_1, \ldots, bm_6\}$) that we used to define the operational profile of the SUT.

5.3.2 Usage Profiles. For each testing session, we considered a behavior mix as reported in Table 4. The two behavior mixes in the table represent two usage profiles derived from two alternative assumptions: Ω, that is, a balanced distribution of the users per component (i.e., 50% EndUsers and 50% Admins); and Ω', that is, an unbalanced distribution of the users per component (i.e., 75% EndUsers and 25% Admins).

5.3.3 Operational Profile. With the information reported earlier in mind, throughout the experiments we assumed an arbitrary target operational profile that represents the expected workload in production. For each testing session, we set the discretized distribution of the load as follows:

$$\begin{aligned} \Lambda &= \{2, 50, 100, 150, 200, 250, 300, 350\} \\ f(\Lambda) &= \{0.008, 0.045, 0.049, 0.207, 0.239, 0.247, 0.155, 0.031\} \end{aligned} \quad (15)$$

5.4 Testing Environment

5.4.1 Hardware Constraints. To deploy and test the SUT, we used two blade servers[12] managed by VMWare vSphere and ESXi hypervisor. Each blade is equipped with 64 GB of RAM and

[12]The blade servers have been kindly provided by TIM S.p.A. in the framework of a research collaboration.

two physical processors INTEL XEON E5-2667 v3 at 3.20GHz. Each processor is composed of eight physical cores with hyper-treading (32 virtual cores per blade). The blades are connected to a 2-TB capacity datastore.

5.4.2 Virtualization Layers. The hardware infrastructure is virtualized by using OPENSTACK.[13] We created six virtual machines (VMs) with six virtual cores (vCPUs) each and a maximum frequency of 3.20GHz per virtual core. The six VMs provided by OPENSTACK are used to deploy OPENSHIFT on top of KUBERNETES.[14] Two of these machines are used for infrastructure management, collecting metrics, managing images, routing, and load balancing. The remaining four nodes (`compute1`, . . . , `compute4`), have 5.6 cores used to run the SUT. In this environment, we used KUBERNETES Pods as atomic deployment units.

5.4.3 Platforms. On the virtualized hardware, we deployed the graph-based DBMS NEO4J[15] in causal clustering with three core servers to ensure fault-tolerance and a variable number of read replica servers. Each core server is deployed into a separate KUBERNETES Pod of a *StatefulSet* that guarantees data persistence. The application business logic is encapsulated and deployed onto the WILDFLY[16] application servers that expose RESTful endpoints. The SUT running on the aforementioned virtualization layers have been tested by using JMETER.[17]

5.5 Deployment Architectures

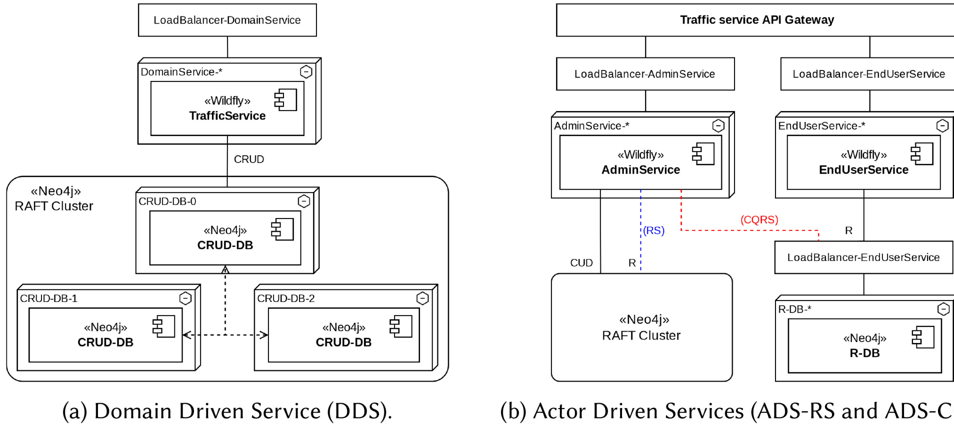
Taking available resources into consideration, we evaluated different deployment architectures with our methodology. In particular, we adopted the same deployments of application servers on software pods ($deployment_{sp}$) and we changed the deployments of pods to virtual machines ($deployment_{pm}$), as follows.

5.5.1 $Deployment_{sp}$. Figure 15 illustrates the $deployment_{sp}$ of the original DDS and two alternative decompositions of the ADS.

Domain-Driven Service. According to the schema in Figure 15(a), the microservice is encapsulated and deployed within a WILDFLY pod that can be replicated for scaling purposes, as noted by symbol $*$ in the name. This tier is directly connected to the database tier formed by a cluster of pods hosting NEO4J DBMS instances. The Neo4j RAFT CLUSTER manages the CRUD operations requested from the upper tier. To face increasing loads, client requests can be mediated by a load balancer that ensures horizontal scaling by replicating the microservice.

Actor-Driven Services. As shown in Figure 15(b), ADD has been applied by splitting the `TrafficService` component of the DDS into two different pods, `AdminService` and `EndUserService`, corresponding to the two actors interacting with it. According to Table 2, the operations of Admin ($o_1, \dots o_7$) have been moved to the first pod, whereas EndUser operations ($o_8, \dots, o_{11}$) have been moved into the latter one. Departing from the former architecture, we considered also read-only replica servers of a NEO4J cluster. These replicas mirror the values stored by core servers by periodically executing update procedures to ensure eventual consistency. We derived two alternative $deployment_{sp}$ ADS-RS and ADS-CQRS in which core and replica servers were used in two different configurations, according to the patterns RS and CQRS, introduced in Section 3.2. Figure 15(b) shows the deployment of the two alternative decompositions. The schema

[13]https://www.openstack.org/.
[14]https://kubernetes.io/.
[15]https://neo4j.com/.
[16]https://www.wildfly.org/.
[17]https://jmeter.apache.org/.



(a) Domain Driven Service (DDS). (b) Actor Driven Services (ADS-RS and ADS-CQRS).

Fig. 15. Application $deployment_{sp}$ with Domain-Driven and Actor-Driven Decomposition architectures.

illustrates common interactions and components (black), the interactions that hold only for ADS-RS (blue), and the ones that hold only for ADS-CQRS (red). In both cases, an API Gateway is used to route the client requests towards the right microservice.

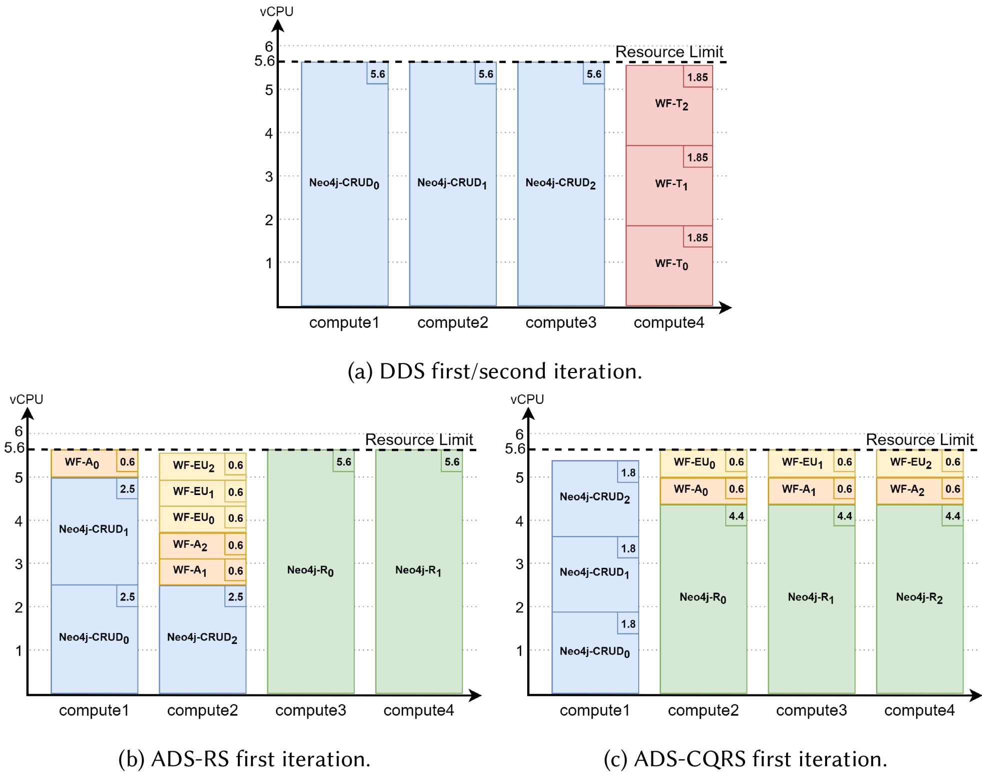
Considering ADS-RS, `EndUserService` can leverage horizontal scaling of read queries provided by the `R-DB` servers. This option is more convenient than scaling up `CRUD-DB` servers since they do not need additional work for leader election and consistency management. There is a complete separation of `AdminService` and `EndUserService` through the presence of two different execution environments. The deployment of `AdminService` is quite similar to the `TrafficService` architecture, whereas `EndUserService` requires an additional `Kubernetes Load Balancer` placed between the Wildfly and Neo4j pods, since read replicas scaling is not directly managed by Neo4j.

Considering ADS-CQRS, Read (R) and Write (CUD) operations are assigned to core servers and read replicas, respectively, as shown in Figure 15(b) (red arrow).

5.5.2 $Deployment_{pm}$. For each alternative $deployment_{sp}$ architecture (DDS, ADS-RS, and ADS-CQRS), we considered different deployments of containers to virtual machines in each of the two aforementioned iterations. Essentially, we reproduced alternative conditions in which all of the available resources are allocated to the set of microservices, as discussed in Section 3.1. It is worth noting that this is usually achieved by means of autoscaling components in charge of dynamically assigning the available resources to the services. To replicate the same experimental settings over multiple runs, in our experiments we systematically applied the same resource mapping according to Equation (7). The resulting resource mapping of the first iteration is shown in Figure 16, which illustrates the deployment of the Kubernetes pods hosting the components described in Figure 15. The labels on the x-axis indicate the computational nodes ($compute_1, \ldots, compute_4$) used for the deployment in our virtualized testing environment.

The DDS deployment (Figure 16(a)) assigns a dedicated machine to each `Neo4j-CRUD` instance, due to the high demand of DBMS resources of the exposed operations, while the remaining resources ($compute_4$) were assigned to the Business Logic (BL) split into three Wildfly pods ($\texttt{WF-T}_0$, $\texttt{WF-T}_1$, $\texttt{WF-T}_2$). The splitting between the resources assigned to BL and DBMS was empirically computed by analyzing the actual consumption of resources under the maximum throughput sustainable by the system.

The deployment of the two actor-driven architectures splits the functions of the DBMS component by exploiting the optimized `Neo4j-R` instances. Resource allocation was carried out by



(a) DDS first/second iteration.

(b) ADS-RS first iteration.

(c) ADS-CQRS first iteration.

Fig. 16. Initial $deployment_{pm}$ and resource allocation.

considering the higher computational complexity of the operations performed by `EndUsers` (e.g., calculation of shortest paths) compared with `Admins`. We reserved either two or three compute nodes for the `Neo4j-Rs`. Then, we equally split the three BL pods between the `AdminService` and `EndUserService` and obtained 6 Kubernetes pods ($\texttt{WF-A}_0$, $\texttt{WF-A}_1$, $\texttt{WF-A}_2$, $\texttt{WF-EU}_0$, $\texttt{WF-EU}_1$, $\texttt{WF-EU}_2$) as replicated BL of the two services. Finally, we allocated the vCPUs to optimize the available resources taking into account the technology constraints (e.g., the Wildfly application server for the BL pod needs a minimum number of vCPUs greater than 0.5). As a result, we obtained the two deployment configurations in Figure 16(b) and Figure 16(c) that have been used as the initial configuration for assessing the scalability level of the two architectures.

5.6 Experiment Execution

To answer our research questions (RQ1–RQ4), we designed and conducted a set of controlled experiments in two iterations. In each iteration, the three alternative software architectures (DDS, ADS-RS, ADS-CQRS) have been deployed in our testing environment. For each iteration and deployment architecture, we executed our multi-level scalability assessment (Section 4) with the two behavior mixes Ω and Ω' defined in Table 4 and increasing loads as in Λ (Equation (15)). In each testing session, we sampled a large amount of data to avoid the risk of obtaining results by chance. We collected $\sim 15k$ operation calls, excluding possible spurious data during the rump-up/ramp-down phases as recommended in [27]. Then, each session was repeated three times and the absence of statistical difference among consecutive sessions was assessed by conducting a pairwise comparison through the Mann-Whitney U Test. A total amount of $\sim 45k$ operation calls was sampled per

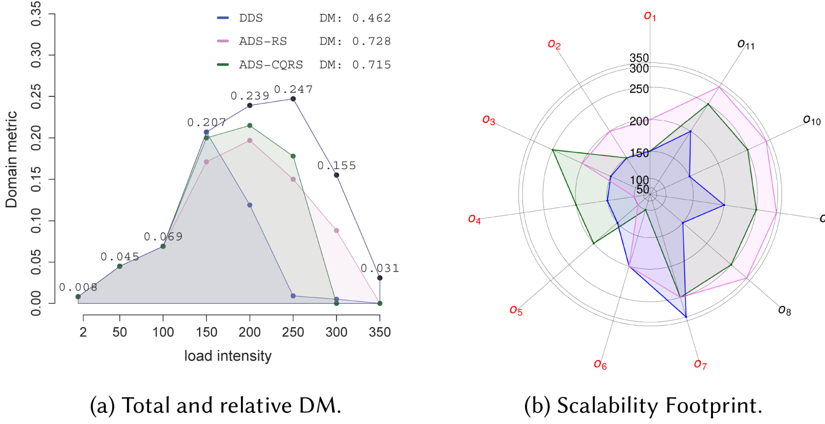
Table 5. $\mathcal{DM}$ Over Iterations and Usage Profiles

	First Iteration		Second Iteration	
	Ω	Ω'	Ω	Ω'
DDS	0.462	0.461	0.462	0.461
ADS-RS	**0.728**	**0.768**	0.688	0.779
ADS-CQRS	0.715	**0.768**	**0.827**	**0.808**

Table 6. Mann-Whitney Analysis of System-Level Footprints in the First Iteration

Iteration	Profile	$U_{\text{DDS RS}}$	$U_{\text{DDS CQRS}}$	$U_{\text{CQRS RS}}$
First	Ω	(29%, M)	(31%, M)	(57%, N)
	Ω'	(5%*, L)	(5%*, L)	(50%, N)
Second	Ω	(45%, N)	(13%*, L)	(30%, M)
	Ω'	(4%*, L)	(1%*, L)	(62%, S)

*Statistical significance (p-value < 0.05).



(a) Total and relative DM.

(b) Scalability Footprint.

Fig. 17. First iteration under profile Ω. Architectures: DDS ■, ADS-RS ■, and ADS-CQRS ■. Components: `Admin` (red) and `EndUser` (black).

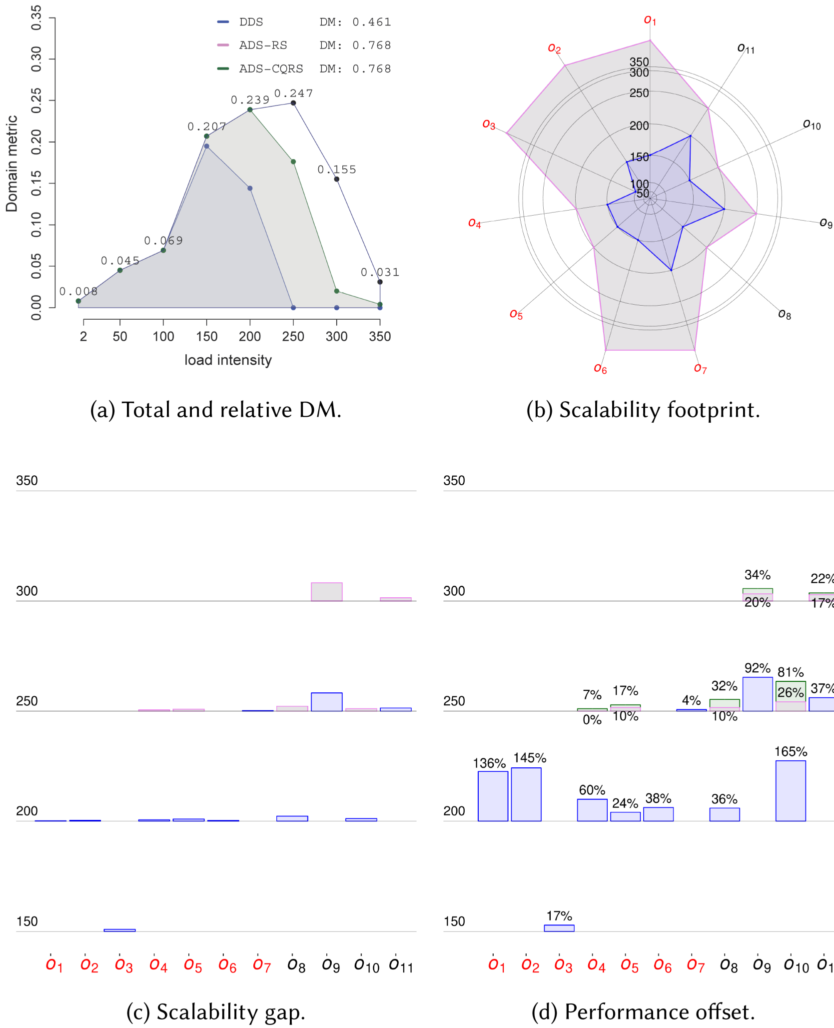
each individual deployed architecture in each iteration. Overall, a total of 240 load testing sessions was executed throughout the two iterations.

In each iteration, we used the DDS as baseline deployment architecture α_0 executed under the baseline load $\lambda_0 = 2$ from which a scalability requirement (1) was extracted. After executing the measurement workflow, we applied the analysis workflow to extract insights and possibly guide future iterations. Results from the first iteration were interpreted to plan and apply architectural changes that were assessed in the second iteration.

5.7 Experiment Results

5.7.1 Results of the First Iteration. We evaluated the three architectures by means of measurement workflow under the deployment presented in Figure 16 and the two usage profiles Ω and Ω' in Table 4. Then, we applied the analysis workflow to extract insights and drive the decisions of the software architect, as reported in the following.

Usage profile Ω. The results in Table 5 highlight better scalability with the ADS-RS deployment architecture over DDS and ADS-CQRS. The DDS yields the lowest $\mathcal{DM}$ score equal to 0.462. ADS-RS improves the scores up to 0.728, whereas ADS-CQRS exhibits a slightly lower score equal to 0.715. The results of the scalability footprint in Table 6 confirm better scalability with the ADS-RS architecture over the Domain Driven architecture and ADS-CQRS. The comparison (ADS-CQRS, ADS-RS) returns 57% with a negligible effect size. To understand this outcome at the system and component levels, we further inspected the visualizations reported in Figure 17. As shown in Figure 17(a), DDS exhibits optimal performance up to $\lambda = 150$ users and shows thereafter a rapid decrease toward poor performance. Both ADS-RS and ADS-CQRS architectures are optimal up



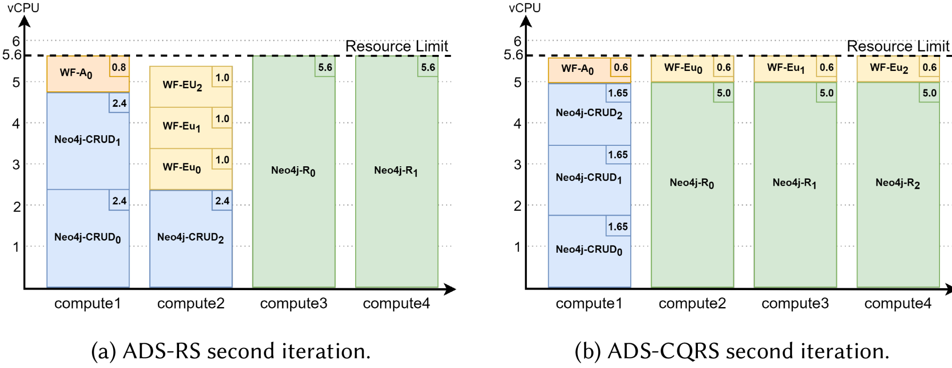
(a) Total and relative DM. (b) Scalability footprint.

(c) Scalability gap. (d) Performance offset.

Fig. 18. First iteration under profile Ω'. Architectures: DDS ■, ADS-RS ■, and ADS-CQRS ■. Components: `Admin` (red) and `EndUser` (black).

to $\lambda = 100$ and the ADS-RS architecture outperforms ADS-CQRS for the highest loads. The scalability footprint at component-level in Figure 17(b) explains the difference between ADS-RS and ADS-CQRS. The radar plot shows that with ADS-CQRS, the operations of the `EndUser` component (i.e., $o_8, \ldots, o_{11}$) fail at lower load compared with ADS-RS (i.e., 250 vs. 300), whereas the operations exposed by `Admin` have no uniform scalability behavior for all of the architectures.

Usage profile Ω'. Table 5 shows that ADS-RS and ADS-CQRS have the same $\mathcal{DM}$ score equal to 0.768. The results in Table 6 confirm an inconclusive pairwise comparison between ADS-CQRS and ADS-RS. Both ADS-CQRS and ADS-RS are significantly largely closer to optimal compared with DDS. By inspecting the visualization reported in Figure 18(a), we can observe that both ADS-RS and ADS-CQRS achieve optimal scalability up to $\lambda = 200$ and have better scalability than DDS for all loads. The scalability footprint in Figure 18(b) shows that both ADS-RS and ADS-CQRS achieve optimal scalability for five out of seven operations exposed by the `Admin` component, whereas



(a) ADS-RS second iteration.

(b) ADS-CQRS second iteration.

Fig. 19. Refined $deployment_{pm}$ and resource allocation.

the `EndUser` component is in general less scalable as the GSL is at most 250. This outcome is consistent with profile Ω' since 75% of the load target operations impacted the `EndUser` component. This suggests that all of the deployment architectures are not able to properly support this load condition.

We deepen the investigation by analyzing the operations in terms of scalability gap and performance offset reported in Figure 18(c) and Figure 18(d), respectively. The scalability gap is higher for all `EndUser` component operations across all deployment architectures. We detected that the negative impact associated with failing operation o_9 is the highest impact. Figure 18(d) confirms its relevance in terms of performance offset. With DDS, o_9 fails at load 250 and the average response time is 92% higher with respect to its scalability requirement. With ADS-RS and ADS-CQRS, o_9 fails at load 300 and exhibits the performance offset values 20% and 34%, respectively. According to Figure 18(d), we can grasp fine-grained differences between ADS-RS and ADS-CQRS. For instance, operation o_{10} has the same scalability gap at load 250 but different performance offset values.

To sum up, during the first iteration, we found that `EndUserService` for both ADS-RS and ADS-CQRS architectures was the most critical when the system is overloaded. We leveraged this observation to guide a second iteration in which we reallocated the available resources from the BL of `AdminService` to that of the `EndUserService` in order to increase the scale of operation of this latter component (i.e., the most critical one according to Figure 18(c)).

5.7.2 Results of the Second Iteration. Figure 19 shows the refined $deployment_{pm}$ used for ADS-RS and ADS-CQRS in the second iteration. We deployed four BL pods: one for `AdminService` and three for `EndUserService`.

Due to the resource constraints of the underlying infrastructure and those imposed by the architectural choices for RS (Figure 19(a)), we were not able to allocate additional resources to the `Neo4j-R` replicas. Hence, we decided to reduce the `AdminService` BL to a single pod and reduce to 2.4 vCPUs the resources assigned to each `Neo4j-CRUD` instance in order to save about 2 vCPUs and assign them to the `EndUserService` BL pods. ADS-CQRS offered instead more flexibility (see Figure 19(b)). In this case, we reduced the amount of resources for `Neo4j-CRUDs` (from 1.8 vCPUs to 1.65 vCPUs) and the number of BL pods of `AdminService` in order to increase the amount of resources assigned to `Neo4j-Rs` from 13.2 vCPUs to 15 vCPUs.

Profile Ω. Both ADS-RS and ADS-CQRS improve the $\mathcal{DM}$ score up to 0.768, as shown in Table 5.

Table 6 shows significantly large scalability improvement for both ADS-CQRS and ADS-RS over the core architecture. This result is confirmed by the relative DM in Figure 20(a). As shown by Figure 20(b), ADS-CQRS also yields homogeneous scalability behavior within each component: all

Table 7. Mann-Whitney Analysis of Component-Level Footprints in the Second Iteration

Component	Profile	$U_{\mathtt{DDS\ RS}}$	$U_{\mathtt{DDS\ CQRS}}$	$U_{\mathtt{CQRS\ RS}}$
`Admin`	Ω	(67%, M)	(19%*, L)	(88%*, L)
`EndUser`	Ω	(0%*, L)	(0%*, L)	(75%, L)
`Admin`	Ω′	(0%*, L)	(1%*, L)	(17%*, L)
`EndUser`	Ω′	(19%, L)	(0%*, L)	(88%, L)

*Statistical significance (p-value < 0.05).

but one operation exposed by the `Admin` component fail at load 200, whereas all of the `EndUser` component operations do not fail under the highest load, 350. Table 7 shows that the `Admin` component with ADS-CQRS is significantly closer to optimal compared with ADS-RS (i.e., Mann-Whitney U-test, 88%). It is worth noting that for this component, RS is even worse than the core (i.e., Mann-Whitney U-test, 67%) with a medium effect size. For the `EndUser` component, both ADS-RS and ADS-CQRS are significantly closer to optimal compared with DDS. Finally, data exhibit significant support of ADS-CQRS over ADS-RS for both `Admin` and `EndUser`.

As can be seen in Figure 20(d), the analysis at operation level detects a very large performance offset for the operations exposed by the `Admin` component with ADS-CQRS (e.g., the offset of o_3 is 267%). Thus, improving this operation is challenging. A similar observation holds for the `Admin` component operations with RS. Considering the `EndUser` operations, we can observe that o_8 has the same scalability gap (highest gap at load 350) in ADS-CQRS and ADS-RS, whereas the offset is different: 6% in ADS-CQRS and 31% in ADS-RS. Hence, further improvement of ADS-CQRS is easier in this case. It is worth noting that o_9 with Monolith has a very high scalability gap. Since the corresponding offset is relatively low (i.e., 13%) we could reserve more resources to `EndUser` in order to improve o_9, which has a severe negative impact. Nonetheless, fine-grained reallocation of resources between the two components is not feasible in this case since, as described earlier, the Monolith deploys the two components within the same execution unit.

Profile Ω′. Table 5 identifies ADS-CQRS as superior to DDS and ADS-RS. Compared with Ω, the score slightly decreases for ADS-CQRS (0.808), whereas it increases for ADS-RS (0.779). This effect is due to the larger amount of resources for `Admin` with ADS-RS that produce the difference in performance of the two architectures at 250 and 300, as illustrated in Figure 21(a). Figure 21(b) shows that both RS and CQRS are closer to optimal compared with DDS. However, the ADS-CQRS architecture achieves a homogeneous scalability footprint, where the GSL for all operations but two is 250. RS yields instead nonhomogeneous results. The GSL of all of the `Admin` operations is higher than `EndUser` operations. Also, Table 7 shows a significant dominance of ADS-RS over ADS-CQRS (i.e., Mann-Whitney U-test 17%) for `Admin`, while for `EndUser`, it shows a significant dominance of ADS-CQRS over both DDS and ADS-RS.

According to Ω′, the scalability gap of the `EndUser` operations is higher with respect to `Admin` operations. As shown in Figure 21(c), failures of o_9 to o_{11} yield in general a more severe negative impact compared with failures of o_1 to o_8. We detected that the negative impact associated with the failing operation o_9 is the highest one. This operation fails at load 300 for both ADS-CQRS and ADS-RS. Nevertheless, the results in Figure 18(d) show diverse performance offsets: 5% and 48%, respectively. The smaller performance offset of ADS-CQRS implies higher potential improvement through allocation of additional resources to the `EndUser` component. These fine-grained insights could be used in principle to drive another iteration of our methodology. In our case, the physical constraints of our testing environment leave limited room for further improvement since all of the available resources have been saturated.

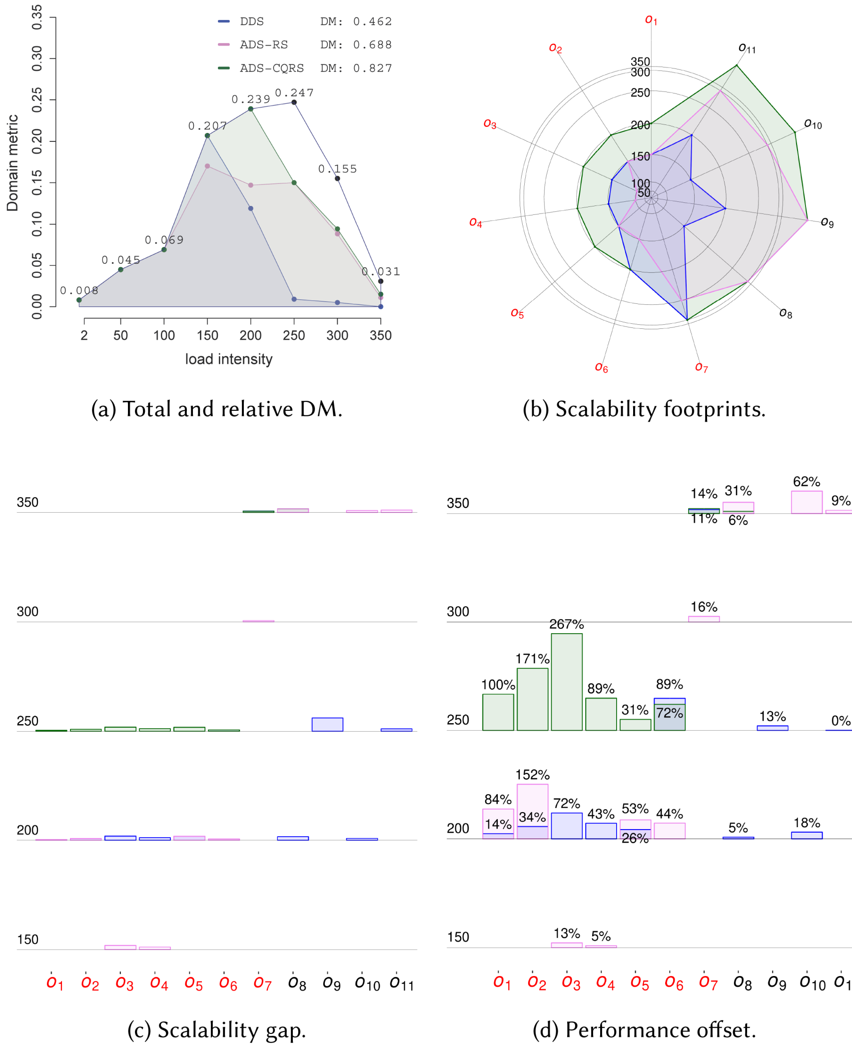
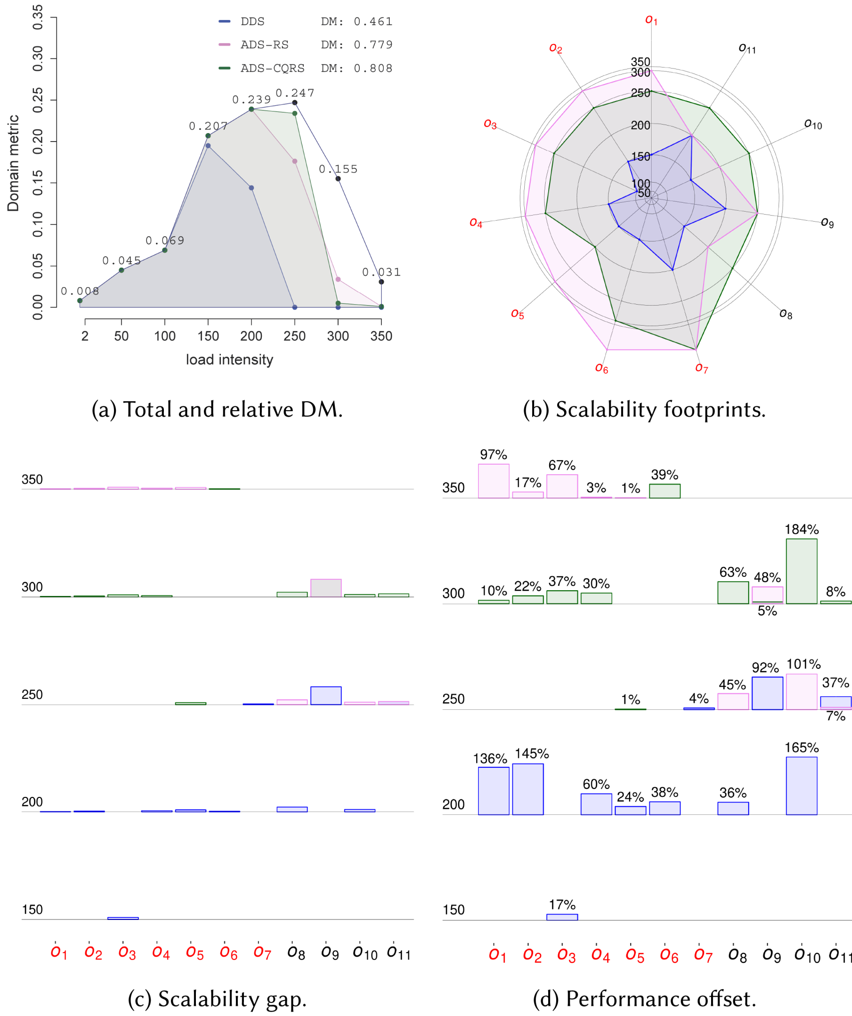


Fig. 20. Second iteration under profile Ω. Architectures: DDS ■, ADS-RS ■, and ADS-CQRS ■. Components: `Admin` (red) and `EndUser` (black).

6 CASE STUDY 2: TRAINTICKET SYSTEM

In this section, we discuss another case study in which we applied our methodology to the *TrainTicket* system [15]. This is a popular open-source microservices system, widely used in the scientific community as a benchmark. We use this second case study to further generalize our results by enlarging the scope of our experiments.

In Section 6.1, we briefly present the SUT. In Section 6.2, we describe again the application of ADD to refine the existing services. In Section 6.3, we discuss the operational setting that has been reproduced in our testing environment, which we present in Section 6.4, to carry out controlled experiments with multiple deployment architectures as discussed in Section 6.5. In Section 6.6, we describe the workflow of our experiments and analyze major results in Section 6.7.



(a) Total and relative DM.

(b) Scalability footprints.

(c) Scalability gap.

(d) Performance offset.

Fig. 21. Second iteration under profile Ω′. Architectures: DDS ■, ADS-RS ■, and ADS-CQRS ■. Components: `Admin` (red) and `EndUser` (black).

6.1 System Under Test

TrainTicket is an e-commerce application designed to manage train ticket booking for the Chinese train system. It exposes an end-point for Web browsers and a REST API gateway that conveys external requests to 41 internal services and 22 databases.

The application can be used by a number of actors that have access to the operations exposed by the API. As shown in Figure 22, the actors can be divided in three groups: (1) *logged* users who need authentication; (2) *external* users who do not need any authentication; and (3) *admin* users who have more privileges and can directly access to CRUD operations for management reasons. Both logged and external users are specialized into *hs* (high speed) and *other* depending on the kind of trains they search or buy (high speed or other, respectively).

In this case study, we focus on the application of our methodology to three selected bounded contexts that collectively compose the main business logic of TrainTicket:

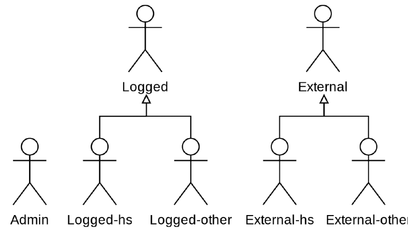


Fig. 22. TrainTicket Actors.

Table 8. Train Ticket Auxiliary APIs

Service	id	description	Action	URI (relative path)	actor
Dashboard	d_1	Home	GET	/index.html	External, Logged,
	d_2	Admin Home	GET	/admin.html	Admin
User	u_1	Perform Login	POST	/api/v1/users/login	Logged, Admin
Route	r_1	Get all routes	GET	/api/v1/adminrouteservice/adminroute	Admin
Assurance	a_1	Get assurance type	GET	/api/v1/assuranceservice/assurances/types	Logged
Food	f_1	Get travel food	GET	/api/v1/foodservice/foods/{date}/{startStation}/{endStation}/{tripId}	Logged
Contact	c_1	Get user's contacts	GET	/api/v1/contactservice/contacts/account/{accountId}	Logged
Payment	y_1	Pay an order	POST	/api/v1/inside_pay_service/inside_payment	Logged

Table 9. Main REST Operations Exposed by the Train Ticket Gateway

DDD	id	description	Action	URI (relative path)	actor	ADD
Travel	t_1	Search hs travel	POST	/api/v1/travelservice/trips/left	Logged-hs External-hs	Travel-hs
	t_2	Search other travel	POST	/api/v1/travel2service/trips/left	Logged-other External-other	Travel-other
	t_3	Get all Travels	GET	/api/v1/admintravelservice/admintravel	Admin	Travel-admin
	t_4	Create Travel	POST	/api/v1/admintravelservice/admintravel		
Preserve	p_1	Preserve hs train	POST	/api/v1/preserveservice/preserve	Logged-hs	Preserve-hs
	p_2	Preserve other train	POST	/api/v1/preserveotherservice/preserveOther	Logged-other	Preserve-other
Order	o_1	Refresh hs order	POST	/api/v1/orderservice/order/refresh	Logged-hs	Order-hs
	o_2	Refresh other order	POST	/api/v1/orderOtherService/order/refresh	Logged-other	Order-other
	o_3	Get all Orders	GET	/api/v1/adminorderservice/adminorder	Admin	Order-admin
	o_4	Update Order	PUT	/api/v1/adminorderservice/adminorder		

- *Travel*: Allows the trains to be searched by using start and destination stations as input as well as management operations to be invoked by administrators.
- *Preserve*: Allows tickets to booked by logged users specifying a train and other options, such as assurance and food.
- *Order*: Allows reserved tickets to be managed by logged users and CRUD operations to be executed by administrators.

Since we do not focus on the other (auxiliary) microservices, we considered their exposed operations (see Table 8) as a whole group hosted by a large execution unit running on a large set of resources (see Figure 25).

6.2 Actor-Driven Decomposition

Taking into account the identified actors, we applied ADD to further decompose the three main bounded contexts into smaller microservices, as reported in Table 9. Figure 23 shows a schema of the ADD outputs, where the three initial bounded contexts are decomposed into smaller actor-driven microservices, following two dimensions of the Scale Cube model: functions (y-axis)

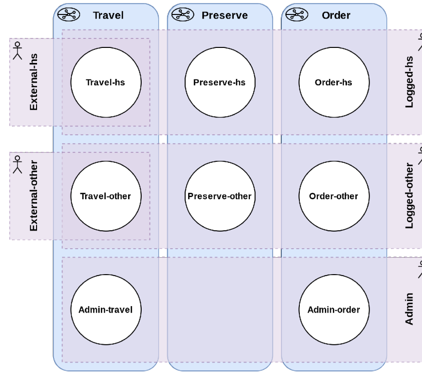


Fig. 23. DDD and ADD decomposition of TrainTicket.

Table 10. Behavior Models of `Admin`, `Logged-hs`, `Logged-other`, `External-hs`, and `External-other`

Actor	model ID	Operations' sequence
`Admin`	bm_{t_1}	$\{d_2 \xrightarrow{t_u} u_1 \xrightarrow{t_a} d_2 \xrightarrow{t_a} t_3 \xrightarrow{t_u} r_1 \xrightarrow{t_u} t_4\}$
	bm_{t_2}	$\{d_2 \xrightarrow{t_u} u_1 \xrightarrow{t_a} d_2 \xrightarrow{t_a} o_3 \xrightarrow{t_u} o_4\}$
`Logged-hs`	bm_{t_3}	$\{d_1 \xrightarrow{t_u} u_1 \xrightarrow{t_a} d_1 \xrightarrow{t_u} t_1 \xrightarrow{t_u} a_1 \xrightarrow{t_a} f_1 \xrightarrow{t_a} c_1 \xrightarrow{t_u} p_1 \xrightarrow{t_u} o_1 \xrightarrow{t_u} y_1 \xrightarrow{t_u} t_1 \xrightarrow{t_u} a_1 \xrightarrow{t_a} f_1 \xrightarrow{t_a} c_1 \xrightarrow{t_u} p_1 \xrightarrow{t_u} o_1 \xrightarrow{t_u} y_1\}$
	bm_{t_4}	$\{d_1 \xrightarrow{t_u} u_1 \xrightarrow{t_a} d_1 \xrightarrow{t_u} t_1 \xrightarrow{t_u} a_1 \xrightarrow{t_a} f_1 \xrightarrow{t_a} c_1 \xrightarrow{t_u} p_1 \xrightarrow{t_u} o_1 \xrightarrow{t_u} y_1\}$
`Logged-other`	bm_{t_5}	$\{d_1 \xrightarrow{t_u} u_1 \xrightarrow{t_a} d_1 \xrightarrow{t_u} t_2 \xrightarrow{t_u} a_1 \xrightarrow{t_a} f_1 \xrightarrow{t_a} c_1 \xrightarrow{t_u} p_2 \xrightarrow{t_u} o_2 \xrightarrow{t_u} y_1 \xrightarrow{t_u} t_2 \xrightarrow{t_u} a_1 \xrightarrow{t_a} f_1 \xrightarrow{t_a} c_1 \xrightarrow{t_u} p_2 \xrightarrow{t_u} o_2 \xrightarrow{t_u} y_1\}$
	bm_{t_6}	$\{d_1 \xrightarrow{t_u} u_1 \xrightarrow{t_a} d_1 \xrightarrow{t_u} t_2 \xrightarrow{t_u} a_1 \xrightarrow{t_a} f_1 \xrightarrow{t_a} c_1 \xrightarrow{t_u} p_2 \xrightarrow{t_u} o_2 \xrightarrow{t_u} y_1\}$
`External-hs`	bm_{t_7}	$\{d_1 \xrightarrow{t_u} t_1 \xrightarrow{t_u} t_1\}$
	bm_{t_8}	$\{d_1 \xrightarrow{t_u} t_1\}$
`External-other`	bm_{t_9}	$\{d_1 \xrightarrow{t_u} t_2 \xrightarrow{t_u} t_2\}$
	$bm_{t_{10}}$	$\{d_1 \xrightarrow{t_u} t_2\}$

and data (z-axis). For instance, `Preserve-hs` serves only `Logged` users who are interested in high-speed trains. `Travel-hs` is another example of granular service that handles a subset of the whole travel dataset but serves both `External` and `Logged` users who can both search for high-speed trains.

As illustrated in Figure 23, ADD identifies finer building blocks implemented as microservices and deployed onto dedicated execution units. This enables a differentiated replication schema of the operations according to the expected load generated by different actors of the system.

To separate the inner logical levels of each microservice as well, we also decomposed the databases using the data-partitioning approach introduced in Section 3.2. The details of this decomposition and the corresponding deployment architecture are discussed in Section 6.5.

6.3 Operational Setting

6.3.1 Workload Specification and Behavior Models. By taking into account the pre- and post-conditions of the operations in Table 9 and Table 8, we defined 10 behavior models operating on 5 sets of REST operations, each one for each actor. Each behavior model is defined by sequences of operations and a *thinking time* for each service invocation sampled from two different uniform distributions: $\mathtt{t_a}$ between 1 and 200 msec to simulate automated elaboration of results, and $\mathtt{t_u}$ between 1,000 and 5,000 msec to simulate human interaction. Table 10 lists the models $\{bm_{t_1}, \ldots, bm_{t_{10}}\}$ we used to define the operational profile of the SUT.

Table 11. Behavior Mixes Specified by Usage Profiles

Actors Distribution		model Id	behavior mix	#operations
	Admin 5%	bm_{t_1}	0.025	$\sim 0.5k$
		bm_{t_2}	0.025	$\sim 0.5k$
Logged 35%	Logged-hs 7%	bm_{t_3}	0.035	$\sim 0.7k$
		bm_{t_4}	0.035	$\sim 0.7k$
	Logged-other 28%	bm_{t_5}	0.140	$\sim 2.8k$
		bm_{t_6}	0.140	$\sim 2.8k$
External 60%	External-hs 12%	bm_{t_7}	0.060	$\sim 1.2k$
		bm_{t_8}	0.060	$\sim 1.2k$
	External-other 48%	bm_{t_9}	0.240	$\sim 4.8k$
		$bm_{t_{10}}$	0.240	$\sim 4.8k$

6.3.2 Usage Profile. We derived the usage profile from the documentation of TrainTicket[18] and by considering our prior knowledge of the usage of similar e-commerce applications. We assumed the following realistic distribution of actors: 5% working as `Admin`, 35% behaving as `Logged` users and 60% behaving as `External` users. We further characterized `Logged` and `External` according to a common distribution of travelers into the *high speed* and *other* categories. We assumed that 20% of the whole set of users (either Logged or External) are `hs`, while 80% are `other`. The distribution is reported in Table 11.

6.3.3 Operational Profile. We assumed a target workload for the system in production and, for each testing session, we discretized the load as follows:

$$\begin{aligned} \Lambda &= \{5, 10, 20, 40, 50, 60, 80, 100, 150, 200, 250, 300, 350\} \\ f(\Lambda) &= \{0.008, 0.010, 0.015, 0.023, 0.037, 0.045, 0.059, 0.124, 0.211, 0.198, 0.152, 0.107, 0.011\} \end{aligned} \tag{16}$$

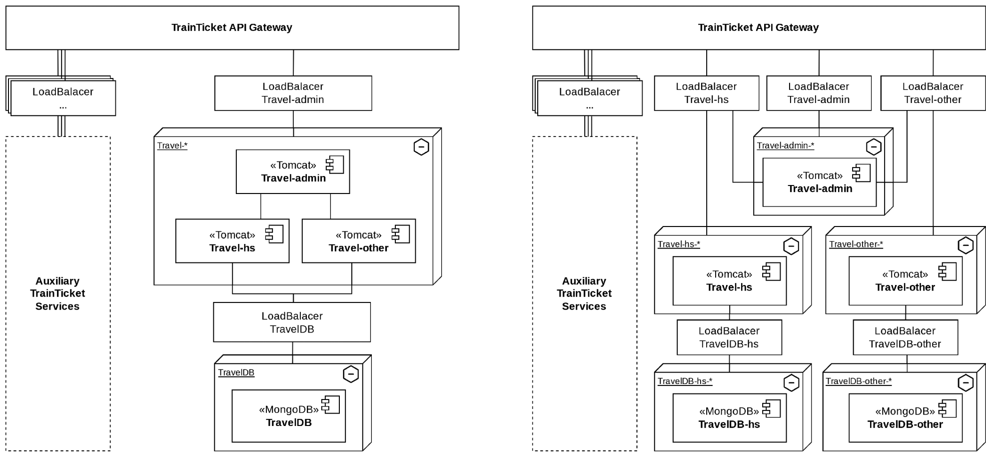
6.4 Testing Environment

6.4.1 Hardware Constraints. To deploy and test this second SUT, we used a server machine[19] equipped with two CPUs, Intel Xeon Gold 6238R 2.20GHz, each equipped with 28 physical cores (a total of 112 virtual cores). The machine is also equipped with 256 GB of RAM, 15.36 TB of SSD SATA, and 3.2 TB SSD NVMe disk storage.

6.4.2 Virtualization Layers. The server machine is managed by OpenStack, which we have used to create a cluster of five VMs. We have configured on them a Kubernetes cluster using the Rancher Kubernetes engine.[20] Four of these VMs are equipped with 10 vCPUs, 16 GB of RAM and 100 GB of storage. An additional machine is equipped with 30 vCPUs, 64 GB or RAM and 200 GB of storage. We can exploit 9.2 vCPUs of the 10-vCPU machine (0.8 vCPUs are reserved by the virtualization system) and 27.6 vCPUs of 30-vCPU machine. One of the 10-vCPU machine is used as the Kubernetes master. To prevent control interference on the application, we did not deploy other services on this machine.

6.4.3 Platforms. The virtualized hardware resources have been exploited to deploy TrainTicket with the support of Kubernetes pods. The VM equipped with 30 vCPUs (labelled as `machineAux`)

[18]https://github.com/FudanSELab/train-ticket.
[19]Hosted by RCOST (Research Center On Software Technology) at University of Sannio, Department of Engineering.
[20]https://www.rancher.com/products/rke.



(a) Train Ticket Domain Driven Service (TT-DDS). (b) TraiTicket Actor Driven Services (TT-ADS).

Fig. 24. TrainTicket $deployment_{sp}$ with DDD and ADD architectures.

hosts all of the auxiliary service (see Table 8) and their DBs, the Web-UI, and the REST API Gateway. One of the 10-vCPU machines (labelled as `machineDB`) hosts the databases of Table 9. The remaining two 10-vCPU machines (labelled as `machineX` and `machineY`) host the application business logic of the services in Table 9. The API gateway is implemented with OpenResty.[21] The services hosting the business logic are implemented using the Spring[22] framework and Apache Tomcat[23] server engine. The data are managed using MongoDB.[24]

6.5 Deployment Architectures

6.5.1 $Deployment_{sp}$. Figure 24 illustrates how TrainTicket domain-driven services (TT-DDS) and TrainTicket actor-driven services (TT-ADS) are containerized for their deployment. For the sake of readability, we present only one representative service (Travel) among the TT-DDS and how it has refined into a TT-ADS. The figure also shows the related infrastructural elements supporting the deployment onto the underlying hardware as well as horizontal scaling.

Domain-Driven Service. As shown in Figure 24(a), the TT-DDS encapsulates the whole *Travel* domain business logic in a single execution unit (a Kubernetes pod). It groups different components hosted by Tomcat: `Travel-admin`, which manages the service's stored data and operates as an authorization proxy towards the other two components, `Travel-hs` and `Travel-other`, used for accessing high-speed or other trains and for the related administration operations. All service data are stored in an instance of MongoDB (the `TravelDB`). As denoted by the symbol $*$, the whole service, including the three components, can be replicated.

Actor-Driven Service. As shown in Figure 24(b), the TT-ADS decouples the deployment of the constituent parts of each service to dedicated execution units (pods). The API Gateway redirects the incoming requests to a larger number of load balancers (3 load balancers in the case of *Travel*). Under this decomposition, requests are handled by different services according to the needs of the

[21] https://openresty.org/.
[22] https://spring.io/.
[23] https://tomcat.apache.org/.
[24] https://www.mongodb.com/.

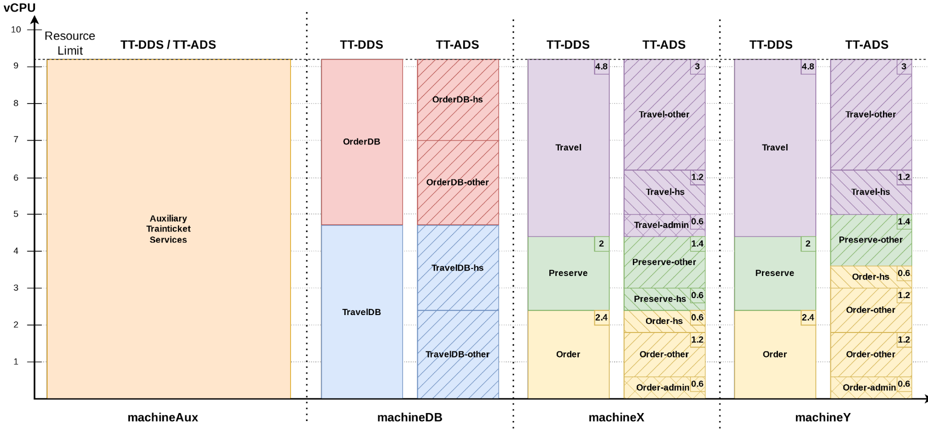


Fig. 25. TT-DDS and TT-ADS $deployment_{pm}$ and resource allocation.

actors. For instance, Logged-hs and External-hs users interact only with Travel-hs. The service Travel-admin does not directly communicate with Travel-hs and Travel-other. The requests can then be distributed among a higher number of execution units that better exploit the available resources due to their finer granularity. As shown in Figure 24, the database of the Travel service is partitioned into two different datasets managed by dedicated instances of MongoDB. One dataset holds the high-speed trains data only, while the other holds the data of the other trains. This way, each microservice serves an individual actor and operates on a specific database. This yields a finer replication schema that can be applied in case of an unbalanced distribution of actors generating a heavy workload.

6.5.2 $Deployment_{pm}$. The $deployment_{sp}$ architectures introduced in the previous section map to the IT resources managed by a Kubernetes cluster with 4 virtual machines. Figure 25 shows the $deployment_{pm}$, for both the TT-DDS and TT-ADS, obtained by applying our methodology.

The machineAux VM has enough computational resources (30 vCPUs) to host TrainTicket auxiliary services, whose operations are reported in Table 8. In order to avoid bottlenecks in this part of the system, we did not assign limits to the services hosted by this machine.

The machineDB VM hosts the DBs of the main services we consider in our analysis (see Table 9). In the case of TT-DDS, the VM hosts OrderDB and TravelDB, used by the related microservices, whereas with TT-ADS, the VM hosts the partitions of the DBs of the original service (OrderDB-hs, OrderDB-other, TravelDB-hs, TravelDB-other).

Both machineX and machineY host the main components of the business logic. In the case of TT-DDS, we deployed one instance for each service identified with DDD. In particular, nearly half of the available computational resources (4.8 vCPUs) were assigned to the Travel service, which is the most loaded service according to our operational setting (see Table 11). The rest of the resources have been assigned to the remaining two services (2.4 vCPUs to Order and 2 vCPUs to Preserve). In the case of TT-DDS, machineY uses the same resource allocation schema as machineX.

As illustrated in Figure 25, ADD adopts a finer resource allocation schema. The machineX VM hosts the same mapping per domain as in TT-DDS, but the amount of resources assigned to individual services can be controlled with higher granularity. This yields finer replication strategies and, therefore, better usage of the resources for x-axis scaling. Computational resources not used by Travel-admin and Preserve-hs can be assigned to a third replica of Order-other to handle a higher load.

6.6 Experiment Workflow

To address the research questions (RQ1–RQ4) and further generalize the previous results, we designed and conducted additional controlled experiments. We followed a number of iterations of our methodology with the same design used in the first case study (see Section 5). In each iteration, TT-DDS and TT-ADS have been deployed using the corresponding $deployment_{sp}$ and $deployment_{pm}$, and we applied the behavior mix reported in Table 11 under the increasing load, according to Λ (Equation (16)). A total amount of $\sim 20k$ operation calls was sampled per each deployed architecture in each iteration. Table 11 reports the sample size of each operation per testing session. We used the starting point, TT-DDS with no business logic replication, as baseline deployment architecture α_0 and extracted the scalability requirement under the baseline load $\lambda_0 = 5$ (i.e., the number of actors). In each iteration, the multi-level scalability assessment was executed to understand and improve the scalability of the system by refining the $deployment_{pm}$ of each architectural solution.

6.7 Experimental Results

For the sake of brevity, we do not provide the results of each iteration for this second case study. We present a summary of the final results obtained using $deployment_{sp}$ in Figure 24 and $deployment_{pm}$ in Figure 25. Figure 26 contains the plots obtained by applying the scalability assessment framework.

The system-level $\mathcal{DM}$ in Figure 26(a) shows that both architectures can handle up to 100 concurrent users without failures. The total $\mathcal{DM}$ of TT-ADS (0.657) is 30% higher compared with TT-DDS (0.941). The plot shows that TT-ADS is optimal up to $\lambda = 200$. Furthermore, for all $\lambda > 200$, TT-ADS is closer to optimal compared with TT-DDS.

Even though smaller services may lead to communication overhead, the finer resource management obtained through ADD yields higher scalability.

The scalability footprint in Figure 26(b) shows that TT-ADS yields a bigger area compared with TT-DDS. All of the operations but for p_1 and t_1 exhibit higher GSL value. Even though p_1 and t_1 exhibit better results in TT-DDS, the GSL values are very close to optimal also in TT-ADS. The TT-DDS polygon is also less regular than TT-ADS. This means that the components yield a very diverse scalability attitude since some of them cannot exploit the available resources (better allocated by using ADD).

Figure 26(c) and Figure 26(d) show a scalability gap and performance offset, respectively. Here, we can observe that the operation t_2 is the most critical one since it exhibits the highest scalability gap. In TT-DDS, t_2 fails at $\lambda = 200$ with very high performance offset (177%), whereas in TT-ADS it fails at the highest load $\lambda = 350$ with small performance offset (6%). This result is consistent with the $\mathcal{DM}$ metric and explains the rapid degradation of TT-DDS from 150 to 200 concurrent users.

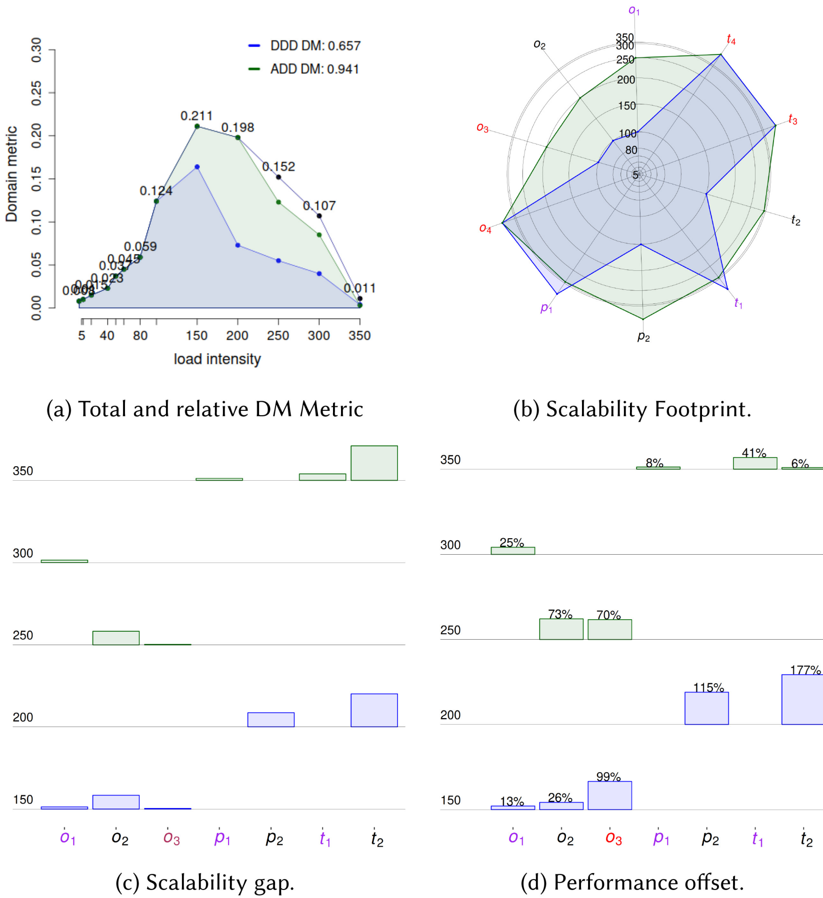
7 FINDINGS AND THREATS TO VALIDITY

On the basis of our experience in operating the proposed methodology and the results obtained from the controlled experiments, in this section we answer the research questions (RQ1–RQ4) introduced in Section 2. Then, we discuss validity threats in the following categories [39]: *external*, *internal*, *conclusion*, and *construct* validity.

7.1 Answers to the Research Questions

RQ1. To what extent can our scalability assessment workflow support decision-making over alternative microservices architectures at the system level?

The results of the controlled experiments show that both the domain metric and the derived scalability footprint are suitable quantitative instruments for providing engineers with the ability to



(a) Total and relative DM Metric

(b) Scalability Footprint.

(c) Scalability gap.

(d) Performance offset.

Fig. 26. TrainTicket methodology application results. Architectures: TT-DDS ■ and TT-ADS ■. Actors: Admin (red), {Actor}-hs (purple), {Actor}-other (black).

compare alternative architectures at the system level. In both case studies, we consistently found that the initial microservices (identified through DDD) yield worst scalability for all of the usage profiles and deployment configurations. We also found an edge case in which the coarse-grained system-level analysis prevented us from recognizing differences between the target decomposed architectures. In the first iteration of our first case study (i.e., smart mobility application), we could not find differences between ADS-RS and ADS-CQRS under the unbalanced usage profile Ω'. In this case, an analysis at finer granularity levels was required.

RQ2. To what extent can our scalability assessment workflow support resource allocation improvement at the microservice (or component) level?
According to our experience with the two case studies, the analysis workflow at component level can help achieve better performance and scalability by identifying a proper configuration in terms of software architecture, deployment, and resource allocation. For instance, during the first iteration with a smart traffic application, we identified the `EndUser` component as the most critical one. This result guided the reallocation of the available resources to obtain scalability improvements. The component-level analysis shows that the reallocation is effective, especially with the

architecture ADS-CQRS (under the unbalanced operational profile Ω'). The TrainTicket case study confirmed this result. The component-level analysis shows that the resource reallocation in TT-ADS is better compared with TT-DDS. In both case studies, the ADD yields higher scalability across operations of all components.

RQ3. Is the scalability assessment workflow able to spot scalability and performance issues at the operational level?
In both case studies, we were able to spot failing operations and measure their contribution to the overall scalability and performance degradation in each iterative step of our methodology. The results at the operational level show that the behavior of the failing operations is heterogeneous. By quantifying the scalability gap and performance offset, we were able to spot operations having a similar impact on scalability but different performance degradation. This highlights existing room for improvements by means of resource reallocation. For instance, with the smart mobility application, ADS-CQRS yields the same scalability gap for o_{10} and o_{11} (second iteration under profile Ω'). Nevertheless, o_{10} exhibits a very high performance offset compared with o_{11}.

RQ4. Can ADD practically drive further decomposition of bounded contexts yet enforce performance and scalability improvements?
The results of our controlled experiments show that ADD can be effectively used to further decompose microservices and achieve scalability improvements by taking proper architectural choices to better fit the target operational setting. The results obtained in our two case studies show that our methodology guided us towards a fine-grained decomposition yet enforced scalability improvements. In our experience, the analysis workflow provided us with decision support instruments able to compare alternative architectures and identify system components that require better resource allocation. Concerning practicability, we would like to point out that ADD can refine each bounded context that has limited dimension even though the size of the overall system is large. For instance, TrainTicket has 41 microservices, but we could selectively apply ADD to a subset of them. In this way, each bounded context can be refined by a different development and quality assurance team to split the effort and improve scalability.

7.2 Threats to Validity

External validity. External validity concerns the generalizabilty of the results. Threats in this category have been addressed by selecting two representative case studies in our target application domain, that is, performance-sensitive, service-based systems. One is representative of a wide class of modern data-intensive, edge-cloud systems whereas the other is a widespread benchmark microservices application in the e-commerce domain. We also chose a common technology stack that is modern enough to justify a decomposition through incremental refactoring actions rather than a complete rewrite and replacement. Our approach needs to work within a preproduction testing environment in which we can have control over the factors of interest. This setting is common with DevOps practices and tools in modern infrastructure that support continuous deployment [11].

Internal validity. Threats to internal validity are related to internal factors that could have influenced the results. To reduce threats in this class, we carefully designed our experimental campaign to control the factors of interest in both case studies. We controlled load, usage profiles, deployment architectures, and resources allocation. We avoided the adoption of auto-scaling mechanisms provided by our deployment infrastructure to reduce the risk of obscuring causal inferences due to uncontrolled events. This manual manipulation has been crucial to assess cause-and-effect relations between external factors and the effects measured by using the multi-level analysis. We also mitigated the effects of uncontrolled events as much as we could. The tests have been repeated

multiple times and during all the tests we monitored the network to avoid anomalous traffic conditions (e.g., peaks) affecting the experiments' results.

Conclusion validity. Since testing sessions were guided by stochastic sampling of synthetic users from multiple categories, there existed the possibility that results had been produced by chance. We addressed this threat by following the practical guidelines in [40]. We sampled a large number of invocations per individual operation by repeating each test session three times. We also conducted a pairwise comparison among samples between consecutive sessions to assess absence of statistical difference using the Mann-Whitney U-test. We also adopted this latter statistical test to compare the scalability footprints and compute the p-value with significance level $\alpha = 0.05$ (detailed results are available in the dataset paired with this article). The Cliff delta has been adopted to measure the effect size. Finally, we use multiple visualization techniques complemented with quantitative analysis to draw our conclusions. This helped us to collectively agree on the interpretation of the analysis and collectively formulate the answers to the research questions.

Construct validity. The applicability of our multi-level analysis approach depends on the accuracy of the target operational setting under consideration, which determines the scalability requirement described in Section 4. Therefore, a careful analysis of the production usage is required. As discussed in [12], approaches to deal with this threat include: using related systems as a proxy for the SUT, conducting user surveys, and analyzing log data from an existing version of the SUT. Therefore, while some specific results about the quality of the considered architectures depend on the settings considered for the case studies, the overall methodology, decomposition strategy, and multi-level scalability assessment can be generalized.

8 RELATED WORK

The main contribution of this article is a methodology that complements existing approaches and provides better mechanisms to decompose microservices enforcing scalability improvements. We discuss related work by focusing on approaches for decomposing and modularizing microservices and approaches to assess their scale of operation. In the following, for both areas, we discuss the state-of-the-art, highlighting the main shortcomings.

8.1 Microservices Decomposition

Architectural patterns and antipatterns. A broad literature review of the challenges associated with the microservices architectural style can be found in [41]. According to the existing principles in microservices architecture, there are common mistakes occurring during the decomposition process. These pitfalls can be identified by looking at the so-called bad smells in the source code [42]. A number of bad practices, antipatterns, and their potential solutions have been collected by conducting interviews with experienced developers of cloud-native applications based on microservices [43–45]. The outcome is a taxonomy of bad practices and possible refactoring actions both from the organization perspective (e.g., organization of the development team) and from the technical perspective (e.g., code smells and communication issues). The work in [46] introduces guidelines for companies that need to migrate to microservices based on the analysis of a set of metrics they should collect before re-architecting their monolithic system. To extract these metrics, the authors conducted interviews with professionals to derive an assessment framework based on grounded theory [47]. As stated in [6, 43, 46], (anti) patterns and guidelines are system agnostic and can be adopted by companies to decide whether to adopt microservices or not and how to define migration plans by following successful stories from past experience. These approaches cannot be reasonably used to conduct a quantitative assessment of system-specific alternative architectures, which is instead the major goal of our methodology.

Automated decomposition of monolithic systems. Automated suggestions of candidate service cuts based on static analysis of a target codebase have been recently proposed [48, 49]. The authors propose the adoption of clustering techniques to extract microservices from monoliths by maximizing high cohesion and minimizing low coupling. These approaches do not take into account the impact of possible dependencies due to dynamically typed languages on architecture-level maintainability. Recent studies reveal that the impact is higher than that of explicit dependencies [50]. Other existing semi-automated approaches based on clustering recommend microservice candidates and suggest possible refactoring actions to be applied in a migration scenario [51]. Microservice candidates are selected by limiting the average development team size and the service size (lines of code) according to recent empirical studies [52]. Even though these approaches suggest possible decomposition strategies grounded on traditional architectural principles, they do not provide engineers with suitable methods to assess the given decomposition taking into account the expected operational setting and perceived quality of the product by end users. The Mono2Micro platform [53] applies semi-automated refactoring of Java monolithic applications into microservices. The approach uses static analysis if the codebase, combined with runtime dynamic analysis of the execution, traces to make recommendations in terms of groupings of classes in the monolith that can serve as starting points for the identification of bounded contexts. The Functionality-Oriented Service Candidate Identification framework [54] suggests service candidates by analyzing the execution traces using a search-based approach that optimizes three quality criteria of service candidates: independence of functionality, modularity, and independence of evolvability. CARGO [55] is a microservice partitioning and refinement tool that statically analyzes JEE applications to build a system dependency graph enriched with semantically meaningful relationships (e.g., call-return, and database transaction). This graph yields better community detection (candidate microservices) by reducing distributed database transactions.

These approaches are conceptually different from ADD since they automatically recommend candidate microservices starting from a monolithic system. Our approach can then complement these methods to refine the suggested microservices contexts taking into account the actors and their role.

Actor- and role-driven decomposition. Actor-based and role-based abstractions provide alternative perspectives to develop decomposition processes. Even though the notion of actor is an established modeling concept used to decompose and modularize complex service collaborations —for instance, in business process management [56] and multi-agent systems [57] — it received less attention for microservices systems. Mainstream approaches focus instead on the notion of bounded context. For this reason, there is a lack of knowledge regarding how actors should be systematically integrated in established decomposition frameworks as well as the impact of this integration. Workload data–aided approaches have been proposed [44], still with the ultimate goal of identifying the domain entities that may collectively represent the bounded contexts of a monolithic system. ADD is conceptually different since it assumes the existence of bounded contexts and refines them according to the actors and their role, that is, the embodiment of the participation of an actor [58].

8.2 Scalability Assessment of Microservices

The systematic gray literature review in [4] recognizes performance testing and scalability analysis of microservices as painful activities. Motivated by this issue, the research community conceived of novel methodologies that provide engineers with decomposition decision support based on quantitative analysis of nonfunctional qualities. The process presented in [59] evaluates a target architecture by balancing nonfunctional requirement satisfaction at the level of individual microservices as well as the overall system. The authors introduce a managing subsystem (i.e., adaptation layer)

that monitors the trade-off and assists the managed microservices in achieving such a balance. However, the managing layer can introduce non-negligible overhead that could even limit performance or scalability. Other common approaches based on runtime management of resources are those enacting auto-scaling mechanisms. The work presented in [60] addresses the problem of selecting appropriate performance metrics to activate such mechanisms. The authors investigate the use of relative and absolute metrics to understand when to activate the appropriate actions. In contrast to our approach, these frameworks work at runtime along with the target system in production. In principle, they could complement our assessment methodology since they assume the existence of a microservices system, whereas we aim at guiding towards the "right" architecture before the production phase.

The conceptual framework introduced in [5] envisions an iterative technique to guide the migration steps by continuously monitoring a system in production in order to collect operational data and use performance analysis as feedback for the decomposition process. Following the same principle, our assessment methodology leverages instead the runtime evidence collected through load testing. Our scalability assessment framework is grounded on the quantitative system-level *domain metric* introduced in [12, 21]. This approach aims at evaluating the performance of alternative deployment configurations for microservice systems. Performance here is defined as a function of the workload intensity as originally introduced in [61]. The PPTAM software tool implements the domain metric assessment method and has been presented in [33]. The domain metric approach as well as its own modifiable software tool [62] turned out to be effective in evaluating microservice architectures based on the notion of *scalability requirement* [21], as a performance threshold empirically extracted by observing the responsiveness of individual services under "no load" [63–65]. A first draft of our vision has been introduced in [8]. This work describes a scalability assessment methodology able to guide the decomposition process through coarse-grained information extracted by applying the domain metric to evaluate the target system as a whole. Our assessment methodology extends this approach by introducing a multi-level analysis method to extract insights on the target application from different angles: system, service, and operation. The idea is to aid engineering decisions at different granularity levels in order to choose among alternative decomposition strategies as well as extract insights for resource allocation.

9 CONCLUSION

In this article, we introduced the ADD strategy to refine bounded-contexts into fine-grained microservices used by multiple actors with the aim of increasing the scalability of the target system. ADD is complemented by a scalability assessment framework to compare alternative deployment architectures at multiple granularity levels (system, component, and operation). The framework builds on qualitative and quantitative methods that exploit the so-called domain metric and guide the refinement of resource allocation and the re-deployment. We adopted engineering research to evaluate our methodology by conducting an in-depth, detailed examination of two case studies: (1) a real smart mobility system and (2) a popular e-commerce benchmark commonly adopted by the research community. We demonstrated the benefits of the proposed strategy and the assessment methodology, showing that a finer-grained decomposition of bounded contexts driven by ADD led to scalability improvements in both case studies.

As future work, we aim to leverage the metrics and instruments introduced in this work to automatically apply architectural reconfiguration under the constraints imposed by the available resources. We believe that this automated reconfiguration along with the ADD approach has the potential of improving existing autoscaling methods. We also plan to study the usage of ADD over evolutionary steps of the life cycle. Understanding how to approach evolution in a cost-effective manner requires systematic techniques to interpret cause-and-effect relations between changes of

domain aspects and changes of the actors as well as the magnitude of such changes to determine the minimal set of assets (modified bounded contexts, services, actors) that need to be refined and (re)analyzed. In addition to performance and scalability, we are going to consider rigorous and quantitative analysis of the impact of the ADD on other important nonfunctional properties, such as reliability as well as the relationships between reliability and performance [29]. Concerning reliability, we envisage potential benefits since ADD microservices (1) are more granular, thus, faults are more localized; and (2) are used by specific actors, thus, a faulty microservice typically affects only a limited number of users. Preliminary results collected over our experimental campaign confirmed these observations as we observed in general less failures (higher reliability) after applying ADD.

REFERENCES

[1] Sam Newman. 2021. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Incorporated. https://books.google.it/books?id=MTClswEACAAJ.

[2] Mark Richards. 2016. *Microservices vs. Service-Oriented Architecture*. O'Reilly Media, Inc.

[3] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. 2018. Migrating towards microservice architectures: An industrial survey. In *2018 IEEE International Conference on Software Architecture (ICSA'18)*. IEEE Computer Society, 29–39.

[4] Jacopo Soldani, Damian A. Tamburri, and Willem-Jan Van Den Heuvel. 2018. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software* 146 (2018), 215–232. DOI: http://dx.doi.org/10.1016/j.jss.2018.09.082

[5] Andrea Janes and Barbara Russo. 2019. Automatic performance monitoring and regression testing during the transition from monolith to microservices. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'19)*. Berlin, Germany, 163–168. DOI: http://dx.doi.org/10.1109/ISSREW.2019.00067

[6] Chris Richardson. 2018. *Microservices Patterns: With Examples in Java*. Manning Publications. https://books.google.it/books?id=UeK1swEACAAJ.

[7] Florian Rademacher, Jonas Sorgalla, and Sabine Sachweh. 2018. Challenges of domain-driven microservice design: A model-driven perspective. *IEEE Software* 35, 3 (2018), 36–43. DOI: http://dx.doi.org/10.1109/MS.2018.2141028

[8] Matteo Camilli, Carmine Colarusso, Barbara Russo, and Eugenio Zimeo. 2020. Domain metric driven decomposition of data-intensive applications. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'20)*. Coimbra, Portugal, 189–196. DOI: http://dx.doi.org/10.1109/ISSREW51248.2020.00071

[9] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy E. Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. 2017. Performance engineering for microservices: Research challenges and directions. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion (ICPE'17 Companion)*. ACM, New York, NY, 223–226. DOI: http://dx.doi.org/10.1145/3053600.3053653

[10] Matteo Camilli, Andrea Janes, and Barbara Russo. 2022. Automated test-based learning and verification of performance models for microservices systems. *Journal of Systems and Software* 187 (2022), 111225. DOI: http://dx.doi.org/10.1016/j.jss.2022.111225

[11] Antonia Bertolino, Guglielmo Angelis, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2023. DevOpRET: Continuous reliability testing in DevOps. *Journal of Software: Evolution and Process* 35, 3 (2013), e2298. DOI: http://dx.doi.org/10.1002/smr.2298

[12] Alberto Avritzer, Vincenzo Ferme, Andrea Janes, Barbara Russo, André van Hoorn, Henning Schulz, Daniel Menasché, and Vilc Rufino. 2020. Scalability assessment of microservice architecture deployment configurations: A domain-based approach leveraging operational profiles and load tests. *Journal of Systems and Software* 165 (2020), 110564. DOI: http://dx.doi.org/10.1016/j.jss.2020.110564

[13] Paul Ralph, Sebastian Baltes, Domenico Bianculli, Yvonne Dittrich, Michael Felderer, Robert Feldt, Antonio Filieri, Carlo A. Furia, Daniel Graziotin, Pinjia He, Rashina Hoda, Natalia Juristo, Barbara A. Kitchenham, Romain Robbes, Daniel Méndez, Jefferson Molleri, Diomidis Spinellis, Miroslaw Staron, Klaas-Jan Stol, Damian A. Tamburri, Marco Torchiano, Christoph Treude, Burak Turhan, and Sira Vegas. 2020. ACM SIGSOFT empirical standards. *CoRR* abs/2010.03525 (2020). arXiv:2010.03525 https://arxiv.org/abs/2010.03525.

[14] Antonio De Iasio, Angelo Furno, Lorenzo Goglia, and Eugenio Zimeo. 2019. A microservices platform for monitoring and analysis of IoT traffic data in smart cities. In *2019 IEEE International Conference on Big Data (Big Data'19), Los Angeles, CA, December 9-12, 2019*. 5223–5232. DOI: http://dx.doi.org/10.1109/BigData47090.2019.9006025

[15] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Benchmarking microservice systems for software engineering research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*. ACM. Gothenburg Sweden, DOI: http://dx.doi.org/10.1145/3183440.3194991

[16] Mary B. Rosson and John M. Carroll. 2002. *Usability Engineering: Scenario-Based Development of Human-Computer Interaction*. Morgan Kaufmann.

[17] Eric Evans. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

[18] Paul Clements, David Garlan, Reed Little, Robert Nord, and Judith Stafford. 2003. Documenting software architectures: Views and beyond. In *International Conference on Software Engineering,* IEEE, Los Alamitos, CA, 740. DOI: http://dx.doi.org/10.1109/ICSE.2003.1201264

[19] Martin L. Abbott and Michael T. Fisher. 2015. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Addison-Wesley Professional.

[20] Martin L. Abbott and Michael T. Fisher. 2017. *Scalability Rules: Principles for Scaling Web Sites*. Addison-Wesley.

[21] Alberto Avritzer, Vincenzo Ferme, Andrea Janes, Barbara Russo, Henning Schulz, and André van Hoorn. 2018. A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing. In *Proceedings of the 12th European Conference on Software Architecture (ECSA'18)*. 159–174. DOI: http://dx.doi.org/doi.org/10.1007/978-3-030-00761-4_11

[22] Friedrich Pukelsheim. 1994. The three sigma rule. *The American Statistician* 48, 2 (1994), 88–91.

[23] Oliver C. Ibe. 2013. Basic concepts in probability. In *Markov Processes for Stochastic Modeling (Second Edition)* , Oliver C. Ibe (Ed.). Elsevier, Oxford, 1–27. DOI: http://dx.doi.org/10.1016/B978-0-12-407795-9.00001-3

[24] Rick Durrett. 2019. *Probability: Theory and Examples*, Vol. 49. Cambridge University Press.

[25] OMG Available Specification. 2007. OMG unified modeling language (OMG UML), superstructure, v2. 1.2. *Object Management Group* 70 (2007).

[26] Vincenzo Ferme and Cesare Pautasso. 2018. A declarative approach for performance tests execution in continuous software development environments. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE'18)*. ACM, New York, NY, 261–272. DOI: http://dx.doi.org/10.1145/3184407.3184417

[27] Christian Vögele, André van Hoorn, Eike Schulz, Wilhelm Hasselbring, and Helmut Krcmar. 2018. WESSBAS: Extraction of probabilistic workload specifications for load testing and performance prediction–a model-driven approach for session-based application systems. *Software and Systems Modeling* 17, 2 (May 2018), 443–477. DOI: http://dx.doi.org/10.1007/s10270-016-0566-5

[28] Microsoft Azure. 2022. Using tactical DDD to design microservices. (Dec. 2022.) https://learn.microsoft.com/en-us/azure/architecture/microservices/model/tactical-ddd.

[29] Matteo Camilli, Antonio Guerriero, Andrea Janes, Barbara Russo, and Stefano Russo. 2022. Microservices integrated performance and reliability testing. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test (AST'22)*. ACM, New York, NY, 29–39. DOI: http://dx.doi.org/10.1145/3524481.3527233

[30] Matteo Camilli, Carlo Bellettini, Angelo Gargantini, and Patrizia Scandurra. 2018. Online model-based testing under uncertainty. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE'18)*. IEEE, 36–46.

[31] Daniele Gadler, Michael Mairegger, Andrea Janes, and Barbara Russo. 2017. Mining logs to model the use of a system. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'17)*. Toronto Ontario, 334–343. DOI: http://dx.doi.org/10.1109/ESEM.2017.47

[32] Nadia Ranaldo and Eugenio Zimeo. 2016. Capacity-driven utility model for service level agreement negotiation of cloud services. *Future Generation Computer Systems* 55 (2016), 186–199. DOI: http://dx.doi.org/10.1016/j.future.2015.03.007

[33] Alberto Avritzer, Daniel S. Menasché, Vilc Rufino, Barbara Russo, Andrea Janes, Vincenzo Ferme, André van Hoorn, and Henning Schulz. 2019. PPTAM: Production and performance testing based application monitoring. In *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE'19)*. Mumbai India, 39–40. DOI: http://dx.doi.org/10.1145/3302541.3311961

[34] Robert Grissom and John Kim. 2012. *Effect Sizes for Research: Univariate and Multivariate Applications, Second Edition*. Routledge. 1–434. DOI: http://dx.doi.org/10.4324/9780203803233

[35] Norman Cliff. 2014. *Ordinal Methods for Behavioral Data Analysis*. Taylor & Francis. DOI: http://dx.doi.org/10.4324/9781315806730

[36] Jeanine Romano, Jeffrey D. Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys. In *Annual Meeting of the Florida Association of Institutional Research*. 1–33.

[37] Indika P. Kumara, Mohamed Ariz, Mohan Baruwal Chhetri, Majeed Mohammadi, Willem-Jan van den Heuvel, and Damian A. Tamburri. 2022. FOCloud: Feature model guided performance prediction and explanation for deployment configurable cloud applications. *IEEE Transactions on Services Computing* 16, 1 (2022), 302–314. DOI: http://dx.doi.org/10.1109/TSC.2022.3142853

[38] Carmine Colarusso, Antonio De Iasio, Angelo Furno, Lorenzo Goglia, Mohammed A. Merzoug, and Eugenio Zimeo. 2022. PROMENADE: A big data platform for handling city complex networks with dynamic graphs. *Future Generation Computer Systems* 137 (2022), 129–145. DOI: http://dx.doi.org/10.1016/j.future.2022.07.008

[39] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2000. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA.
[40] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, New York, NY, 1–10. DOI: http://dx.doi.org/10.1145/1985793.1985795
[41] Nuha Alshuqayran, Nour Ali, and Roger Evans. 2016. A systematic mapping study in microservice architecture. In *Proceedings of the IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA'16)*. Macau China, 44–51. DOI: http://dx.doi.org/10.1109/SOCA.2016.15
[42] Andrés Carrasco, Brent van Bladel, and Serge Demeyer. 2018. Migrating towards microservices: Migration and architecture smells. In *Proceedings of the 2nd International Workshop on Refactoring (IWoR'18)*. ACM, New York, NY, 1–6. DOI: http://dx.doi.org/10.1145/3242163.3242164
[43] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A. Tamburri, and Theo Lynn. 2018. Microservices migration patterns. *Software: Practice and Experience* 48, 11 (2018), 2019–2042. DOI: http://dx.doi.org/10.1002/spe.2608 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2608.
[44] Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, and Stefan Wagner. 2019. From monolith to microservices: A classification of refactoring approaches. In *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer (Eds.). Springer International Publishing, Cham, 128–141. DOI: http://dx.doi.org/10.1007/978-3-030-06019-0_10
[45] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. 2020. *Microservices Anti-patterns: A Taxonomy*. Springer International Publishing, Cham, 111–128. DOI: http://dx.doi.org/10.1007/978-3-030-31646-4_5
[46] Florian Auer, Valentina Lenarduzzi, Michael Felderer, and Davide Taibi. 2021. From monolithic systems to microservices: An assessment framework. *Information and Software Technology* 137 (2021), 106600. DOI: http://dx.doi.org/10.1016/j.infsof.2021.106600
[47] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded theory in software engineering research: A critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. ACM, New York, NY, 120–131. DOI: http://dx.doi.org/10.1145/2884781.2884833
[48] Sinan Eski and Feza Buzluca. 2018. An automatic extraction approach: Transition to microservices architecture from monolithic application. In *Proceedings of the 19th International Conference on Agile Software Development: Companion (XP'18)*. ACM, New York, NY, Article 25, 6 pages. DOI: http://dx.doi.org/10.1145/3234152.3234195
[49] Miguel Brito, Jácome Cunha, and João Saraiva. 2021. Identification of microservices from monolithic applications through topic modelling. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC'21)*. ACM, New York, NY, 1409–1418. DOI: http://dx.doi.org/10.1145/3412841.3442016
[50] Wuxia Jin, Dinghong Zhong, Yuanfang Cai, Rick Kazman, and Ting Liu. 2022. Evaluating the impact of possible dependencies on architecture-level maintainability. *IEEE Transactions on Software Engineering* (2022), 1–1. DOI: http://dx.doi.org/10.1109/TSE.2022.3171288
[51] Genc Mazlami, Jürgen Cito, and Philipp Leitner. 2017. Extraction of microservices from monolithic software architectures. In *2017 IEEE International Conference on Web Services (ICWS'17)*. Honolulu Hawaii, 524–531. DOI: http://dx.doi.org/10.1109/ICWS.2017.61
[52] Gerald Schermann, Jürgen Cito, and Philipp Leitner. 2016. All the services large and micro: Revisiting industrial practice in services computing. In *Service-Oriented Computing – ICSOC 2015 Workshops*, Alex Norta, Walid Gaaloul, G. R. Gangadharan, and Hoa Khanh Dam (Eds.). Springer, Berlin, 36–47. DOI: http://dx.doi.org/10.1007/978-3-662-50539-7_4
[53] Anup K. Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. 2021. Mono2Micro: A practical and effective tool for decomposing monolithic Java applications to microservices. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. ACM, New York, NY, 1214–1224. DOI: http://dx.doi.org/10.1145/3468264.3473915
[54] Wuxia Jin, Ting Liu, Yuanfang Cai, Rick Kazman, Ran Mo, and Qinghua Zheng. 2021. Service candidate identification from monolithic systems based on execution traces. *IEEE Transactions on Software Engineering* 47, 5 (2021), 987–1007. DOI: http://dx.doi.org/10.1109/TSE.2019.2910531
[55] Vikram Nitin, Shubhi Asthana, Baishakhi Ray, and Rahul Krishna. 2022. CARGO: AI-Guided Dependency Analysis for Migrating Monolithic Applications to Microservices Architecture. (2022). DOI: http://dx.doi.org/10.48550/ARXIV.2207.11784
[56] OMG. 2011. Business Process Model And Notation. (2011). http://www.omg.org/spec/BPMN/2.0/. Last accessed on Oct, 2022.
[57] Virginia Dignum. 2009. *Handbook of Research on Multi-agent Systems: Semantics and Dynamics of Organizational Models*. IGI Global.

[58] Robrecht Haesevoets, Danny Weyns, and Tom Holvoet. 2014. Architecture-centric support for adaptive service collaborations. *ACM Transactions on Software Engineering and Methodology* 23, 1, Article 2 (feb 2014), 40 pages. DOI: http://dx.doi.org/10.1145/2559937

[59] Sara Hassan and Rami Bahsoon. 2016. Microservices and their design trade-offs: A self-adaptive roadmap. In *2016 IEEE International Conference on Services Computing (SCC'16)*. San Francisco California, 813–818. DOI: http://dx.doi.org/10.1109/SCC.2016.113

[60] Emiliano Casalicchio and Vanessa Perciballi. 2017. Auto-scaling of containers: The impact of relative and absolute metrics. In *Proceedings of FAS*W@SASO/ICCAC*. Tucson Arizona, 207–214. DOI: http://dx.doi.org/10.1109/FAS-W.2017.149

[61] Elaine J. Weyuker and Alberto Avritzer. 2002. A metric for predicting the performance of an application under a growing workload. *IBM Systems Journal* 41, 1 (2002), 45–54. DOI: http://dx.doi.org/10.1147/sj.411.0045

[62] Alberto Avritzer, Matteo Camilli, Andrea Janes, Barbara Russo, Jasmin Jahič, André van Hoorn, Ricardo Britto, and Catia Trubiani. 2021. PPTAM$^{\lambda}$: What, where, and how of cross-domain scalability assessment. In *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C'21)*. Stuttgart Germany, 62–69. DOI: http://dx.doi.org/10.1109/ICSA-C52384.2021.00016

[63] Alberto Avritzer, Ricardo Britto, Catia Trubiani, Matteo Camilli, Andrea Janes, Barbara Russo, André van Hoorn, Robert Heinrich, Martina Rapp, Jörg Henß, and Ram K. Chalawadi. 2022. Scalability testing automation using multivariate characterization and detection of software performance antipatterns. *Journal of Systems and Software* 193 (2022), 111446. DOI: http://dx.doi.org/10.1016/j.jss.2022.111446

[64] Alberto Avritzer, Ricardo Britto, Catia Trubiani, Barbara Russo, Andrea Janes, Matteo Camilli, André van Hoorn, Robert Heinrich, Martina Rapp, and Jörg Henß. 2021. A multivariate characterization and detection of software performance antipatterns. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE'21)*. ACM, New York, NY, 61–72. DOI: http://dx.doi.org/10.1145/3427921.3450246

[65] Matteo Camilli and Barbara Russo. 2022. Modeling performance of microservices systems with growth theory. *Empirical Software Engineering* 27, 2 (11 Jan 2022), 39. DOI: http://dx.doi.org/10.1007/s10664-021-10088-0

Received 7 May 2022; revised 2 January 2023; accepted 10 January 2023