Proactive-Reactive Microservice Architecture Global Scaling

Lorenzo Bacchiani[a,c], Mario Bravetti[a], Saverio Giallorenzo[a,b], Maurizio Gabbrielli[a], Gianluigi Zavattaro[a,b], Stefano Pio Zingaro[a]

[a] *DISI, University of Bologna, Via Zamboni, 33, Bologna, 40126, Italy,*
[b] *OLAS Team, INRIA, Sophia Antipolis, France,*
[c] *Corresponding author: Lorenzo Bacchiani, lorenzo.bacchiani2@unibo.it*

Abstract

We develop a novel approach for run-time global adaptation of microservice applications, based on synthesis of architecture-level reconfigurations. More precisely, we devise an algorithm for proactive-reactive automatic scaling that reaches a target system's Maximum Computational Load by performing optimal deployment orchestrations. We evaluate our approach by developing a platform for the modelling and simulation of microservice architectures, and we use such a platform to compare local/global and reactive/proactive scaling. Empirical benchmarks, obtained through our platform, show that that proactive global scaling consistently outperforms the reactive approach, but the best performances can be obtained by our original approach for mixing proactivity and reactivity. In particular, our approach surpasses the state-of-the-art when both performance and resource consumption are considered.

Keywords: Microservices, Global Scaling, Proactive-Reactive Scaling

1. Introduction

Modern Cloud architectures use microservices as their highly modular and scalable components, which, in turn, enable effective practices such as continuous deployment and horizontal (auto)scaling. Although these practices are already beneficial, they can be further improved by exploiting the interdependencies within an architecture (functional dependencies between microservice requests), instead of focusing on a single microservice. Architecture-level dynamic deployment orchestrations bring significant advances over the traditional local scaling technique: they eliminate the *"domino" effect* of un-

structured scaling, i.e., single services scaling one after the other (cascading slowdowns) due to local workload monitoring, as done in, e.g., Kubernetes [1]. A fundamental feature of these approaches is that, besides improving system performance (e.g., latency) by eliminating the domino effect, they also optimise resource usages, e.g., number of deployed microservice instances.

In this work, we first propose a new approach for architecture-level adaptation, called *global scaling*, that overcomes the drawbacks of the traditional scaling approach. The global scaling algorithm leverages the knowledge of the microservice dependencies and it reaches, via architecture-level reconfigurations, a target system Maximum Computational Load (MCL), i.e., the maximum supported frequency for inbound requests. The idea is that, in a *reactive* approach, i.e., by monitoring at run-time the inbound workload, our algorithm causes the system to be always in the reachable configuration, with the least amount of deployed microservice instances, that better fits such workload. Global reconfigurations are targeted at guaranteeing a given increment/decrement of the system MCL.

We then endow our approach with *proactive* capabilities using an off-the-shelf machine learning module to forecast the inbound workload, further improving performance. However, predictors are weak against exceptional events, resulting in the application of inappropriate deployment orchestrations (either wasting resources or degrading the level of service). Thus, in this article, we also contribute a novel *proactive-reactive* algorithm for mixing the measured workload and the predicted one. Our algorithm casts the comparison as the capacity of the system to deal with a given workload (system MCL), obtained by its current scaling reconfiguration. Hence, we have a way to estimate both over- and under-scaling of proactive global scaling, given by the distance w.r.t. the system MCL induced by the actual traffic.

Summarising, the main concepts considered in this article are *global scaling* of microservice architectures based on microservice functional dependencies, and *proactive-reactive* adaptation to time-varying workload. We investigate these concepts driven by the following research questions:

RQ1. Is a global scaling approach based on microservice request functional dependencies feasible and effective?

RQ2. Is dealing with the complexity of mixing reactive and proactive global scaling worthwhile?

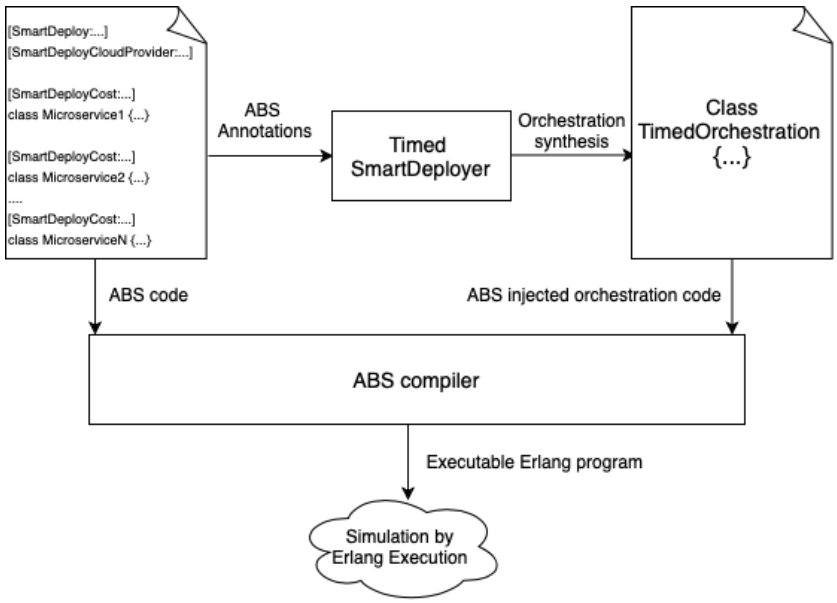


Figure 1: Integrated timed architectural modelling/execution language toolchain.

RQ3. Can we devise a mixing algorithm that effectively focuses both on performance and saving resources?

To answer these research questions we proceed by modelling and simulating our global scaling approach (in its reactive and/or proactive variants) thanks to a novel *integrated timed architectural modelling/execution language* based on a *timed extension* of the SmartDeployer tool [2, 3] for the Abstract Behavioral Specification (ABS) language [4].

ABS is an actor-based timed object-oriented language, suitable for designing, verifying and evaluating concurrent/distributed systems. In particular, it allows for modelling and simulation by exploiting its double nature: it is both a process algebra (with probabilistic/timed formal semantics) and a programming language (compiled and executed, e.g., via Erlang backend).

SmartDeployer exploits dedicated ABS code annotations expressing *architectural properties* of: the modeled distributed system (global architectural invariants and allowed reconfigurations), its VMs (their characteristics and the resource they provide) and its software components/services (accounting for architectural dependencies and invariants). Such annotations are read by SmartDeployer that, at compile-time, checks them for satisfiability (accounting for the desired target configuration requirements [2, 3], and archi-

tectural invariants) and synthesizes *deployment* orchestrations that build the system architecture and each of its specified reconfigurations. Simmetrically, it generates the *undeployment* orchestrations to undo such reconfigurations. SmartDeployer implements the algorithm of [2, 3] and solves the *optimal deployment problem*: given an initial microservice system, a set of available nodes, and a new target microservice to deploy, find a sequence of reconfiguration actions that, applied to the initial system, leads to a new deployment that includes the target microservice. Thus, SmartDeployer generates optimal (un)deployment functions using ABS as orchestration language. Such functions can be invoked by services, thus realising run-time adaptation.

Here, we introduce the *Timed SmartDeployer* tool that fully integrates, also accounting for time aspects, ABS with annotation-based specification of architectural properties. In particular, Timed SmartDeployer, as can be seen in Fig. 1, generates *timed deployment orchestrations* that also manage time aspects of the execution: i.e., they use ABS timed primitives to dynamically set VM speeds (based on actually used CPU cores) and overall startup time for the architectural reconfiguration. As we will see, the timed features of orchestrations are essential to model, in an MCL-consistent way, adaptation actions enacted by our global scaling algorithm.

The modelling and execution capabilities of our timed integrated language, make it possible to anticipate at design level performance-related issues. This fosters an approach where the analysis of the consequences of deployment decisions are available early on. As a matter of fact, our ABS model (built with the timed integrated language) allows us to evaluate the performance of our approach on a realistic microservice system: an Email Pipeline Processing System [2, 3]. The system model is built by considering: static aspects of the architecture (annotations) and ABS code modelling service behaviour. We simulate system execution using inbound traffic taken from the Enron dataset [5], a real diurnal load pattern inspired to that in [6] and part of an IMAPS email traffic similar to that in [7]. To highlight the extent of the advantages of our global scaling w.r.t. the local one (traditionally used in the literature [8, 9, 10] and by, e.g., Kubernetes [1]), we produce two ABS programs: one implementing our scaling approach (and all its variants) and one just dealing with scaling needs at the level of single microservices. Our results show that our scaling algorithm avoids cascading slowdowns that affect local scaling. Moreover, to show the need for our proactive-reactive algorithm w.r.t. a purely proactive one, we selectively pick outliers from the Enron dataset and run benchmarks to evaluate its performance. Finally,

we implement in our ABS models the proactive-reactive algorithm of [11] to compare it with ours. The ABS models we use to run our benchmarks are publicly available at [12].

Wrapping up, our work has led to the following contributions: (*i*) a novel algorithm for proactive-reactive global scaling that efficiently adapts microservice architectures to time-varying workloads (as we will see in Section 4.1, our algorithm crucially exploits microservice request functional dependencies); (*ii*) a novel integrated timed architectural modelling/execution language based on a timed extension of SmartDeployer that makes use of timed instructions of ABS to automatically generate timed deployment orchestrations; (*iii*) implementation of system service execution/scaling mechanism for the Email Pipeline Processing System [2, 3] and the TeaStore [13, 14], via such integrated modelling/execution language; and (*iv*) benchmarks, based on different datasets [5, 6, 7], to prove the effectiveness of our scaling approach and proactive-reactive algorithm.

The remainder of the article is structured as follows. In Section 2, we review the current state of the art of proactive and/or reactive scaling approaches. In Section 3, we introduce the main basic concepts of this work. In Section 4, we present the framework we use to build our global scaling approach. In Section 5 and Section 6, we, respectively, describe and evaluate our scaling algorithm. Finally, in Section 7, we conclude the article.

This article represents a journal version of the work done in [15, 16] that additionally includes: (*i*) the proactive-reactive global scaling algorithm ABS code; (*ii*) the local scaling algorithm ABS code; and (*iii*) novel performance evaluation, e.g., oracle local scaling against the reactive global one.

2. State of the Art in Microservices Autoscaling

The main contribution of this article regards the introduction of a novel approach to support global scaling in microservice architectures. In this section, we review the current state of the art, starting with the approaches adopted for local scaling.

Local Scaling. Local scaling focuses on adjusting the number of instances at the level of a single microservice. These approaches can be reactive (triggered by specific events) or proactive (aimed at preventing undesired events). Recent examples of reactive local scaling include Bayesian Optimization techniques [17] and Fuzzy Logic [18]. Proactive local scaling often involves predic-

tion techniques to create early scaling mechanisms, using probabilistic modelling frameworks or time series analysis techniques, such as k-means [19] and neural networks [20, 21]. Researchers have also proposed hybrid approaches that mix reactive and proactive elements to improve system behavior and manage unexpected traffic fluctuations [22, 20, 23].

Industry solutions from major cloud vendors like Amazon and Google typically follow reactive scaling based on user-defined thresholds, with recent additions of predictive capabilities exploiting historical data for automatic adaptation [24, 25, 26].

Going towards global scaling, SmartHPA [27] is a Horizontal Pod Autoscaler for Kubernetes that adapts according to the resources available to the infrastructure. SmartHPA uses decentralised autoscaling under resource-rich infrastructures and a hierarchical approach under resource limitations so that the auto-scaler allocates and deallocates microservice replicas based on their relative load. While the hierarchical approach considers some aspects of the global state of the system (e.g., microservice replicas vs load), it does not perform a coordinated scaling of the architecture, as found in global scaling.

Global Scaling. Global scaling involves coordinating the scaling of multiple interacting microservices. Previous work in this area includes decidability results for optimal deployment of microservices [2, 3]. Other approaches, such as those proposed by [11, 28], rely on performance models, but suffer from limitations, e.g., delayed system capacity assessments and restrictions to specific architectures.

Recent studies highlight the potential of machine learning techniques combined with performance-aware approaches in improving microservice autoscaling efficiency. For example, GRAF uses a graph neural network to proactively allocate resources while minimizing CPU usage and meeting latency requirements, outperforming traditional autoscalers in resource savings and latency convergence [29, 30]. Similarly, MS-RA, a self-adaptive, requirements-driven solution, shows superior performance compared to Kubernetes' Horizontal Pod Autoscaler, achieving good performance with fewer resources [31]. The Polaris framework introduces a performance-aware autoscaler that uses high-level latency requirements, showing advantages over low-level CPU-based approaches [32].

Other notable contributions include Showar et al. [33], who proposed an efficient scheduling framework to optimise resource allocation for microservices, and Burstaware predictive autoscaling, which leverages burst patterns

in workloads to ensure high performance during demand spikes [34]. PB-Scaler addresses bottlenecks by adjusting resource allocations in real-time, preventing performance degradation due to resource constraints [35].

In conclusion, the combination of proactive and reactive global scaling approaches, along with advanced prediction techniques, can significantly enhance the scalability and efficiency of microservice architectures.

3. Background

3.1. Real-Time ABS

Abstract Behavioral Specification (ABS) [4] is a modelling executable language suitable for designing, verifying and evaluating concurrent and distributed systems. It is an actor-based object-oriented specification language (a process algebra) offering algebraic user-defined data types, side effect-free functions and immutable data. ABS objects are organized into Concurrent Object Groups (COGs), representing software components/services, and communicate via asynchronous method calls, i.e., *o!m()*. The ABS toolchain makes it possible to write ABS algebraic models by conveniently using a programming language syntax and executing them via, e.g., the Erlang backend.

Timed ABS is an extension to the ABS core language that introduces a notion of *abstract discrete time*, expressing the amount of *time units* elapsed since the system start. Such an extension makes it possible to evaluate time-related behaviour of distributed systems. Timed ABS has also *probabilistic* features that allow modelers to create uniform distributions, e.g., the average number of email attachments, as we will see in our running example. To represent VMs, Timed ABS introduces the notion of Deployment Component (DC) as a *location* where a COG can be deployed. As VMs, ABS DCs are associated with several kinds of resources, expressed via a dedicated annotation. In particular, virtual CPU speed is represented in ABS by the DC *speed*: it models the amount of *computational resource* per time unit a DC can supply to the hosted COGs. This resource is consumed by ABS instructions that are marked with the *Cost* tag, e.g., *[Cost: 30] skip*. Instructions tagged with a cost consume the hosting DC computational resource still available for the current time unit (the instruction above consumes 30 speed units): if not enough computational resource is left in the current time unit, then the instruction terminates its execution in the next one.

3.2. Automated Deployment of Microservices

In [2, 3], Bravetti et al. formalised component-based software systems and the problem of their automated deployment as the synthesis of deployment orchestrations (which allocate instances of software components on VMs) to reach a given target system configuration. In particular, the deployment life-cycle of each component type is formalised as a finite-state automaton, whose states denote a deployment stage. Each state corresponds to a set of *provided ports* (operations exposed by a component that other components can use) and a set of *required ports* (operations of other components needed by a component to work at that deployment stage). More specifically, Bravetti et al. [2, 3] consider the case of microservices, components whose deployment life cycle consists of two phases: (*i*) creation, which entails the *mandatory* establishment of initial connections, via so-called *strongly required ports*, with other available microservices, and (*ii*) binding/unbinding, which corresponds to the establishment of *optional* connections, specified as so-called *weakly required ports*, to other available microservices. The two phases make it possible to manage circular dependencies among microservices. The notions of strongly and weakly required ports are present also in state-of-the-art deployment technologies, e.g., Docker Compose [36]. In addition, Bravetti et al. [2, 3] consider resource/cost-aware deployments by modelling also memory and number of virtual CPU cores. In particular, the authors enrich both microservice specifications and VM descriptions of the resources they, respectively, need and supply.

A microservice *deployment orchestration* is a program in an *orchestration language* that includes primitives for (*i*) creating/removing a microservice together with its strongly required bindings and (*ii*) adding/removing weak-required bindings among microservices. Given an initial microservice system, a set of available VMs, and a new target system configuration (corresponding to the set of microservices to be deployed), the *optimal deployment problem* looks for the deployment orchestration that (*a*) satisfies core and memory requirements, (*b*) leads to a new system configuration where target microservices are deployed, and (*c*) chooses the solution optimising resource usage, if more than one is available. As an example of objective function to optimise, the reader can consider *cost minimisation*, i.e., select among all possible deployment orchestrations the one which minimises the sum of the cost-per-hour of the VMs hosting deployed microservices.

While Di Cosmo et al. [37] proved that the optimal deployment problem is undecidable when components have arbitrary deployment life-cycles, Bravetti

et al. showed that the latter becomes decidable when considering the simplified life-cycle of microservices described above, consisting of *creation* and *binding/unbinding* phases [2, 3]. The authors presented a constraint-solving algorithm whose result is the new system configuration, i.e., the microservices to be deployed, their distribution over the VMs, and the bindings to be established among their strong/weak required and provided ports.

3.3. SmartDeployer

SmartDeployer is executed at ABS compile time: it statically solves the optimal deployment problem described at the end of Section 3.2, i.e., synthesis of deployment orchestrations that reach a given target system configuration. In particular, it exploits the constraint solver Zephyrus2 [38] to solve the deployment problem. SmartDeployer takes its input from dedicated ABS annotations, included in the compiled ABS program, and produces its output as ABS code — synthesised (un)deployment orchestration — which is added to the initial annotated ABS program. The JSON-based ABS annotations from which SmartDeployer extracts its input are:

- *[SmartDeployCost : JSONstring]* class annotation. It is bound to an ABS class representing a given microservice type. It describes functional dependencies (provided and weak/strong required ports) and resources (e.g., number of cores) a microservice needs.
- *[SmartDeployCloudProvider : JSONstring]* global annotation. It defines DCs properties (e.g., *Cores, Memory, Speed, StartupTime*) and cost-per-hour created in the synthesized orchestration execution.
- *[SmartDeploy : JSONstring]* global annotation. It describes the desired properties and constraints of the deployment orchestration.

SmartDeployer produces as output the desired *(un)deployment orchestration*: a ABS program, injected in the initial annotated one, containing the set of *orchestration language* instructions (expressed as ABS code). The execution of the newly synthesised orchestration causes the system to reach a deployment configuration with the desired properties.

3.4. The Email Message Analysis Pipeline

In Fig. 2, we show the microservice architecture we use in this work: the Email Message Analysis Pipeline [2, 3]. The architecture includes 12

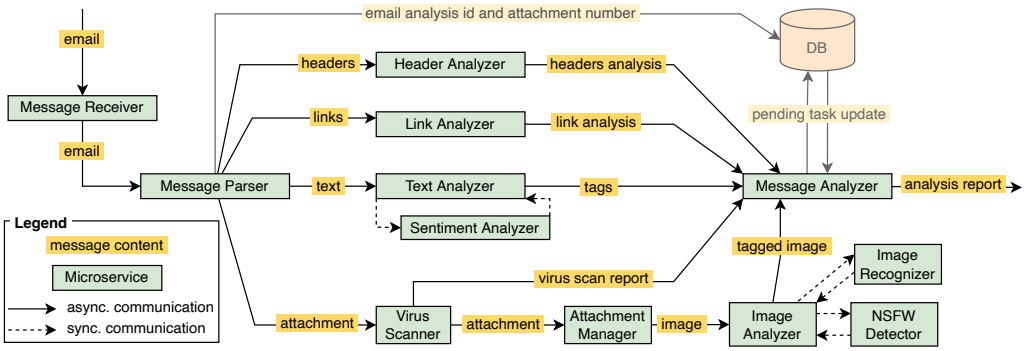


Figure 2: Microservice Architecture of the Email Message Analysis Pipeline.

types of microservices, each equipped with its dedicated load balancer. Each load balancer distributes inbound requests among the set of microservice instances, whose number can change at runtime. We can logically partition our microservice application in four pipelines, each dedicated to the analysis or different parts of an email, namely its headers, links, text, and attachments. Messages enter the system through the *MessageReceiver*, which forwards them to the *MessageParser*. This microservice, in turn, extracts data from the email and routes them to the proper pipeline. Once each email component has been processed asynchronously (each taking its specific processing time), the *MessageAnalyzer* aggregates the outputs of each pipeline corresponding to an email and produces a report for that email.

Before illustrating, in the next section, how one can apply to this example our approach for the automated deployment and scaling of microservice applications (cf. Section 3.2), we present our representation of cloud resources.

We consider virtual CPU cores both for machines (providing them) and for microservices (requiring them). In particular, here, we assume microservices to be deployed on Amazon EC2 VMs of type *large*, *xlarge*, *2xlarge*, and *4xlarge*, each respectively providing 2, 4, 8, and 16 virtual CPU cores (called vCPUs in Amazon EC2). Notice that we model computational resources supplied by VMs (and required by microservices) using *virtual* cores with some speed fixed by the Cloud provider.

4. Framework

In this Section, we will present the fundamental elements our global scaling approach relies on: (*i*) Multiplicative Factor (MF) and Max Computa-

tional Load (MCL), which are properties related to microservice types; (*ii*) the mathematical procedure to compute system target configurations (based on microservice type properties) targeted at handling a time-varying workload; and (*iii*) the automatic generation of *timed* deployment orchestrations via our *Timed SmartDeployer*, an extension to SmartDeployer.

4.1. Microservice MF and MCL

The MF of a microservice type is determined by the role it plays in the architecture, e.g., in our running example, by the email part it receives. For example, assuming an email has 2 attachments on average, the *Virus Scanner* microservice receives, for each email entering the system (i.e., request to the initial *Message Receiver* microservice), a mean of 2 requests. Thus, microservice MF is determined by the established *functional dependence between requests* to such microservice type and requests entering the system.

Therefore, concerning our running example, we base the calculation of the MF of its microservice types, on the following estimation of the structure of emails entering the system. On average: (*i*) a single header; (*ii*) a set of links (treated as a whole); (*iii*) a text body split into $\mathsf{N_{blocks}} = 2.5$ text blocks; and (*iv*) $\mathsf{N_{attach}} = 2$ attachments (individually sent to the attachment sub-pipeline), each having average size of $\mathsf{size_{attach}} = 7\mathsf{MB}$ and containing a virus with probability $\mathsf{P_V} = 0.25$.

Given the emails average structure, MFs are calculated as follows. *Header Analyser, Link Analyser* and *TextAnalyser* have $\mathsf{MF} = 1$ since emails have a single header, a set of links treated as a whole and a single text body. As text blocks and attachments are individually sent, each one generates a request to *Sentiment Analyser* and *Virus Scanner*, therefore they have $\mathsf{MF} = \mathsf{N_{blocks}}$ and $\mathsf{MF} = \mathsf{N_{attach}}$, respectively. The MF of microservices following the *Virus Scanner* is represented by the number of virus-free attachments, computed as $\mathsf{MF} = \mathsf{N_{attach}} \cdot (1 - \mathsf{P_V})$. Finally, the MF of the *Message Analyser* is the sum of the email parts (1 header, 1 set of links, 1 text body and $\mathsf{N_{attach}}$ attachments).

The MCL of a microservice type is the maximum amount of requests an instance of that type can handle in a second. It is computed as follows:

$$\mathsf{MCL} = 1/(\tfrac{\mathsf{size_{req}}}{\mathsf{data\_rate}} + \mathsf{pf})$$

where: (*i*) $\mathsf{size_{req}}$ is the average request size of microservices in MB; (*ii*) $\mathsf{data\_rate}$ is the microservice rate in MB/s to manage requests, determined accounting for microservice required cores (taken from server data in [39]); and (*iii*) pf is a penalty factor expressing additional time microservices

need to manage requests, e.g., those performing CPU-intensive tasks. We compute microservice $\mathsf{size}_{\mathsf{req}}$ as follows. For microservices handling attachments, but *Message Analyser*, we have: $\mathsf{size}_{\mathsf{req}} = \mathsf{N}_{\mathsf{attach\_per\_req}} \cdot \mathsf{size}_{\mathsf{attach}}$ where $\mathsf{N}_{\mathsf{attach\_per\_req}} = \mathsf{N}_{\mathsf{attach}}$ for microservices receiving entire emails, while, for the others, $\mathsf{N}_{\mathsf{attach\_per\_req}} = 1$. For *Header Analyser, Link Analyser* and *Text Analyser*, we consider $\mathsf{size}_{\mathsf{req}}$ to be negligible, thus (since $\mathsf{pf} = 0$) their MCLs are infinite. Concerning *Message Analyser* request size, we compute the average size of the MF requests an email entering the system generates (since we consider only attachments to have a non-negligible size), i.e.

$$\mathsf{size}_{\mathsf{req\_MA}} = \frac{\mathsf{N}_{\mathsf{attach}} \cdot (1 - \mathsf{P}_{\mathsf{V}}) \cdot \mathsf{size}_{\mathsf{attach}}}{\mathsf{MF}}.$$

MCL and MF microservice type are important properties, since they are used to calculate the minimum instance number of that type to guarantee an overall system MCL $\mathsf{sys\_MCL}$. Formally, be $\lceil x \rceil$ the ceiling function taking as input a real number and returns the least integer greater than/equal to x,

$$\mathsf{N}_{\mathsf{instances}} = \left\lceil \frac{\mathsf{sys\_MCL} \cdot \mathsf{MF}}{\mathsf{MCL}} \right\rceil$$

We now describe how we model microservice MF and MCL in our ABS models. The MF is implicitly modelled in the ABS code by the method call sequence required to analyse an email (see Figure 2). The MCL is explicitly modelled via the *Cost* tag (see Section 3.1). Let us consider an example from our ABS models: we consider an ABS time unit to be $1/30$ s and each VM to supply 5 $\mathsf{core\_speed}$. According to the calculation above, in case of attachments of 7 MB, it turns out the *Image Recognizer* has a MCL of 91 req/s. Since it requires 6 cores, we obtain the MCL of 91 req/s as follows:

Listing 1: Image Recognizer MCL

```
1 class ImageRecognizer() implements ImageRecognizerInterface {
2   Int mcl = 91;
3   String recImage(ImageRecognizer_LoadBalancerInterface balancer){
4     [Cost: 5 * 6 * 30 / mcl] balancer!removeMessage();
5     Int category = random(9);
6     return "Category Recognized: " + toString(category);
7   }
8 }
```

where `recImage` is executed at each request. The *Cost* tag above causes each request to consume $\mathsf{core\_speed} \cdot \mathsf{cores\_required} \cdot 30/\mathsf{MCL}$ computational resources at each time unit. Thus, since $\mathsf{MCL}/30$ is the service MCL expressed in requests per time unit, this realizes the desired service MCL.

Microservice	B	Δ1	Δ2	Δ3	Δ4	Microservice	B	Δ1	Δ2	Δ3	Δ4
Message Receiver	1	+1	+0	+1	+1	**Virus Scanner**	1	+1	+2	+1	+2
Message Parser	1	+1	+0	+1	+1	**Attachment Manager**	1	+0	+1	+0	+1
Header Analyser	1	+0	+0	+0	+0	**Image Analyser**	1	+0	+1	+0	+1
Link Analyser	1	+0	+0	+0	+0	**NSFW Detector**	1	+1	+2	+1	+2
Text Analyser	1	+0	+0	+0	+0	**Image Recognizer**	1	+1	+2	+1	+2
Sentiment Analyser	2	+1	+3	+2	+2	**Message Analyser**	1	+1	+2	+1	+2

Table 1: Base **B** (60 $\frac{emails}{sec}$) and incremental Δ configurations.

Scale 1 (+60 $\frac{emails}{sec}$)	**Scale 2 (+150 $\frac{emails}{sec}$)**	**Scale 3 (+240 $\frac{emails}{sec}$)**	**Scale 4 (+330 $\frac{emails}{sec}$)**
Δ1	Δ1 + Δ2	Δ1 + Δ2 + Δ3	Δ1 + Δ2 + Δ3 + Δ4

Table 2: Incremental Scale configurations.

4.2. Calculation of Scaling Configurations

We start with a base system configuration **B**, which guarantees a system MCL of 60 emails/sec. In Table 1, we present the number of instances for each microservice type, calculated according to the formula in Section 4.1. We also consider four incremental configurations, each adding a number of instances to each microservice type, see Table 1. Those incremental configurations are used as target configurations for automatic (un)deployment orchestration synthesis to perform run-time architecture-level reconfiguration. As shown in Table 2, Δ configurations are used, in turn, to build (summing them up element-wise as arrays) the incremental configurations Scale1, Scale2, Scale3 and Scale4 that guarantee an additional system MCLs (see Table 1).

The reason for not considering our Scales as monolithic blocks and defining them as combinations of the Δ incremental configurations is the following. Let us suppose the system is, e.g., in a **B** + Scale1 configuration and the increase in incoming workload requires the deployment of Scale2 and the undeployment of Scale1. Without Δ configurations, we would need to perform an undeployment of Scale1 followed by a deployment of Scale2. With Δ configurations, instead, we can simply additionally deploy Δ2.

For each microservice type, the number of additional instances considered in Tables 1 and 2 for the Scale configurations is calculated as follows. Given the additional system MCL, the number $\mathsf{N}_{\mathsf{deployed}}$ of already deployed instances of that microservice type, its MF and MCL, we have:

$$\mathsf{N}_{\mathsf{instances}} = \left\lceil \frac{(\mathsf{base\_MCL} + \mathsf{additional\_MCL}) \cdot \mathsf{MF}}{\mathsf{MCL}} - \mathsf{N}_{\mathsf{deployed}} \right\rceil$$

4.3. Timed SmartDeployer

The original SmartDeployer implicitly handles time aspects by simply copying Deployment Components (DC) properties from annotations, causing static assignments of speed and startup_time to each DC instance. The first causes microservices, deployed in a DC with unused cores, to unrealistically proceed faster: as if they could exploit the computational power of unused cores. The second causes the overall startup time to be the *sum* of that of individual DCs (since in the orchestrations DCs are sequentially created).

To solve these problems, we introduce Timed SmartDeployer, an extension to SmartDeployer [2, 3], synthesising *timed* deployment orchestrations. They additionally encompass dynamic management of overall DC *speed* and *startup time*. In particular, the solution to the speed problem is to dynamically evaluate, during orchestration, the number of cores actually being used, and adjust speed to: speed - core_speed $\cdot$ unused_cores. The solution to the startup_time problem is to dynamically set such a time to the *maximum* of DCs startup time. The above is realised by automatically synthesizing orchestrations, whose language additionally includes (w.r.t. SmartDeployer) two primitives *explicitly* managing time aspects in ABS: *decrementResources* to decrement speed and *duration* to set the overall the startup time of DCs.

Solving the speed problem is fundamental: such dynamic management based on cores actually being used, guarantees a microservice will always access the same amount of VM speed, no matter where it is deployed. Recall the code presented in List. 1, *ImageRecognizer* will always have $5 \cdot 6$ (speed_per_core $\cdot$ cores_required) maximum amount of speed.

5. Scaling Algorithms

This section explores the dichotomy of scaling strategies, namely local and global adaptation. Local adaptation enacts scaling actions at microservice level, while global adaptation scales the system as a whole based on the overall workload and request functional dependencies.

Central to our scaling framework is the concept of MCL, a critical component in both local and global contexts. MCL serves as the fundamental metric for triggering scaling actions, ensuring optimal resource allocation and system stability. We also introduce the concept of proactive scaling, a significant evolution from traditional reactive scaling. Proactive scaling uses advanced analytics to anticipate future workload demands, enabling more

efficient resource management than the traditional reactive approach, which scales resources based on current demand.

Finally, we present a novel proactive-reactive scaling algorithm that seamlessly integrates both reactive and proactive methods. This algorithm improves accuracy and responsiveness by effectively combining real-time demand analysis with forward-looking forecasting. Through extensive discussion and analysis, we show that global adaptation in its reactive-proactive form, is the most effective and accurate scaling strategy.

As it follows, we present the characterisation of the MCL-based scaling logic at the local and global levels. Then, we introduce the difference between the reactive and proactive approaches, concentrating on the latter and presenting a concrete implementation of a proactive system based on data analytics. While proactiveness increases the performance of global scaling, the approach is prone to outliers (e.g., sudden spikes of traffic). We mitigate the problem by presenting a solution to mix reactive and proactive modalities.

5.1. Local Scaling Algorithm

In the local scaling adaptation algorithm, following the Kubernetes approach [1], each microservice (type) has a dedicated monitor, and it is locally replicated by creating new instances every time scaling needs are detected. The monitor code below works as follows. We use a scaling condition on monitored inbound workload involving two constants called K, to leave a margin under the guaranteed service MCL and k to prevent sequences of scale up and down. The algorithm first applies the above scaling conditions, with the constant mcl being the microservice MCL and the variable deplInst the number of deployed instances (initially set to baseInstN, i.e., the number of instances deployed in the **B** configuration, see Table 1), and is updated in case of scaling needs. Then it computes the minimum number of microservice instances needed to handle the incoming workload as $\lceil(\mathsf{tw}+\mathsf{K})/\mathsf{MCL}\rceil$, with tw being the inbound workload. Finally, it (un)deploys instances to reach such a number (configInst below). If scale down occurs, the system keeps installed at least baseInstN instances.

```
1  if(tw - (mcl*deplInst-kbig) > k || (mcl*deplInst-kbig) - tw > k) {
2    Int configInst = ceil(float((tw + kbig)/ mcl));
3    if(configInst > deplInst) {
4      sw!deploy(configInst - deplInst);
5    }
6    else if(configInst < deplInst && deplInst >= baseInstN) {
7      sw!undeploy(deplInst - configInst);
```

```
8    }
9    deplInst = configInst;
10 }
```

5.2. Global Scaling Algorithm

Concerning global scaling adaptation, we have a single monitor that periodically executes the global scaling algorithm[1]. Here, `scaler` is an object that implements the methods `computeConf` and `scale`, presented afterwards. In the code below tw indicates the target workload.

```
1  if(tw - (mcl-kbig()) > k() || (mcl-kbig()) - tw > k()) {
2    List<Int> target_config = scaler.computeConf(tw);
3    scaler.scale(target_config);
4  }
```

The `computeConf` method below aims to compute the system configuration to cope with the target workload (i.e., `load`) passed as input. Such configuration is expressed in the form of a `List` where index i represents Δi and the *i-th* element is the number of Δi applications.

```
1  List<Int> computeConf(Rat load) {
2    List<Int> confDeltas = this.createEmpty(nScales);
3    List<Int> conf = baseConf;
4    mcl = this.mcl(conf);
5    Bool confFound = (mcl - kbig()) - load  >= 0;
6    while(!confFound) {
7      List<Int> candidateConf = baseConf;
8      Int i = -1;
9      while(i < nScales - 1 && !confFound) {
10       i = i + 1;
11       candidateConf = this.vSum(conf, nth(scaleComps,i));
12       mcl = this.mcl(candidateConf);
13       confFound = (mcl - kbig()) - load  >= 0;
14     }
15     conf = candidateConf;
16     confDeltas = this.addDeltas(i,confDeltas);
17   }
18   return confDeltas;
19 }
```

The code above uses constants `nScales`, representing the number of Scale

[1] `kbig()` and `k()` are respectively the K and k constants described above, implemented as constant functions, mimicking global variables.

configurations, and `scaleComps`: an array[2] of `nScales` elements (those in Table 2) that stores, in each position, an array representing a Scale configuration (i.e., specifying, for each microservice, the number of additional instances to be deployed). The code exploits the variable mcl, containing the current system MCL (assumed to be initially set to the **B** configuration MCL, see Table 1). At first, the code applies the above described scale up/down conditions. Then it loops, starting from the **B** configuration in variable `config` (an array that stores, for each microservice, the number of instances we currently consider), and selecting Scale configurations to add to `config`, until a configuration c is found such that its system MCL satisfies $\mathsf{mcl} - \mathsf{K} - \mathsf{target\_workload} \geq 0$. The system MCL of a configuration c is calculated with method `mcl`, which yields

$$\mathsf{min}_{1 \leq \mathsf{i} \leq \mathsf{length(conf)}} \ \mathsf{nth}(\mathsf{conf}, \mathsf{i}-1) \cdot \mathsf{MCL_i}/\mathsf{MF_i}$$

with $\mathsf{MCL_i}/\mathsf{MF_i}$ denoting those of the i-th microservice. The algorithm uses an external loop updating variables `conf` and `confDeltas` according to the incremental Scale selected by the internal loop: `confDeltas` is an array of `nScales` elements to keep track of the number of currently deployed Δ incremental configurations (initially empty, i.e., with all 0 values). Every time a Scale configuration is selected, `confDeltas` is updated by incrementing the amount of corresponding Δ configurations (see Table 2). The internal loop selects the first Scale configuration that, added to `conf`, yields a candidate configuration, whose system MCL satisfies the condition above. If no configuration is not found, it just selects the last (the biggest) Scale configuration, thus implementing the following invariant: if N Scale reconfigurations are applied and increasingly sorted by system MCL increment, they guarantee the reached system configuration is either **B** or $\mathbf{B} + (\mathsf{n} \cdot \mathsf{ScaleN}) + \mathsf{scale}$, for some $\mathsf{scale} \in \{\mathsf{Scale1}, \mathsf{Scale2}, \ldots, \mathsf{ScaleN}\}$ and $\mathsf{n} \geq 0$.

The invariant property guarantees that multiple deployments of the same Scale configuration are not allowed, except for ScaleN. This is because, the biggest configuration ScaleN should be devised such that the workload rarely yields to additional scaling needs. Even if a sequence of ScaleN occurs, the system would be sufficiently balanced. This is because, differently from smaller Scale configurations, ScaleN is assumed to add, at least, an instance for each microservice with finite MCL (as for Scale4 in our case study).

[2]The ABS instructions `nth(a,i)` and `length(a)` retrieve the i-th element and the length of the a array, respectively.

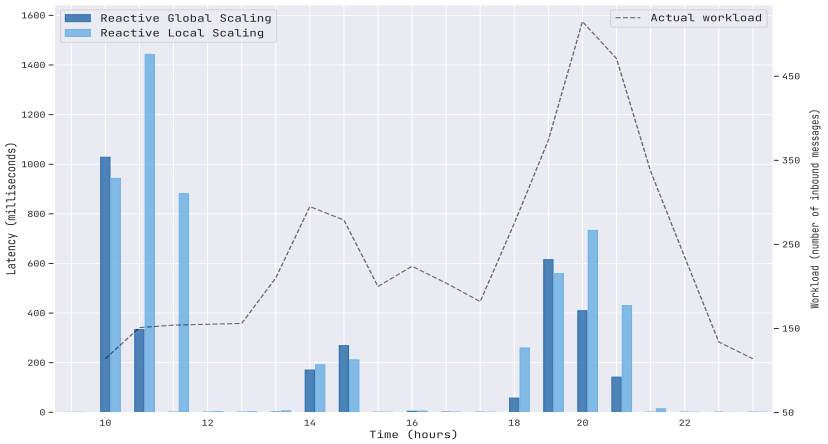
The `scale` method presented below enacts the scaling operations required to reach the system configuration passed as input.

```
1  Unit scale(List<Int> confDeltas) {
2    Int i = 0;
3    while(i < nScales) {
4      Int diff = nth(confDeltas,i) - nth(deplDeltas,i);
5      Rat num = abs(diff);
6      while(num > 0) {
7         if (diff > 0) {nth(orchDeltas,i)!deploy();}
8         else {nth(orchDeltas,i)!undeploy();}
9         num = num - 1;
10     }
11     i = i + 1;
12   }
13   deplDeltas = confDeltas;
14 }
```

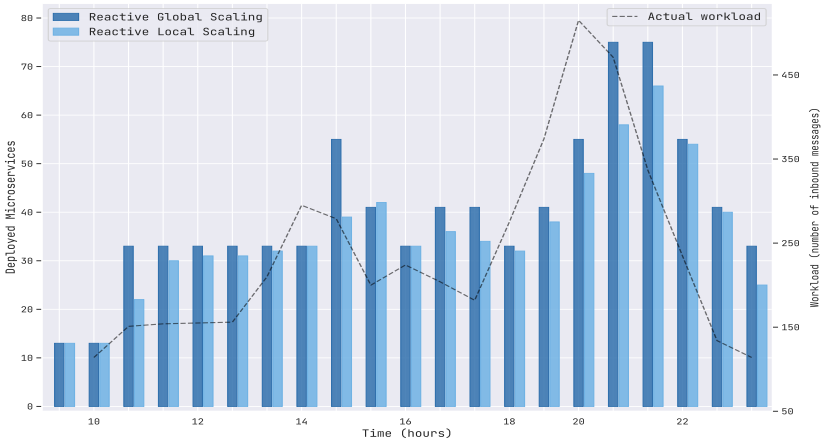
Given the target Δ configurations `confDeltas` to be reached and the current `deplDeltas` (an array with the same structure of `confDeltas`) ones, the `scale` method performs the difference between them so to find the Δ orchestrations that have to be (un)deployed. We use `(un)deploy` methods of the object in the position $i-1$ of the array `orchDeltas` to execute the orchestration of the i-th Δ configuration.

With the presented algorithm, we claim that we can go beyond the state-of-the-art scaling approaches, i.e., the local scaling [1]. To support our claim, we propose the following comparison, taking into account latency and the number of deployed instances on two real-world different datasets: part of an IMAPS traffic [7] (see Figs. 3a and 3b) and a load pattern inspired to that in [6] (see Figs. 4a and 4b), accounting for the fact that here email attachments are also considered. Concerning latency, from Figures 3a and 4a is clear the extent of improvement of our algorithm: while the local scaling approach struggles in adapting to traffic changes, ours easily restores performance. The reason for such performance difference is highlighted in Figures 3b and 4b: our approach, as soon as a peak in inbound workload is detected, deploys all instances needed to cope with such peaks. Local adaptation, instead, suffers from the domino effect: the number of microservice instances grows linearly over time until it reaches a stable situation, i.e., all instances to cope with such workload peak are installed, delaying the adaptation.

Focusing on Global Scaling. In the remainder of the section, we focus only on global scaling, developing our algorithms for proactive and proactive-reactive versions. We ignore proactive local scaling, since we empirically measured



(a) Latency



(b) Deployed microservices

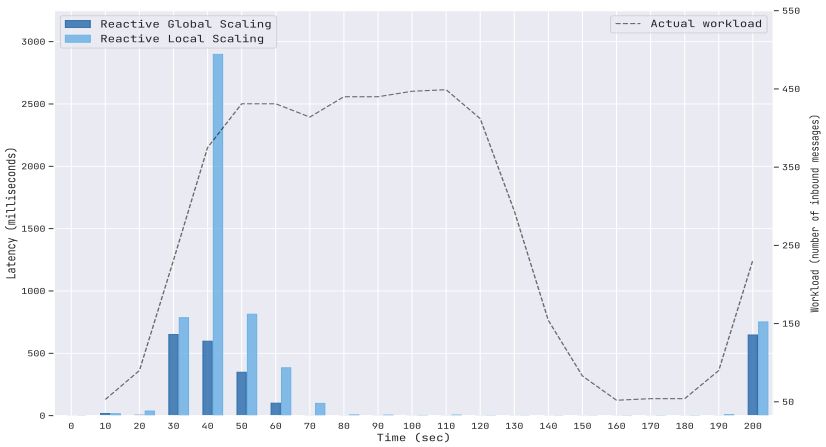
Figure 3: Comparing reactive local and global scaling on [7].

that even an "oracle" (a predictor with 100% accuracy) it cannot outperform the reactive version of global scaling.

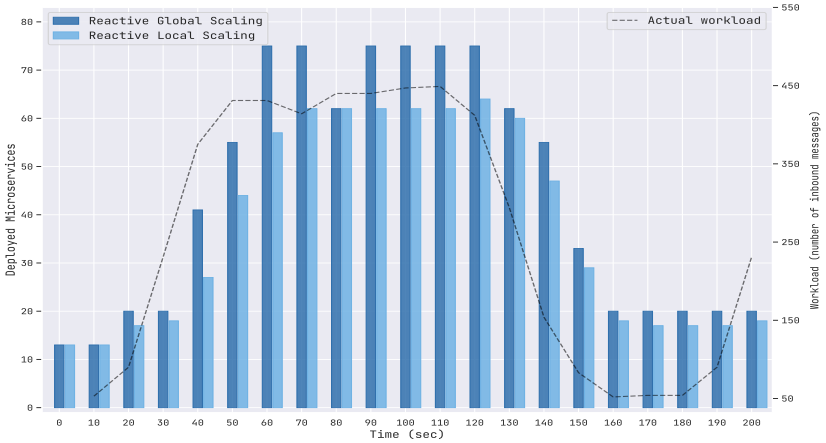
We present the quantitative comparison between the oracle local scaling algorithm and the reactive global one in Section 6.

5.3. Data Analytics for Global Scaling

We provide a general overview of the elements of data analytics [40] (DA) applied to the Enron dataset [5]. The dataset contains 517431 emails from 151 users distributed over a time window of about 10 years. Using data an-



(a) Latency



(b) Deployed microservices

Figure 4: Comparing reactive local and global scaling on [6].

alytics to predict the occurrence of an event in a given time unit, we aim to understand which variables influence the phenomenon (*descriptive DA*) and the most relevant attributes (*diagnostic DA*). In particular, we generate five new attributes: month, day, weekday, hour, and counter, the target attribute. Then, using these attributes, we build a dataset to train a model and infer new events (*predictive DA*). To do that, we use an off-the-shelf machine learning model: a MLP (Multi-Layer Perceptron) to explore nonlinear patterns and increase prediction performance, while containing complexity and resource usage. We categorise numerical variables with the one-hot encoding

technique, to prevent our MLP from attributing wrong semantics to these (e.g., month 12 is "greater than" month 1). Finally, we compute the error rate with the Mean Squared Error (MSE) loss function, and we optimise the model parameters with the Adaptive Moment Estimation (Adam).

To realise a predictive algorithm for global scaling (*prescriptive* and *proactive DA*), we use predictions (instead of monitor signals) as source to drive deployment decisions.

5.4. Proactive-Reactive Global Scaling

As empirically demonstrated in Section 6, predictors are weak against exceptional events. Therefore, we developed a proactive-reactive approach to mitigate the shortcomings of inaccurate predictions, resulting in the application of inappropriate scaling decisions. In particular, we introduce a new algorithm based on a weighted sum, which computes the weights related to the predicted and measured (via reactive signals of the monitor) workloads.

Our algorithm is based on using the MCL to cast the comparison as the capacity of the system to deal with a given workload, defined by its current scaling configuration. Hence, we have a way to detect both over- and under-estimations of proactive global scaling, given by the distance from the MCL (of the scaling configuration) induced by the actual traffic. We do not directly compare the estimated and actual number of inbound requests in a given time unit. The reason is that the dynamic interaction between message queues and scaling times makes it difficult to reliably estimate the accuracy of the predicted scaling configuration w.r.t. traffic fluctuations.

We consider statically-defined scores s_i for each architectural reconfiguration Δ_i, computed based on the increment in system MCL. For each Δ_i, we have a differential system MCL increment of: $\Delta MCL_1 = 60$ for Δ_1 and $\Delta MCL_i = 90$ for Δ_i with $2 \leq i \leq 4$. Given ΔMCL_i, we compute $s_i = \frac{\Delta MCL_i}{\sum_{j=1}^{4} \Delta MCL_j}$. Notice that this yields $\sum_{i=1}^{4} s_i = 1$.

We now describe the mixing procedure we devise. Differently from the previous algorithms, here we start by informally presenting the auxiliary functions together with their ABS implementation. For each time unit t, we proceed as follows. In step 1, as shown in the code below, we compute, for each index i, the difference $diff_i$ between the Δ_i, needed to cope with the predicted workload at $t-1$, and those for the monitored one at t (`lpc` and `ac` below, respectively).

```
1 List<Int> compute_diff(List<Int> lpc, List<Int> ac) {
2   List<Int> diff = list[];
```

```
3    Int i = 0;
4    while(i < length(lpc)) {
5      diff = appendright(diff, nth(lpc, i) - nth(ac, i));
6      i = i + 1;
7    }
8    return diff;
9  }
```

Then, in `compute_weight`, we compute $w \in [0,1]$ used to combine the predicted and measured workload. Since $|\mathit{diff}_i| > 1$ only happens in exceptional cases (e.g., $Scale_4$ is not enough), we compute $w = min\left(\sum_{i=1}^{4} s_i \cdot |\mathit{diff}_i|, 1\right)$. Here, `scores()` is a function returning the list of s_i scores.

```
1  Rat compute_weight(List<Int> lpc, List<Int> ac) {
2    List<Rat> devs = scores();
3    Rat curr_weight = 0;
4    List<Int> diffs = this.compute_diff(lpc, ac);
5    while(!isEmpty(lpc) && !isEmpty(ac)) {
6      curr_weight = curr_weight + abs(head(diffs) * head(devs));
7      lpc = tail(lpc);
8      ac = tail(lpc);
9      diffs = tail(diffs);
10     devs = tail(devs);
11   }
12   return min(curr_weight, 1);
13 }
```

We keep track of w values computed in the last 3 time units using function $h = \{(1, w_{t-2}), (2, w_{t-1}), (3, w_t)\}$, where w_t is the weight computed for the current time unit and w_{t-2}, w_{t-1} are the preceding ones. The pairs $(1, w_{t-2}), (2, w_{t-1})$ added only if the system was already running at those times. This function is modeled below by `store_weights`, where we add the inputed weight to `errors`, possibly removing the oldest one (line 3).

```
1  Unit store_weights(Rat curr_weight) {
2    errors = appendright(errors, abs(curr_weight));
3    if(length(errors) == memory()) errors = tail(errors);
4  }
```

In step 2, we compute the distance $\mathit{dist} = \frac{\sum_{(i,w)\in h} w \cdot i}{\sum_{(i,\_)\in h} i}$ of t, between the measured and predicted workloads (where $w \cdot i$ means that the most recent w is the most influential one). The closer the distance (computed in `compute_distance`) is to 1, the less accurate the prediction is.

```
1  Rat compute_distance() {
2    Rat dist = 0;
3    Int toDivide = 0;
```

```
4     foreach(e,i in errors) {
5       dist = dist + e * (i + 1);
6       toDivide = toDivide + (i + 1);
7     }
8     return dist / toDivide;
9   }
```

In step 3, we mix the predicted and measured workload (i.e., `mp` and `mt` below) via *dist*, to find the load the system has to cope with: $target\_load = (dist \cdot measured\_load) + ((1 - dist) \cdot pred\_load)$ (computed in the `mix` method).

```
1   Rat mix(Rat mt, Rat mp, List<Int> lpc, List<Int> ac) {
2     Rat curr_weight = this.compute_weight(lpc, ac);
3     this.store_weights(curr_weight);
4     Rat react_score = this.compute_distance();
5     Rat pred_score = 1 - react_score;
6     Rat target_scale = (react_score * mt) + (pred_score * mp);
7     return target_scale;
8   }
```

6. Evaluation

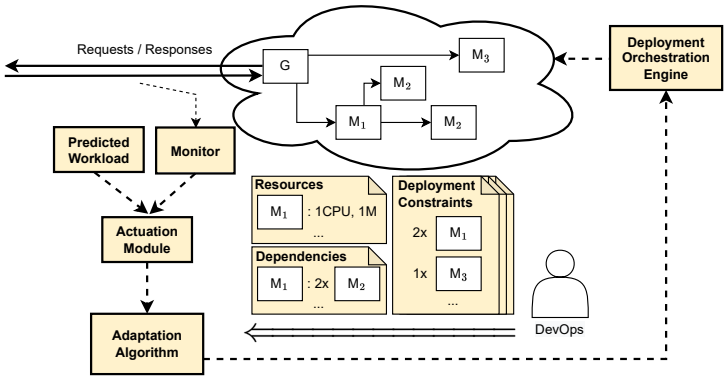


Figure 5: A platform for proactive-reactive architecture-level adaptation.

We model in ABS the platform presented in Fig. 5 (publicly available at [12]) to benchmark the performance of our scaling approach. The platform includes an external microservice system (labelled G, $\mathsf{M}_1, \mathsf{M}_2$, M_3) and internal elements (i.e., orange boxes) building the scaling approach used to scale the system. Since the platform sees microservices as instance parameters, we abstract from their behaviour. In Fig. 5, we distinguish three flows:

(i) $\rightarrow$ showing the inbound workload; (ii) $\dashrightarrow$ modeling the runtime execution of an adaptation process; and (iii) $\Leftarrow$ indicating the synthesis of deployment orchestrations. We now describe each element of the platform.

Deployment Orchestration Engine. This component takes as inputs deployment orchestrations and executes them to perform system reconfigurations. It is a loosely-coupled component, taken from existing solutions (it just needs to provide a programming interface for the application of orchestrations), e.g., Kubernetes [1]. In our setting, this component is represented by the Erlang backend (see Section 3.1), which is in charge of executing the simulation.

Adaptation Algorithm. The Adaptation Algorithm implements the strategy to compute the deployment orchestrations to apply, to cope with inbound workload. Such module needs two inputs to work. The first input, depicted with $\Leftarrow$, represents deployment orchestrations statically computed by Timed SmartDeployer (see Sections 3.3 and 4.3). These orchestrations are such that they satisfy the user (DevOps in Fig. 5) specifications, i.e., Resources, Dependencies and Deployment Constraints. The second input, represented by $\dashrightarrow$, is the workload the system has to support, after the adaptation process.

Monitor. The monitor tracks the traffic flowing on the architecture within a prefixed *time window* and checks the possible occurrence of a *workload deviation*, i.e., the difference between the monitored workload and the globally supported one. When such a condition occurs, the Monitor sends to the Actuation Module the amount of measured workload.

Predicted Workload and Actuation Module. The Predicted Workload is obtained using the ML-based workflow described in Section 5. Such workload is statically injected in the simulation exploiting a standard ABS data structure, i.e., arrays, and it is forwarded to the Actuation Module. The Actuation Module computes the amount of workload given as input, i.e., the *target workload*, to the Adaptation Algorithm. We distinguish among three modalities: (i) *reactive* mode, if the target workload is the one measured by the monitor (the predicted one is discarded); (ii) *proactive* mode, if the target workload is represented by predictions in the Predicted Workload (the measured one is discarded); and (iii) *proactive-reactive* mode, if the target workload is computed combining signals from the Monitor and Predicted Workload.

Concretely, we model the platform in Fig. 5 and the scaling approaches via ABS, compiling it into a system of Erlang programs to run the simulation.

The simulation receives three kinds of inputs, which are statically defined within a simulation run: *deployment orchestrations* (generated by Timed SmartDeployer), an *actual* and a *predicted workload*, both hard-coded in the simulation as arrays. To test our scaling approach, we consider an additional dataset, w.r.t. the ones in Section 5.2. We model the actual workload array taking data from the Enron dataset [5] (accounting for the fact that here email attachments are also considered). With the platform of Fig. 5, we aim to answer the research questions presented in the introduction.

6.1. Oracle and Reactive Local vs Reactive Global Scaling

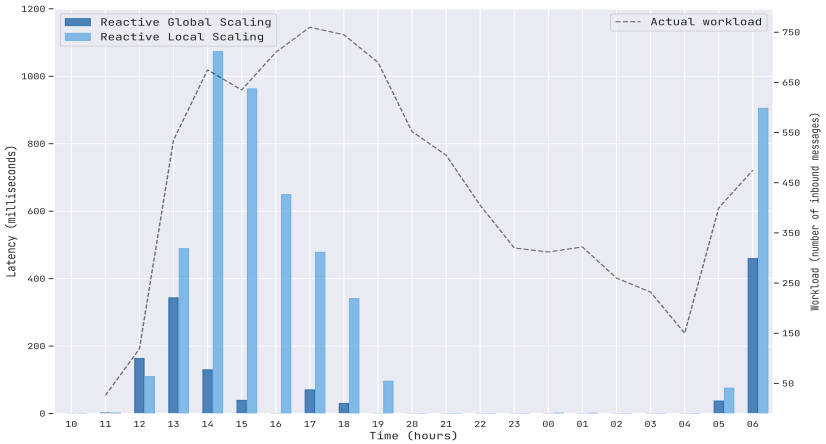
We start answering to the *RQ1* with the following comparison: *reactive local* and *reactive global* scaling. We first analyse performance in terms of latency: as can be seen in Fig. 6a, our scaling approach outperforms the mainstream one, restoring latency much faster. Comparing the number of deployed instances highlights the reasons why the global adaptation performs better. As shown in Fig. 6b, our approach reaches the target configuration, to cope with the inbound workload, faster than the local one. As expected, this increases the adaptation responsiveness to higher workloads. The local adaptation delay in reaching such configuration is caused by the domino effect. Hence, w.r.t. global adaptation, where microservices in the target configuration are deployed together, the number of instances grows slower.

To fully answer to *RQ1*, we endow the local scaling approach with *proactive* capabilities using an *oracle*. Despite local scaling knows in advance the exact amount of requests in each microservice, our approach still performs better. As can be seen in Fig. 7a, our approach has significantly better performance, always keeping latency under acceptable values. The reason is clear when we consider Fig. 7b, which take into account the number of deployed instances: despite proactivity, the local scaling still suffers from the domino effect. The reason is that proactivity, as can be seen, comparing Figs. 6b and 7b, just shifts the problem, without removing it.

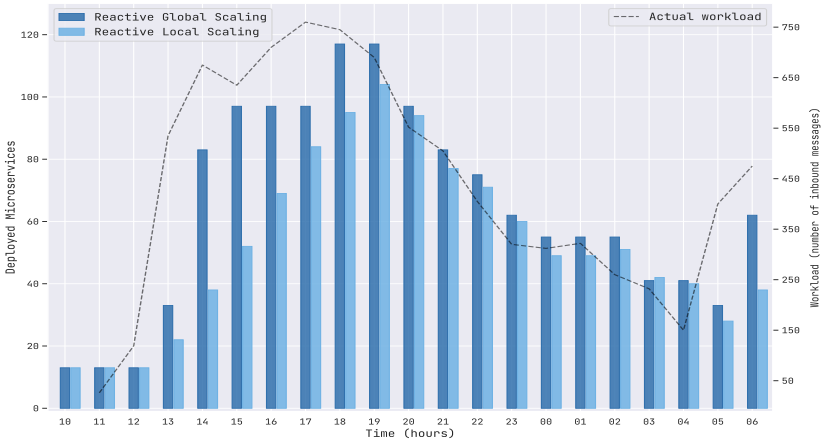
This comparison proves that exploiting functional dependencies brings significant advantages and avoids the drawbacks of the local scaling approach.

6.2. Proactive vs Reactive Global Scaling

To answer to *RQ2*, we start presenting the performance of our proactive global scaling. In this experiment, we focus on the evaluation of the same metrics as in the previous benchmark. As expected, the proactive global scaling not only performs better but, as can be seen, in Fig. 8a, it almost mimics



(a) Latency

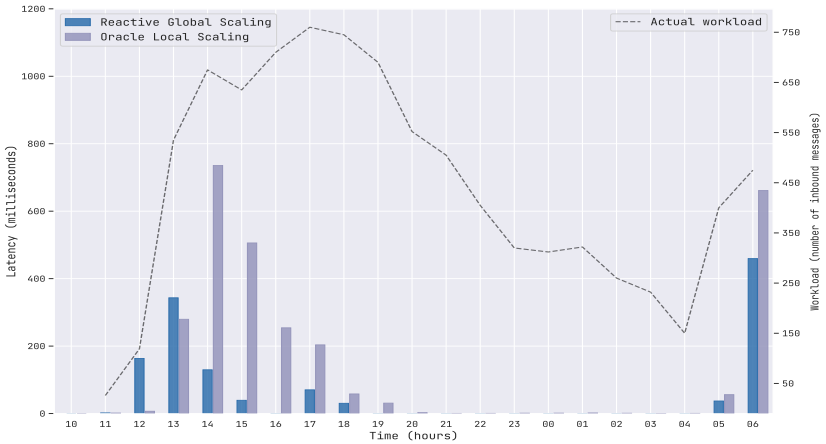


(b) Deployed microservices

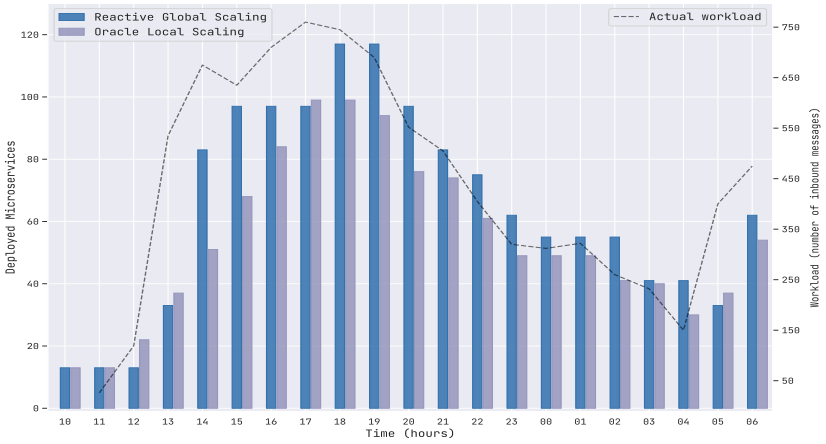
Figure 6: Comparison between local and global scaling both in reactive mode.

the performance of the global scaling algorithm endowed with the oracle (which has 0 latency throughout the entire experiment). Fig. 8b highlights the reason why proactive global scaling performs better w.r.t. the reactive one: the predictor makes it possible to anticipate scaling operations.

However, proactive scaling works as long as no unexpected events happen. In this experiment, we selectively pick outliers from the Enron dataset and run simulations using that flow of requests as input. To better understand the urgent need for a proactive-reactive approach, here, we consider a cost comparison instead of the amount of deployed microservices (although they



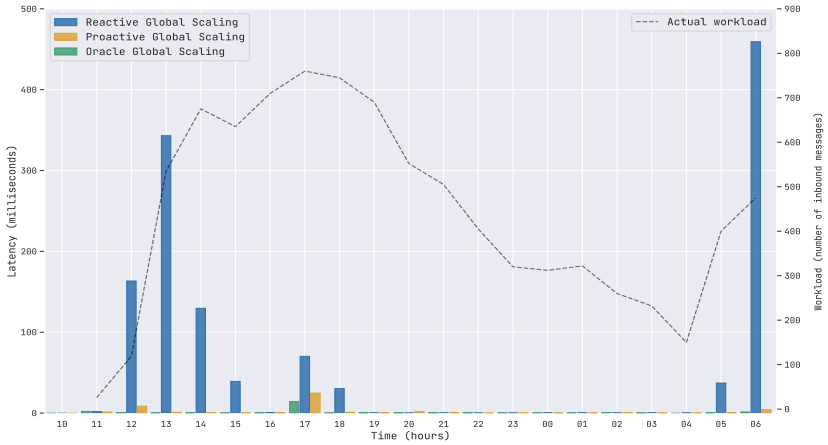
(a) Latency



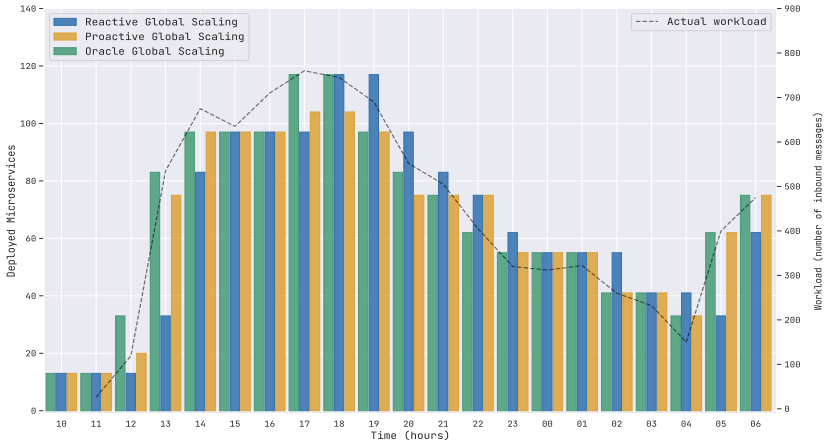
(b) Deployed microservices

Figure 7: Comparison between oracle local scaling and the reactive global one.

are strictly related). As can be seen in Fig. 9a, the proactive global scaling not only is more distant from the oracle one, but it even performs worse than the reactive global one. This happens because the proactive approach fully relies on predictions and, as can be seen from the red dashed line, they are not accurate. Wrong predictions not only worsen performance, but they also lead to money loss due to waste of resources (see Fig. 9b).



(a) Latency

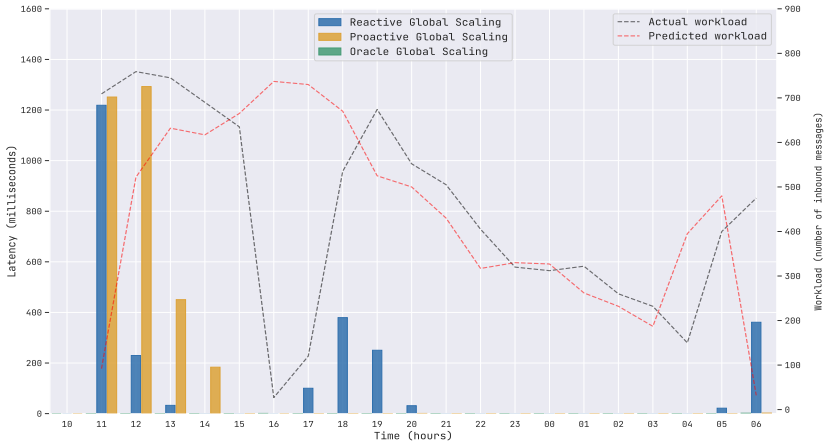


(b) Deployed microservices

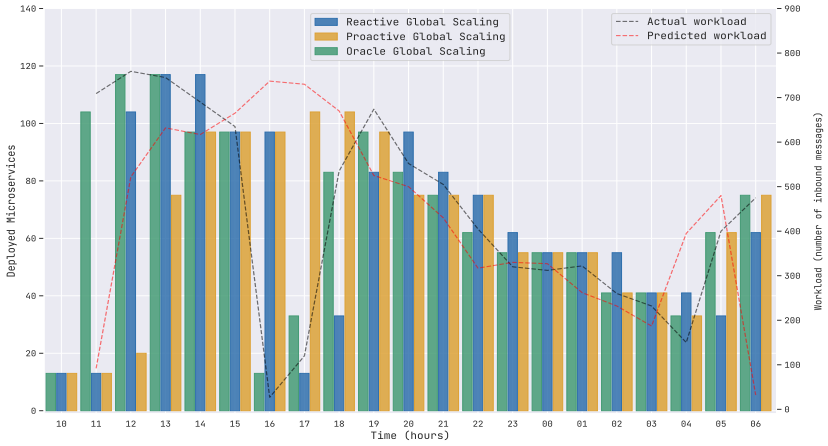
Figure 8: Comparison between reactive, proactive and oracle global scaling.

6.3. Proactive vs Proactive-Reactive Global Scaling

To provide an exhaustive answer to *RQ2*, we realise the following comparison. Here, we test the performance of our proactive-reactive global scaling on the outliers dataset. In particular, in Figs. 10a and 10b, we use the oracle global scaling as an upper bound for performance. As expected, the proactive-reactive algorithm immediately detects wrong predictions and applies the technique presented in Section 5.4: it can restore acceptable performance. Notice that, if unexpected events last over time, the proactive approach has no possibility of restoring performance since it fully relies on



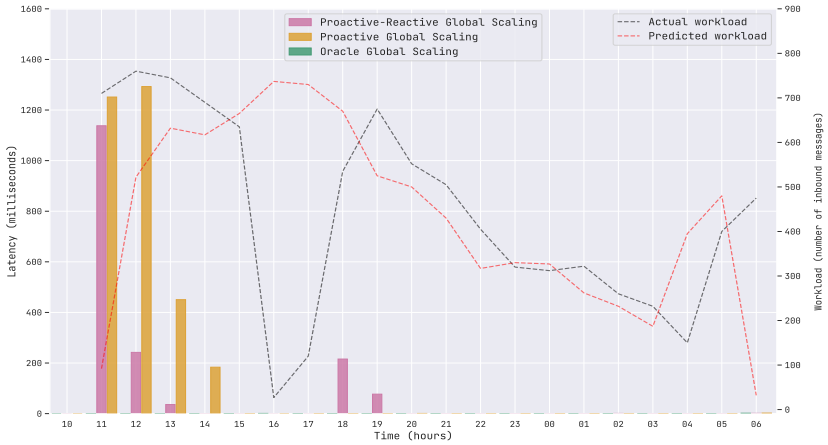
(a) Latency



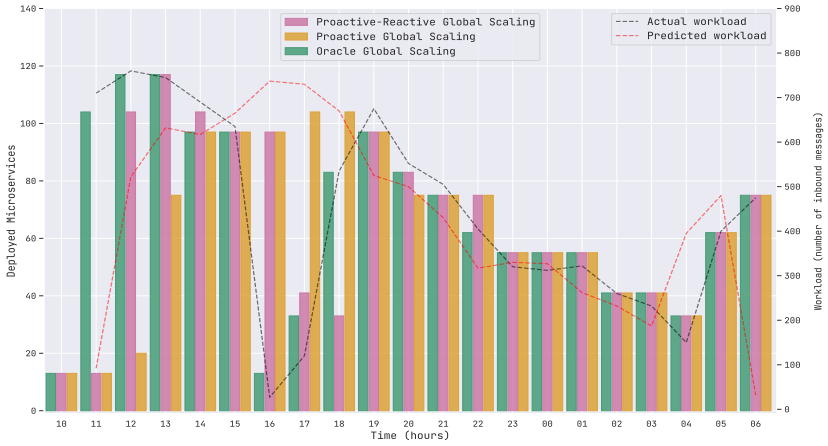
(b) Deployed Instances

Figure 9: Reactive, proactive and oracle global scaling comparison, on outliers dataset.

predictions, i.e., it uses prediction to make deployment decisions. Besides performance, our proactive-reactive approach prevents waste of resources: when predictions overestimate the inbound workload (see Fig. 10b between 15 and 18), the amount of deployed instances is adjusted, considering the actual need for computational resources. Thus, we can conclude that dealing with the complexities of a proactive-reactive approach for global scaling is worthwhile.



(a) Latency



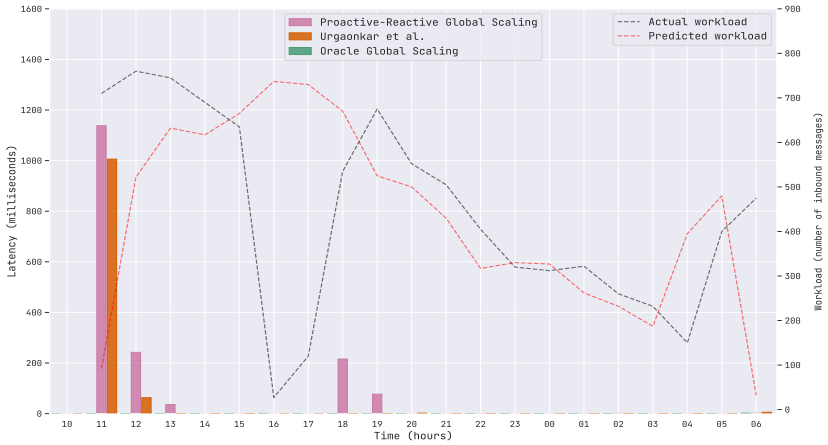
(b) Deployed microservices

Figure 10: Proactive-Reactive global scaling performance evaluation, on outliers dataset.

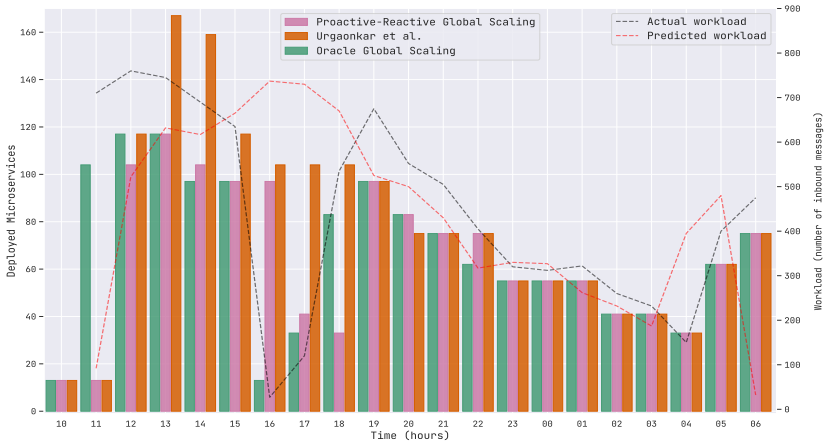
6.4. Alternative Hybridisation Technique

The last comparison we present aims to answer *RQ3*. Here, we evaluate the performance of our proactive-reactive algorithm w.r.t. the one presented in [11], considering latency and cost. Notice that, since, as we do, also in [11] scaling decisions are driven considering the number of requests, we were able to apply their mixing technique to our global scaling algorithm.

The latency comparison shows that both approaches guarantee the same Quality of Service. However, the approach of [11] hides a pitfall: waste of resources. As reported by the authors, they only consider positive errors to



(a) Latency



(b) Deployed microservices

Figure 11: Comparison with the Proactive-Reactive algorithm of [11], on outliers dataset.

correct underestimates of the predicted peak demand. Thus, they just focus on performance without taking into account resources/costs. As can be seen in Figure 11b between 13-19, our approach deploys a number of microservice instances always closer to the oracle one, w.r.t. the approach of [11]. To further highlights the importance of saving resources, we additionally compare the cost caused by deployment decisions induced by the two algorithms. As can be seen in Figure 12, in case of prediction overestimations, the algorithm of [11] produces a configurations much more expensive w.r.t. ours. Thus, we can conclude that our algorithm, not only considers underestima-

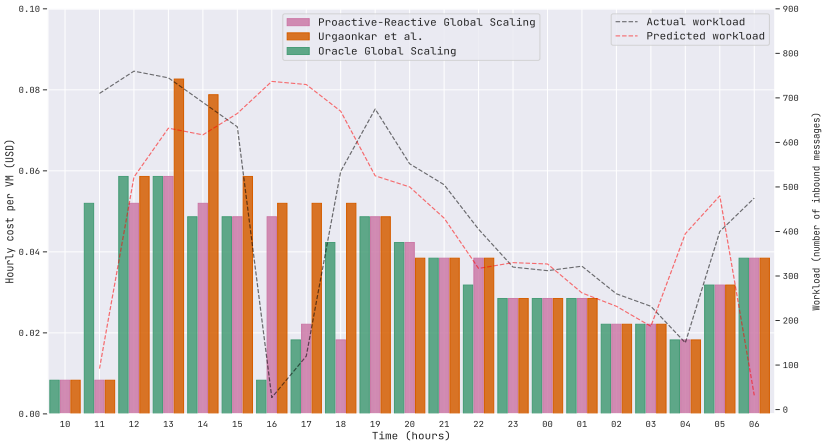


Figure 12: Cost comparison with [11], on outliers dataset.

tions of wrong predictions, but also preserves resources taking into account overestimations.

6.5. Threats to Validity

Our experiments show that the scaling approach we propose significantly outperforms the state-of-the-art and our proactive-reactive algorithm is capable of considering the performance and cost trade-off in the adaptation process. First, our global scaling approach relies on the knowledge of microservice properties, e.g., MCL, MF and resource requirements, which are usually not directly available. However, this information can be easily retrieved by performing online measurements of resource usage and request performance, using off-the-shelf tools, e.g., Prometheus [41]. Second, our global scaling approach has only been simulated in ABS and not tested in a real-world scenario. However, we believe ABS is a realistic simulation environment where the behaviour of distributed systems is precisely reproduced. Indeed, the behaviour of the system developed in [42], which implements an orchestrator for workload migration in the context of an industrial scenario, has been accurately reproduced in [43]. As can be seen in [43], the experimental results of the ABS implementation in [43] closely reproduce those of the real-world implementation [42], proving that ABS is capable of precisely modeling system behaviour.

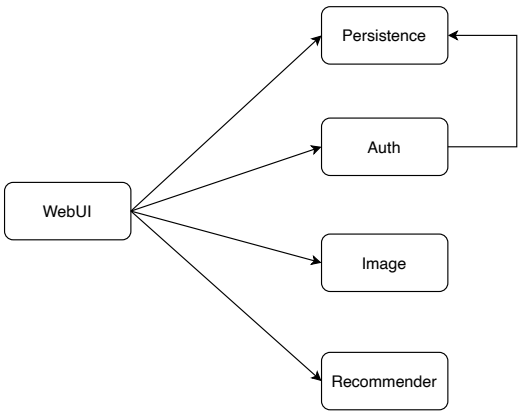


Figure 13: TeaStore architecture.

6.5.1. Application to TeaStore

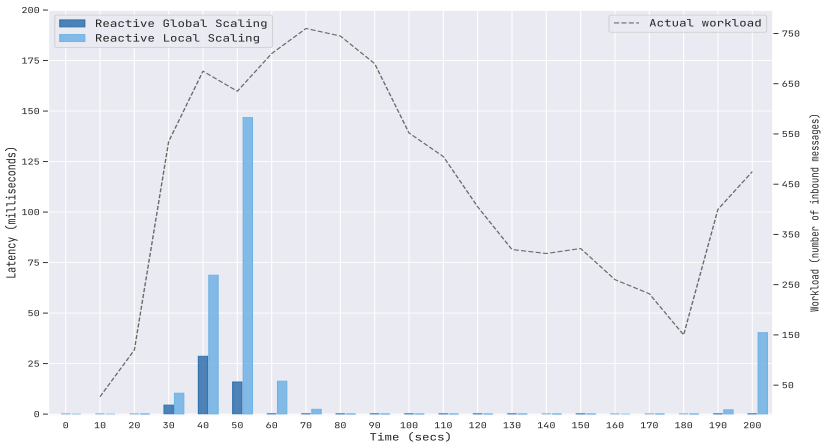
Our experiments are currently limited to evaluation of the performance of a single microservice application, i.e., the Email Pipeline Processing System. Since our approach relies on high-level characteristics of microservices composing the architecture, we do not believe such limitation poses a threat on the effectiveness of our approach and we deem it equally applicable to any kind of microservice architecture. In particular, we now consider the TeaStore microservice application [13, 14]. As can be seen in Figure 13 (taken from [13], where the Registry service is disregarded since it is not involved in scaling activities), it includes five primary services: WebUI provides the user interface; Auth verifies the user credentials and session data; Persistence interacts with the database; Recommender predicts the user preference for different products and recommends appropriate ones; and Image provides an image of products in different sizes.

To apply our proactive-reactive scaling algorithm to the TeaStore, we first compute, in Table 3, the set of Δ scaling configurations used to statically synthesize the deployment orchestrations, where MCL and MF values are taken from the literature [13, 14]. Notice that, $\mathbf{B}$ represents the base system configuration and guarantees a system MCL of 150 requests per second; Δ configurations, as done in Section 4.2, are combined to produce Scale1 and Scale2 configurations guaranteeing, a system MCL of +200 and +400 requests per second, respectively.

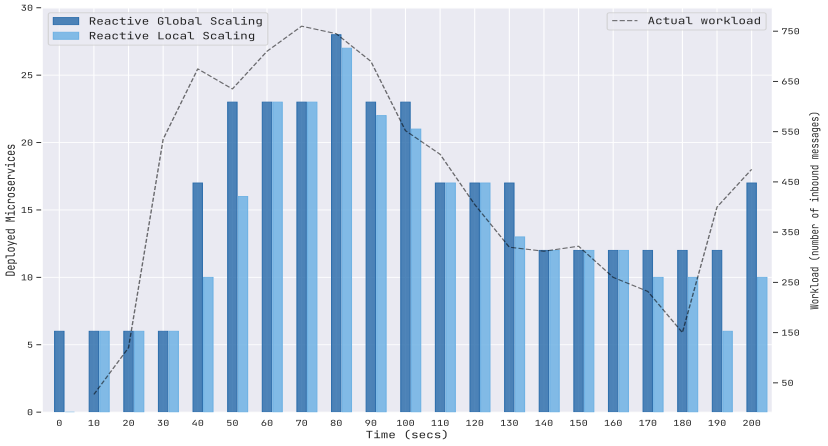
We first benchmark the performance of our reactive algorithm w.r.t. the reactive version of the mainstream approach, i.e., the local scaling. As can be

Microservice	**B**	$\Delta 1$	$\Delta 2$
WebUI	1	+2	+1
Persistence	1	+1	+1
Auth	1	+1	+1
Recommender	1	+0	+0
Image	2	+2	+2

Table 3: TeaStore incremental scaling configuration



(a) Latency



(b) Deployed microservices

Figure 14: Comparing reactive global and reactive local scaling, applied to the TeaStore.

seen in Figure 14a, our global scaling algorithm outperforms the mainstream one: we are able to restore acceptable performance faster. The reason why the local scaling performs worst can be seen in Figure 14b: whenever the workload grows, i.e., interval 40-90, the number of deployed instances grows linearly over time due to the domino effect.

To further show the extent of the improvement of our global scaling algorithm, we compare our reactive approach w.r.t. the local scaling endowed with an oracle. Figure 15a shows that our approach restores acceptable performance faster, since it does not suffer from the domino effect, which, in turn, affects the local scaling, as can be seen in Figure 15b (interval 30-80).

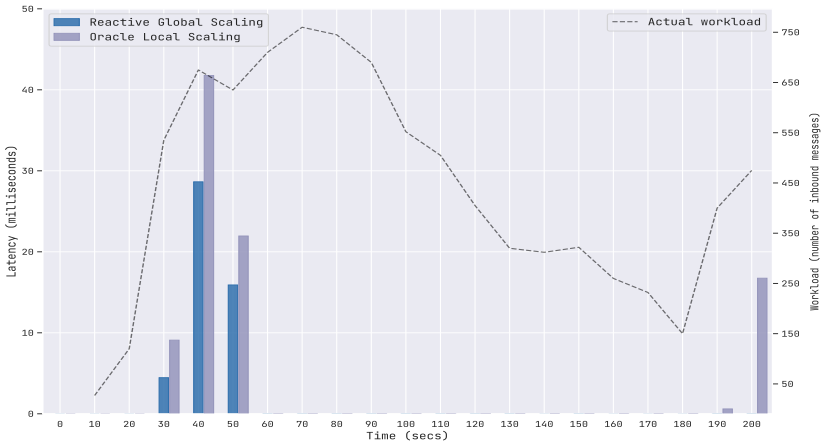
Our last benchmark shows how our proactive-reactive global scaling algorithm performs on the TeaStore system. As done in Section 6.3, we use outliers from the Enron dataset. As can be seen in Figures 16a and 16b, our approach performs better w.r.t. to the proactive version: in case predictions underestimate/overestimate the workload (interval 10-40), our proactive-reactive algorithm uses reactive signals from monitor to adjust prediction errors, reaching the same system configuration as the global scaling endowed with an oracle.

7. Conclusion and Future Work

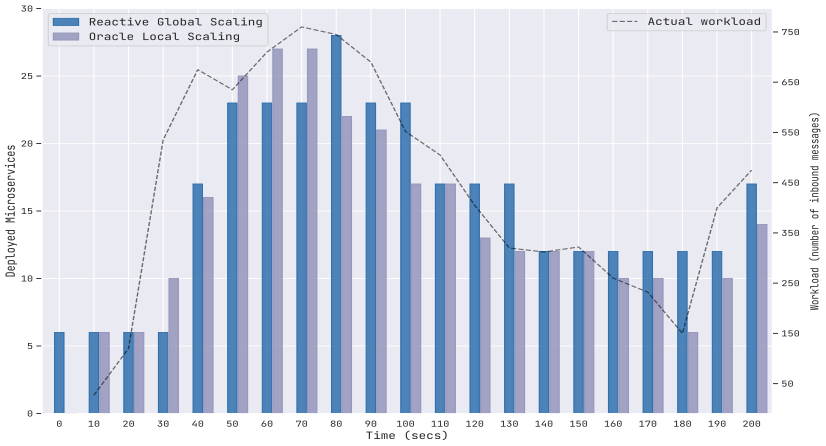
We proposed a novel global scaling approach based on exploiting the functional interdependencies among microservices and a proactive-reactive algorithm based on such an approach. We benchmarked the performance of our global scaling approach using different microservice systems and datasets: all benchmarks show that our approach outperforms the mainstream scaling approach, i.e., the local one. Moreover, we test the performance of our mixing technique by comparing it with an algorithm taken from the literature.

Straightforward directions for future work include both the introduction and refinement of prediction and mixing techniques. For example, one can use natural language processing to extract complementary features for the representation of the regression target (in our case, the inbound requests).

Moreover, we plan to improve system simulation by accounting for failures (e.g., network partitioning, computing hardware failures) and their impact on the deployed system. To this aim, we could evaluate the system following the practice of Chaos Engineering [44], simulating the failures in ABS and making sure that the available resources are enough to guarantee a given level of robustness and resilience.



(a) Latency



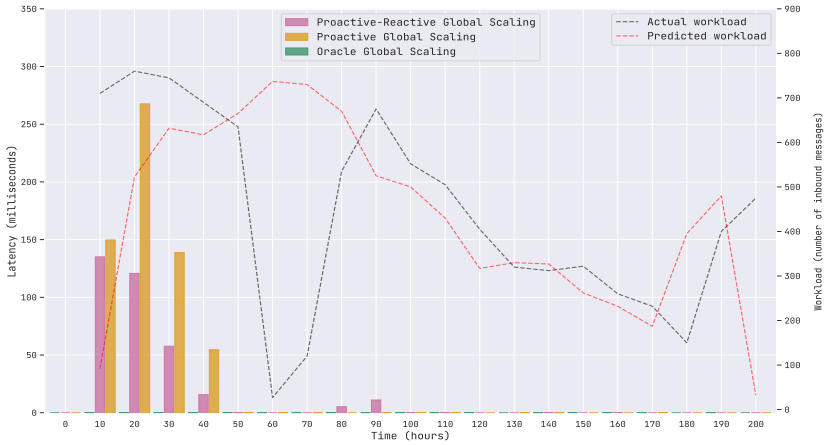
(b) Deployed microservices

Figure 15: Comparing reactive global and reactive local scaling, applied to the TeaStore.

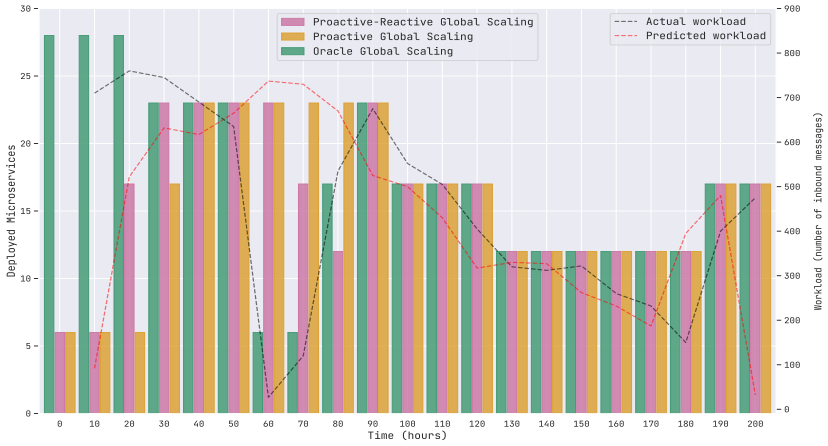
References

[1] K. Hightower, B. Burns, J. Beda, Kubernetes: Up and Running Dive into the Future of Infrastructure, 1st Edition, O'Reilly Media, Inc., 2017.

[2] M. Bravetti, S. Giallorenzo, J. Mauro, I. Talevi, G. Zavattaro, Optimal and automated deployment for microservices, in: Fundamental Approaches to Soft. Eng. - 22nd Intl. Conf., FASE 2019, April 6-11, 2019,Proc., Vol. 11424 of Lecture Notes in Computer Science, Springer, 2019, pp. 351–368. doi:10.1007/978-3-030-16722-6_21.



(a) Latency



(b) Deployed microservices

Figure 16: Proactive-Reactive global scaling performance evaluation (outliers dataset), applied to the TeaStore.

[3] M. Bravetti, S. Giallorenzo, J. Mauro, I. Talevi, G. Zavattaro, A formal approach to microservice architecture deployment, in: Microservices, Science and Eng., Springer, 2020, pp. 183–208. doi:10.1007/978-3-030-31646-4_8.

[4] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: A core language for abstract behavioral specification, in: Formal Methods for Components and Objects - 9th Intl. Symposium, FMCO 2010, Graz,

Austria, November 29 - December 1, 2010. Revised Papers, Vol. 6957 of Lecture Notes in Computer Science, Springer, 2010, pp. 142–164. doi:10.1007/978-3-642-25271-6_8.

[5] B. Klimt, Y. Yang, The enron corpus: A new dataset for email classification research, in: Machine Learning: ECML 2004, 15th European Conference on Machine Learning, Pisa, Italy, September 20-24, 2004, Proc, Vol. 3201 of Lecture Notes in Computer Science, Springer, 2004, pp. 217–226. doi:10.1007/978-3-540-30115-8_22.

[6] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, C. Delimitrou, An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems, in: Proc of the Twenty-Fourth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, Association for Comp. Machinery, New York, NY, USA, 2019, p. 3–18. doi:10.1145/3297858.3304013.

[7] M. Karamollahi, C. Williamson, Characterization of IMAPS email traffic, in: 27th IEEE Intl. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2019, Rennes, France, October 21-25, 2019, IEEE Computer Society, 2019, pp. 214–220. doi:10.1109/MASCOTS.2019.00030.

[8] Amazon, Aws cloudwatch, http://tinyurl.com/44bbyc46.

[9] Apache, Apache mesos, http://tinyurl.com/yt3uttk5.

[10] Docker, Docker swarm, http://tinyurl.com/ymy9m8a4.

[11] B. Urgaonkar, P. J. Shenoy, A. Chandra, P. Goyal, T. Wood, Agile dynamic provisioning of multi-tier internet applications, ACM Trans. Auton. Adapt. Syst. 3 (1) (2008) 1:1–1:39. doi:10.1145/1342171.1342172.

[12] L. Bacchiani, ABS global scaling, http://tinyurl.com/3mp29w2a.

[13] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, S. Kounev, TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research, in: Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS '18, 2018.

[14] S. Eismann, C.-P. Bezemer, W. Shang, D. Okanović, A. van Hoorn, Microservices: A performance tester's dream or nightmare?, in: Proceedings of the ACM/SPEC International Conference on Performance Engineering, ICPE '20, 2020, p. 138–149.

[15] L. Bacchiani, M. Bravetti, S. Giallorenzo, J. Mauro, I. Talevi, G. Zavattaro, Microservice dynamic architecture-level deployment orchestration, in: Coordination Models and Languages - 23rd IFIP WG 6.1 Intl. Conf., COORDINATION 2021, 2021, Valletta, Malta, June 14-18, 2021,Proc., Vol. 12717 of Lecture Notes in Computer Science, Springer, 2021, pp. 257–275. doi:10.1007/978-3-030-78142-2_16.

[16] L. Bacchiani, M. Bravetti, M. Gabbrielli, S. Giallorenzo, G. Zavattaro, S. P. Zingaro, Proactive-reactive global scaling, with analytics, in: Service-Oriented Comp. - 20th Intl. Conf., ICSOC 2022, Seville, Spain, November 29 - December 2, 2022, Proc., Vol. 13740 of Lecture Notes in Computer Science, Springer, 2022, pp. 237–254. doi:10.1007/978-3-031-20984-0_16.

[17] G. Yu, P. Chen, Z. Zheng, Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach, IEEE Trans. Cloud Comput. 10 (2) (2022) 1100–1116. doi:10.1109/TCC.2020.2985352.

[18] B. Liu, R. Buyya, A. N. Toosi, A fuzzy-based auto-scaler for web applications in cloud computing environments, in: Service-Oriented Computing - 16th Intl. Conf., ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proc., Vol. 11236 of Lecture Notes in Computer Science, Springer, 2018, pp. 797–811. doi:10.1007/978-3-030-03596-9_57.

[19] S. Dutta, S. Gera, A. Verma, B. Viswanathan, Smartscale: Automatic application scaling in enterprise clouds, in: 2012 IEEE Fifth Int. Conf.

on Cloud Comp., Honolulu, HI, USA, June 24-29, 2012, IEEE Computer Society, 2012, pp. 221–228. doi:10.1109/CLOUD.2012.12.

[20] N. Marie-Magdelaine, T. Ahmed, Proactive autoscaling for cloud-native applications using machine learning, in: IEEE Global Commu. Conf., GLOBECOM 2020, Virtual Event, Taiwan, December 7-11, 2020, IEEE, 2020, pp. 1–7. doi:10.1109/GLOBECOM42002.2020.9322147.

[21] J. Park, B. Choi, C. Lee, D. Han, GRAF: a graph neural network based proactive resource allocation framework for slo-oriented microservices, in: CoNEXT '21: The 17th Int. Conf. on emerging Networking EXperiments and Technologies, Virtual Event, Munich, Germany, December 7 - 10, 2021, ACM, 2021, pp. 154–167. doi:10.1145/3485983.3494866.

[22] C. Qu, R. N. Calheiros, R. Buyya, Auto-scaling web applications in clouds: A taxonomy and survey, ACM Comput. Surv. 51 (4) (2018) 73:1–73:33. doi:10.1145/3148149.

[23] A. Bauer, V. Lesch, L. Versluis, A. Ilyushkin, N. Herbst, S. Kounev, Chamulteon: Coordinated auto-scaling of micro-services, in: 39th IEEE Int.l Conf. on Distr. Comp. Sys., ICDCS 2019, Dallas, TX, USA, July 7-10, 2019, IEEE, 2019, pp. 2015–2025. doi:10.1109/ICDCS.2019.00199.

[24] Amazon, AWS Auto Scaling, http://tinyurl.com/3kuzyjnw.

[25] Microsoft, Overview of autoscale in Azure, http://tinyurl.com/5fbsdpfc.

[26] Google, Scaling based on predictions, http://tinyurl.com/4h7bezn6.

[27] H. Ahmad, C. Treude, M. Wagner, C. Szabo, Smart HPA: A resource-efficient horizontal pod auto-scaler for microservice architectures, CoRR abs/2403.07909 (2024). arXiv:2403.07909, doi:10.48550/ARXIV.2403.07909.
URL https://doi.org/10.48550/arXiv.2403.07909

[28] A. U. Gias, G. Casale, C. M. Woodside, ATOM: model-driven autoscaling for microservices, in: 39th IEEE Intl. Conf. on Distr. Comp. Sys., ICDCS 2019, Dallas, TX, USA, July 7-10, 2019, IEEE, 2019, pp. 1994–2004. doi:10.1109/ICDCS.2019.00197.

[29] J. Park, B. Choi, C. Lee, D. Han, Graf: a graph neural network based proactive resource allocation framework for slo-oriented microservices, Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies (2021).
URL https://api.semanticscholar.org/CorpusID:244841439

[30] J. Park, B. Choi, C. Lee, D. Han, Graph neural network-based slo-aware proactive resource autoscaling framework for microservices, IEEE/ACM Transactions on Networking (2024).
URL https://api.semanticscholar.org/CorpusID:269574996

[31] J. P. K. S. Nunes, S. Nejati, M. Sabetzadeh, E. Y. Nakagawa, Self-adaptive, requirements-driven autoscaling of microservices, 2024 IEEE/ACM 19th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) (2024) 168–174.
URL https://api.semanticscholar.org/CorpusID:268385435

[32] N. Bartelucci, P. Bellavista, T. W. Pusztai, A. Morichetta, S. Dustdar, High-level metrics for service level objective-aware autoscaling in polaris: a performance evaluation, 2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC) (2022) 73–77.
URL https://api.semanticscholar.org/CorpusID:247441441

[33] A. F. Baarzi, G. Kesidis, Showar: Right-sizing and efficient scheduling of microservices, in: Proceedings of the ACM Symposium on Cloud Computing, SoCC '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 427–441. doi:10.1145/3472883.3486999.
URL https://doi.org/10.1145/3472883.3486999

[34] M. Abdullah, W. Iqbal, J. L. Berral, J. Polo, D. Carrera, Burst-aware predictive autoscaling for containerized microservices, IEEE Transactions on Services Computing 15 (3) (2022) 1448–1460. doi:10.1109/TSC.2020.2995937.

[35] S. Xie, J. Wang, B. Li, Z. Zhang, D. Li, P. C. K. Hung, Pbscaler: A bottleneck-aware autoscaling framework for microservice-based applications, IEEE Transactions on Services Computing 17 (2) (2024) 604–616. doi:10.1109/TSC.2024.3376202.

[36] Docker, Docker compose, http://tinyurl.com/2eaeea52.

[37] R. D. Cosmo, S. Zacchiroli, G. Zavattaro, Towards a formal component model for the cloud, in: Soft. Eng. and Formal Methods - 10th Intl. Conf., SEFM 2012, Thessaloniki, Greece, October 1-5, 2012.Proc., Vol. 7504 of Lecture Notes in Computer Science, Springer, 2012, pp. 156–171. doi:10.1007/978-3-642-33826-7_11.

[38] E. Ábrahám, F. Corzilius, E. B. Johnsen, G. Kremer, J. Mauro, Zephyrus2: On the fly deployment optimization using SMT and CP technologies, in: Dependable Soft. Eng.: Theories, Tools, and Applications - Second Intl. Symposium, SETTA 2016, Beijing, China, November 9-11, 2016,Proc., Vol. 9984 of Lecture Notes in Computer Science, 2016, pp. 229–245. doi:10.1007/978-3-319-47677-3_15.

[39] A. Rawdat, Testing the performance of nginx and nginx plus web servers, http://tinyurl.com/a9n2n8wv.

[40] J. D. Kelleher, B. Mac Namee, A. D'arcy, Fundamentals of machine learning for predictive data analytics: algorithms, worked examples, and case studies, MIT press, 2020.

[41] B. Rabenstein, J. Volz, Prometheus: A Next-Generation monitoring system (talk), USENIX Association, Dublin, 2015.

[42] L. Bacchiani, G. D. Palma, L. Sciullo, M. Bravetti, M. D. Felice, M. Gabbrielli, G. Zavattaro, R. D. Penna, Low-latency anomaly detection on the edge-cloud continuum for industry 4.0 applications: the SEAWALL case study, IEEE Internet Things Mag. 5 (3) (2022) 32–37. doi:10.1109/IOTM.001.2200120.

[43] L. Bacchiani, ABS service migration, http://tinyurl.com/yrhncr5v.

[44] N. J. Casey Rosenthal, Chaos Engineering, 1st Edition, O'Reilly Media, Inc., 2020.