Proactive-Reactive Global Scaling, with Analytics

Lorenzo Bacchiani[1] Mario Bravetti[1,2] Maurizio Gabbrielli[1,2]
Saverio Giallorenzo[1,2] Gianluigi Zavattaro[1,2] Stefano Pio Zingaro[1,2]

[1] Università di Bologna, Italy
[2] Focus Team, INRIA, France

Abstract. In this work, we focus on *by-design global scaling*, a technique that, given a functional specification of a microservice architecture, orchestrates the scaling of all its components, avoiding cascading slowdowns typical of uncoordinated, mainstream autoscaling. State-of-the-art by-design global scaling adopts a reactive approach to traffic fluctuations, undergoing inefficiencies due to the reaction overhead. Here, we tackle this problem by proposing a *proactive* version of *by-design global scaling* able to anticipate future scaling actions. We provide four contributions in this direction: i) a platform able to host both reactive and proactive global scaling; ii) a proactive implementation based on data analytics; iii) a hybrid solution that mixes reactive and proactive scaling; iv) use cases and empirical benchmarks, obtained through our platform, that compare reactive, proactive, and hybrid global scaling performance. From our comparison, proactive global scaling consistently outperforms reactive, while the hybrid solution is the best-performing one.

Keywords: Microservices · Architecture-level Scaling · Predictive 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 [1] and horizontal (auto)scaling [2]. Although a powerful resource, scaling comes with its own challenges. As Ghandi et al. [3] put it:

> [...] it is up to the customer (application owner) to leverage the flexible platform. That is, the user must decide when and how to scale the application deployment to meet the changing workload demand.

Background Our work focuses on *global scaling* [4,5,6,7,8,9]; which orchestrates the scaling of all microservices in a given architecture. This contrasts with *local scaling*, intended as the mainstream interpretation of (auto)scaling [2], which scales microservices in an uncoordinated way. Performance-wise, local scaling suffers from *domino effects*, also called *bottleneck shift*, where the uncoordinated scaling actions cause waves of cascading slowdowns and possibly generate outages [10,4].

Problem Existing global scaling approaches focus on smoothing out domino effects [4,6,8] or on removing them by design [5,7,9]. This "by-design" approach performs the coordinated scaling of the microservices based on a quantification of their functional relations. However, existing work on by-design global scaling only focused on *reacting* to fluctuations of inbound traffic, wasting time to the detriment of customers, who can endure delays, downtimes, and receive a lower-than-expected level of service.

Contributions In this paper, we challenge the existing reactive interpretation of by-design global scaling—hereinafter, we omit the "by-design" suffix. We hypothesise that global scaling might endure some performance inefficiencies due to its reaction overhead, which is the starting point of our contributions.

In Section 2, we *present a platform* able to host both reactive and *proactive global scaling*, e.g., it allows users to programmatically switch between the two approaches. We simulate an ideal, oracle proactive global scaler, and we show *empirical benchmarks* of the inefficiencies of reactive global scaling and of the possible gains of proactive global scaling.

In Section 3, we introduce a *proactive global scaling* implementation based on analytics [11]. We present a use case on email traffic from the Enron dataset [12]. We *benchmark* this implementation, which overcomes the limitations of its reactive counterpart and approximates the ideal performance of the oracle

In Section 4, we present an algorithm that (deployed in our platform) integrates proactive and reactive global scaling. *Benchmarks* on the Enron use case show that this hybrid approach is the best-performing one.

Our datasets, trained models, and simulations are publicly available at [13], which also contains a containerized version of the testbed.

2 Proactive Global Scaling

We introduce a platform that DevOps can use to perform proactive and reactive global scaling. In doing so, we do not start from scratch, and we build on previous work on global scaling, proposing a redesign able to capture proactive scaling on the existing reactive global scaling platform from [5,7,9]. Our new architecture is immediately useful. We use it at the end of this section to quantify the untapped potential of proactive global scaling—comparing the performance of reactive local and global scaling vs an ideal, oracle proactive global scaler. We use our platform also later, in Sections 3 and 4, to benchmark our implementations of proactive and proactive-reactive scalers.

Global Scaling In global scaling, the user provides a specification of the scaling constraints of each component of a given architecture, both in terms of necessary resources (such as CPU and memory) and of its dependencies on other microservices (e.g., microservice M_1 needs two copies of the microservice M_2 to run properly). Then, given one such specification, and using dedicated resolution engines [5], deployment plans can be computed such that: i) scale the whole

architecture w.r.t. an expected increase/decrease of inbound traffic; ii) respect (if any) the constraints of resource allocation and dependency of the scaled microservices; iii) optimise the plan towards some set goals, e.g., minimising the cost of running the scaled architecture, i.e., using the minimal amount of virtual machines that supports the execution of the scaled architecture.

2.1 Design of a Proactive-Reactive Global Scaling Platform

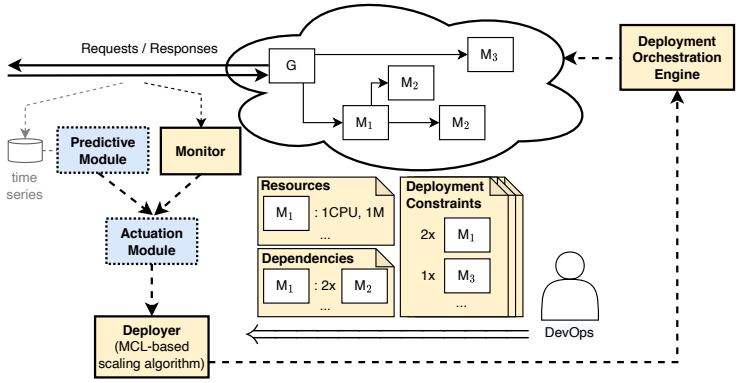


Fig. 1. Architectural view of the proactive-reactive global scaling platform.

We depict the architecture of the platform in Fig. 1, which includes two kinds of elements. The components found in the "cloud" are the microservices of a given architecture, labelled G, M_1, M_2, M_3. The global scaling platform manages the scaling of these microservices. Since the platform sees microservices as instance parameters, we abstract from their actual behaviour and use them in examples. The other elements in Fig. 1 are the components of the platform. Specifically, the elements with continuous-line borders are the ones inherited from previous work [5,7,9]. The main new elements, drawn with a dotted border, are the Predictive Module and the Actuation Module.

For completeness, we first describe the elements of the platform already present in the original proposals [5,7,9], and then dedicate to the new components for proaction, i.e. Actuation and Predictive Modules. Before doing so, we highlight the three kinds of flows in Fig. 1: continuous-line arrows $\rightarrow$ show the traffic addressed to the microservice architecture; dashed-line arrows $\dashrightarrow$ regard the runtime execution of global scaling; the thick arrow $\Leftarrow$ indicates the compilation time of deployment plans.

Deployment Orchestration Engine This component performs the actual scaling, (de)allocating replicas of microservices. It is a loosely-coupled component of the platform, taken from existing solutions (the only requirement is that it

provides a programming interface for the application of deployment plans), such as Kubernetes.

Deployer The Deployer implements the Maximum Computational Load (MCL) scaling algorithm and the deployment strategy proposed in [9], and it regards two flows. The first one, represented by $\Leftarrow$, regards the computation of the deployment plans, applied by the scaling $\dashrightarrow$. As such, this process is asynchronous w.r.t. both the scaling and the traffic flow $\rightarrow$. In $\Leftarrow$, the Deployer takes the specifications given by the user (DevOps in Fig. 1) and computes the deployment plans that satisfy the Resources needed by each microservice (e.g., M_1 needs 1 CPU and 1 Memory), the Dependencies among the microservices (e.g., microservice M_1 needs two copies of M_2 to work), and the Deployment Constraints of different scaling targets. Since these deployment plans represent differential increments/decrements in microservice replicas, we call them *deltas*. The second flow, that of the runtime scaling $\dashrightarrow$, runs alongside the inbound traffic $\rightarrow$. In this case, the Deployer acts as a service that other components call to trigger the application of a target, computed delta. Upon activation, the Deployer interacts with the Deployment Orchestration Engine to perform the scaling.

Monitor In its original formulation, the monitor tracks the traffic flowing on the architecture within a prefixed time unit and checks the possible occurrence of a *workload deviation*, i.e., a discrepancy between the current, tracked workload and the expected one, correspondent to the delta currently applied. When such a condition occurs, the Monitor triggers the Deployer to apply the delta that corresponds to the current traffic load. To support proaction, we break the above, direct relation between the Monitor and the Deployer, as detailed below.

Actuation Module This is the first component we introduce to support the coexistence of proactive and reactive global scaling. This is achieved by breaking and controlling the once-direct triggering relation between the Monitor and the Deployer, i.e., it is now the Actuation Module that decides when/whether to trigger the Deployer. This redesign allows the seamless coexistence of the previous reactive modality with the new proactive one. Indeed, to obtain the same behaviour of the original proposal, we just need to set the Actuation Module in "passive" mode and let it forward triggers from the Monitor to the Deployer. When active, the Actuation Module can choose to act independently of the traffic, e.g., choosing to ignore information coming from the Monitor and trigger the Deployer according to signals coming from other sources, e.g., from the Predictive Module, described below. As discussed in Section 4, the Actuation Module is where the DevOps defines algorithms that can dynamically decide when to follow the anticipated scaling from forecasts or react to the signals from the Monitor.

Predictive Module The Predictive Module acts independently of the actual inbound traffic forwarding the prediction to the Actuation Module. For instance, the Predictive Module can use a static model, e.g., forecasting traffic peaks at predetermined times, or sophisticated techniques to have more accurate predictions

of traffic fluctuations, e.g., based on data analytics. In Fig. 1, we represent the input of the Predictive Module with the greyed-out arrow receiving information from the traffic flow and stores it into a time series dataset for further usage.

2.2 Benchmarking the Platform

To run our benchmarks, we relied on simulations. Specifically, we modelled our platform and the scaling approaches via the ABS programming language [14], compiling it into a system of Erlang programs that run the simulation. These programs and their execution environment form the test-bed of all our simulations, and we provide it as a container in the companion repository [13].

The simulation receives three kinds of inputs, which are statically defined within a simulation run: a *real inbound workload* (RIW), a *predicted inbound workload* (PIW), and the *deployment plans* (DP). The simulation combines these inputs to benchmark the performance of a target microservices architecture.

Notably, while we fix all inputs in simulations, this diverges from real executions only on the source of RIWs. Indeed, in real executions, RIWs corresponds to the traffic reaching the architecture, while, in our simulations, we generate RIWs beforehand–specifically, from samplings of actual traffic. RIW apart, also in real executions PIW and DP are normally computed before their utilisation time-window. For instance, since they tend to be time-consuming calculations, one can compute PIWs and DPs for the coming day during the preceding night.

Since our simulator is parametric to a target microservices architecture, we fix one for the benchmarks throughout this paper: the Email Pipeline Processing System from Bacchiani et al. [9], which includes twelve microservices, each with its own load balancer for distributing requests over the available replicas.

2.3 Reactive Local vs Reactive Global vs Proactive Global Scaling

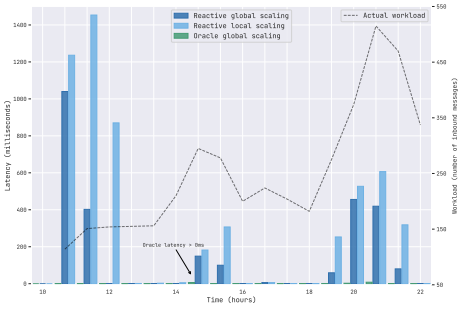


Fig. 2. Latency of proactive oracle global scaling vs reactive global and reactive local scaling.

We use our platform to empirically benchmark the gain of proactive global scaling vs both classic reactive global scaling and local scaling, in terms of latency, considered as the average time for completely processing a request that entered the system. We simulate an oracle, an ideal omniscient predictor that proactively triggers the global scaling of the architecture on forecasted traffic.

Since our benchmarks (throughout the paper) concern time, we partition the inbound traffic within discrete "time units" (e.g., we aggregate all requests received between 9 a.m. and 10 a.m. within a corresponding logical time unit). In this benchmark, we draw the traffic flow from [9].

Technically, the oracle knows the exact number of incoming requests and anticipates by one time unit the triggering of the Deployer and the related provisioning of resources. Here, we assume that a time unit is enough for the Deployer to (de)allocate the necessary resources before the traffic arrives.

As mentioned in the previous section, to run the benchmarks for the global reactive case, we disable the Predictive Module. Indeed, we obtain the same results from [9], with the only difference that we modified the Monitor to keep track of the maximum inbound workload, while Bacchiani et al. [9] use the average. We deem this choice sensible, since it provides results consistent with those of Bacchiani et al. [9], but it is less sensitive to irregular workloads. Also the implementation of the oracle is straightforward: we provide the Predictive Module with the considered traffic flow, shifted by one time unit, so that the Actuation Module anticipates the overhead of triggering the Deployer and applying the delta to handle the traffic of the next time unit. Finally, we adopt the same program (defined in ABS and run with Erlang) Bacchiani et al. [9] used to simulate reactive local scaling for that architecture.

Notably, while the traffic flow (generator) from [9] follows a fixed curve, it generates email attachments randomly. This is the only information unknown to the oracle that we expect to impact on its (otherwise non-existent) latency.

In Fig. 2, we show the comparison between oracle proactive global scaling, reactive global scaling, and local scaling. From the results, local scaling is the worse-performing one, due to both the reactive overhead and the domino effect (see Section 1). Reactive global scaling does not suffer from the latter phenomenon, but (confirming our hypothesis) it endures the overhead of applying deltas in response to traffic fluctuations (e.g., while the scaling takes effect, new messages arrive that are enqueued, increasing latency). As expected, the oracle performs almost perfectly and shows minimal latency at the time units between 14–16, likely due to the number of attachments in emails that exceeded the expected average considered in the deployment plans (this parameter is the same across all three modalities).

To give a quantitative intuition of the performance gap between the ideal proactive oracle and the other reactive approaches, we report the Area Under Curve for the latencies, in seconds, computed using the composite trapezoidal rule: oracle 0.1, reactive global scaling 13, reactive local scaling 29.

Summing up, while reactive global scaling already outperforms reactive local scaling (ca. 44% reduced latency), having an implementation that approximates the behaviour of the proactive oracle could further increase the performance of global-scaling system (ca. 70% reduced latency).

Notably (as illustrated later in Section 3) (de)allocating resources in advance does not have drawbacks from the point of view of costs, since resources are also (de)allocated earlier than in a reactive approach—indeed, by definition, the oracle does not change the scaling sequence applied also by the reactive global scaler, but rather anticipates them by one time unit.

3 Analytics-based Proactive Global Scaling

Given the promising results from Section 2, we demonstrate how one can develop a realistic proactive global scaling implementation, i.e., using a state-of-the-art data analytics technique to obtain a predictor and integrating it in our Predictive Module. Here, we introduce the steps of general data analytics and we specify how these impact the workflow of Fig. 1. Next, we concentrate on the use case and how we applied data analytics to build the predictor. Finally, we compare our implementation of analytics-based proactive global scaling against oracle proactive global scaling (similar to the one presented in Section 2) reactive global scaling, and oracle proactive local scaling.

3.1 Data Analytics for Global Scaling

The Steps of Data Analytics We provide a general overview of the elements of data analytics [11] (DA) and then detail how we applied these in our use case.

Using data analytics to predict the occurrence of the event in a given time unit, we aim to understand which variables influence and describe the phenomenon (*descriptive DA*). Once we selected the variables, we need to understand which attribute values are most relevant (*diagnostic DA*). Then, using the attributes and diagnostic(s) of the phenomenon, we are able to build a dataset to automatically train a model and infer the outcome of a new instance of the phenomenon, i.e., an event (of the same nature as the one being studied) not yet observed (*predictive DA*). The model created provides a description of all the observed and new events. Each of the possible outcomes relates with one or more configurations of the system. Each configuration corresponds to a response strategy to the occurrence of events similar to those already observed (*prescriptive DA*). Given a specific system configuration, we can compute its efficiency and select the one that offers the best cost-benefit trade-off (*proactive DA*). This optimal configuration, if any, is the one sent to the actuator.

The Workflow of Analytics-based Proactive Global Scaling As mentioned in Section 2, we use the new modules introduced in this work in our global scaling platform (cf. Fig. 1), namely the Predictive Module and the Actuation Module, to capture the steps of data analytics. Specifically, the Predictive Module implements the steps of descriptive and diagnostic (prepare the dataset) and predictive data analytics (train and inference) while the Actuation Module realises the prescriptive (define the scaling strategy) and proactive steps (triggering policies).

In pure proactive scaling, the Actuation Module computes the scaling strategy, given the outcomes of the Predictive Module and directly triggers the Deployer, disregarding any inputs coming from the Monitor.

Architecture and Dataset used in the Benchmark After seeing the general workflow of analytics-based proactive global scaling, we introduce our use case and illustrate how each of its parts fit into said workflow.

However, instead of using the fixed traffic from [9], which provides too little information to train a data analytics model, we draw our dataset from another, renowned source—e.g., for training email schedulers and SPAM filters—from the literature, that has a compatible structure (i.e., email traffic): the Enron corpus dataset [12], made public by the Federal Energy Regulatory Commission during investigations concerning the Enron corporation (version of May 7th, 2015). The dataset contains 517,431 emails from 151 users, without attachments, distributed over a time window of about 10 years (1995-2005).

Descriptive and Diagnostic DA in Predictive Module Leveraging the pre-processing routine from [12], we perform the cleaning procedure of the Enron dataset for classification tasks, and then we extract the attributes for predicting the number of incoming emails for a given time. First, we extract the *datetime* attribute for each email in the dataset, and then we sum the number of emails in the desired monitored time unit—i.e., one hour—for each month of the year, day of the month, and day of the week. Thus, we generate five new attributes: *month*, *day*, *weekday*, *hour*, and *counter*—the target—for each dataset instance. This gives us a representation of the email flow in the system at a given hour. The intuition for such a pre-processing is simple. The phenomena of increase or decrease in the flow of emails that occur in a company depend on factors such as the specific time of the working day (peak in the early hours versus the night hours), the month (monthly, bimonthly, etc. deadline), the day of the month (salary) or the day of the week (weekdays versus holidays).

Predictive DA in Predictive Module For the predictive phase, we use off-the-shelf machine learning technique, specifically MLP (Multi-Layer Perceptron), which is capable—in contrast with purely linear models, e.g., linear regression—of exploring nonlinear patterns and increase prediction performance while containing complexity (about 7k parameters) and resource usage (about 1ms inference time). We categorise the numerical variables using the standard one-hot encoding technique, to prevent our model from attributing wrong semantics to these (e.g., month 12 is "greater than" month 1), resulting in a data representation of 70 attributes plus the *counter* target.

Then, we followed the traditional training process for machine learning. We partitioned the cleaned, processed data into three sets: one for training the neural network model, one for validating its hyperparameters (the parameters of the training process and network architecture), and one for testing the accuracy of the model. We use this last set to compute the error rate of the model.

The neural network used in the training process consists of three fully-connected layers. We applied the Rectified Linear Unit (ReLU) nonlinear activation function to the output of each layer. Each level compresses the input into a smaller representation, going from 70 to 64 attributes, in the first level, and from 64 to 32 attributes, in the second level. Finally, the 32 attributes are linearly projected into a single value, corresponding to the target of the regression. To compute the error rate, we adopt the loss function Mean Squared Error (MSE). To optimise the network parameters we use Adaptive Moment Estimation (Adam).

We performed the training process with a learning rate of 0.1 and an exponential decay scheduler with gamma 0.9.

After the training, given a time slot—the tuple month (1–12), day (1–31), weekday (1–7), and hour (0–24)—the predictor forecasts the amount of emails incoming therein.

This is the third and last step of the data analytics workflow that concerns the Predictive Module. Here, we embedded the trained model to make the Predictive Module yield a prediction of the expected traffic, given a target time slot.

Prescriptive and Proactive DA in Actuation Module The last two steps of the data analytics workflow are the prescriptive and proactive ones. We realise these in the Actuation Module. Since we implement pure predictive autoscaling, the prescriptive step is straightforward: we follow the prediction from the Predictive Module. The proactive step is the implementation of the strategy, where we forward of the expected traffic from the Predictive Module to the Deployer.

3.2 Benchmarking the Performance of Analytics-based Global Scaling

Analytics-based Proactive vs Reactive and Oracle Global Scaling To give an intuition of the effectiveness of our analytics-based proactive global scaler, we test its performance against reactive global scaling [9] and an oracle similar to that seen in Section 2—also here, simulated by fixing a traffic flow and applying the related deltas one time-unit before the actual execution time.

Consistently with the oracle in Section 2, we do not fix also here the number of attachments in inbound emails but define them randomly. This comparison mainly aims at showing the performance gap between the analytics-based proactive and the oracle proactive variants (i.e., how close the former approximates the ideal proactive scaler), keeping reactive global scaling as a baseline for the comparison. To this aim, we report latency, message loss, cost, and number of deployed microservices. All benchmark tests shown in this section are performed on email traffic on a weekday in May 2001.

Considering latency, as shown in Fig. 3a, reactive scaling is the worst and presents high peaks of latency when the inbound workload grows. The oracle, similarly as in Section 2, is barely visible because, by construction, it knows in advance the exact amount of inbound messages, thus, it anticipates required scaling actions, with negligible latency. Performance-wise, our analytics-based global scaler closely approximates the oracle. Indeed, it mainly differs in two small spikes, imputable to inaccuracies in the workload predictions. Since latency and message loss (see Fig. 3b) are strictly related, we have similar conclusions: the oracle loses no messages, followed by the analytics-based one, while reactive scaling loses the most, at sudden peaks of workload. The number of deployed microservices and costs are also directly proportional, as seen in respectively Figs. 3c and 3d. Despite the sensible performance difference between the oracle, analytics-based, and reactive scaling, the costs/number of deployed instances are the same, although shifted by a time-unit backwards. The reason is that, since

(a) Latency (b) Lost messages

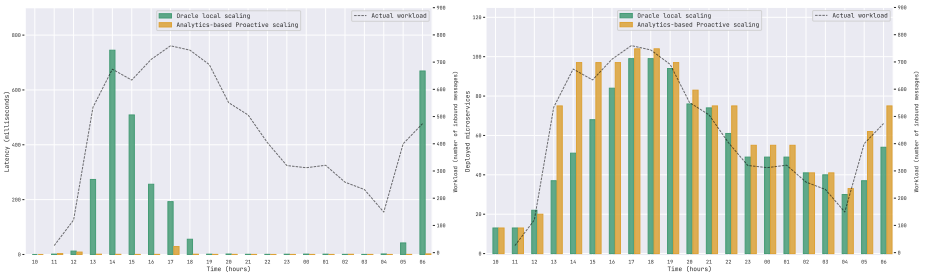
(c) Virtual machine hourly cost (d) Deployed microservices

Fig. 3. Comparison between reactive, analytics-based proactive, and oracle-based proactive global scaling approaches.

the traffic is the same, resource (de)allocations are the same across all scalers, although these happen one time-unit in advance in the oracle and analytics-based proactive cases—divergences of the analytics-based proactive scaling derive from inaccuracies of the trained predictor.

Analytics-based Proactive Global Scaling vs Oracle Local Scaling We also compare our analytics-base proactive global scaler with oracle local scaling in Fig. 4a and Fig. 4b, i.e., a scaler that knows the future traffic of each microservice in an architecture and performs microservice-level scaling in advance. The purpose of this test is to give empirical evidence of the benefits of global vs local scaling, which holds in the reactive case—as proven in [9]—as well as in the proactive one (oracle and analytics-based). The rationale is that, if we show that analytics-based proactive global scaling outperforms ideal proactive local scaling, then i) the latter performs worse than the oracle global scaler and ii) the former outperforms any analytics-based local scaling.

In this experiment, we focus on the evaluation of the same performance as in the previous benchmark, but, for brevity, we only report latency and the number of deployed microservices measures, since these are proxies for the respective other two, directly-proportional measures: message loss and costs. Starting from latency, reported in Fig. 4a, analytics-based proactive global scaling outperforms ideal proactive local scaling. The former has almost 0 latency throughout the



(a) Latency (b) Deployed microservices

Fig. 4. Comparison between analytics-based proactive global scaling and oracle-based proactive local scaling.

entire experiment, except for little spikes within time unit 17. The latter struggles to adapt to rapid changes in inbound requests (intervals 13–18 and 5–6). The deterioration in performance of the oracle-based local scaling is caused by the so-called "domino" effect, i.e., single services scaling one after the other, causing chained slowdowns [9]. Besides worsening the performance, the domino effect also limits the predictive power of the local approach: the first microservice in a call chain is the one that anticipates the scaling, while the other ones cascade-scale only after that moment. This uncoordinated scaling leads to situations where the overheads of scaling accumulate sequentially (instead of executing in parallel, as in the global case), degrading performance. The presence of the domino effect is witnessed in particular in Fig. 4b. In analytics-based proactive global scaling, the number of deployed instances reaches the target amount to handle the inbound workload as soon as it is foreseen by the proactive module. Instead, the ideal proactive local scaler can only grow the number of deployed instances linearly over time, following the chain of scaling/forecast of the single microservices.

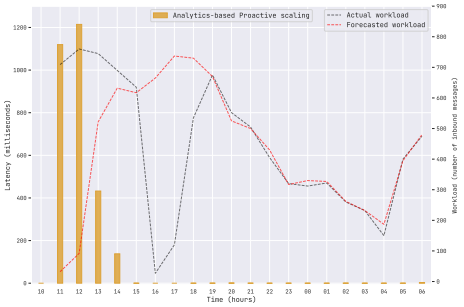


Fig. 5. Latency of the analytics-based proactive global scaling on the outliers test set.

Limitations of Analytics-based Proactive Global Scaling: Outliers The analytics-based proactive approach presented in this section proved to be quite effective. However, predictors are not infallible: if the traffic greatly deviates from the historical data, due to some unprecedented occurrence, the predictor can fail to provide an accurate estimation of the traffic.

This fact, considered in the context of purely analytics-based global scaling (like the one implemented above) where scaling decisions neglect actual traffic fluctuations, can result in over- (wasted resources) or under-scaling (latency, request loss) of the system. To illustrate how much this phenomenon can

affect performance, we selectively picked outliers from the test set described in Section 3.1 and used these to produce a traffic flow that our predictor would struggle to forecast. As shown in Section 3.2, when unpredicted peaks occur, latency grows, causing performance deterioration. In the next section, we propose a solution that overcomes this limitation by mixing reactive and proactive global scaling.

4 Proactive-Reactive Global Scaling

The fact that predictors are weak against exceptional events is a well-known fact (see Section 5), which we concretely showed (see Section 3) can affect pure proactive global scaling too, resulting in the application of inappropriate deltas (either wasting resources or degrading the level of service). In this section, we propose a solution to this limitation, mixing proactive global scaling with reactive global scaling. Our global-scaling platform architecture (Section 2) simplifies this task: we program the Actuation Module to calculate an accuracy threshold which defines when to follow the forecasts of the Predictive Module or switch to the reactive signals of the Monitor.

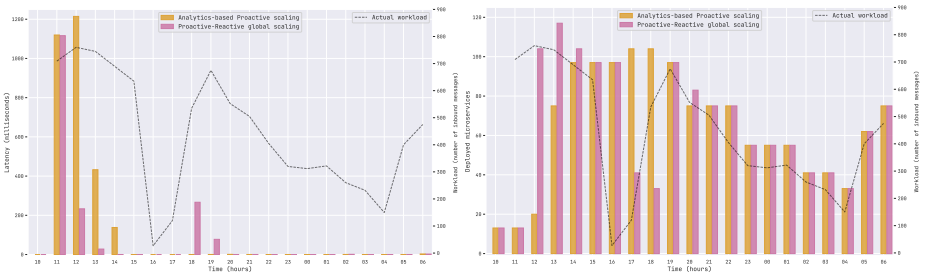
Our algorithm does not rely on comparing the estimated and actual number of inbound requests in a given time unit. The reason is that the dynamic interaction between message queues and scaling times makes it difficult to reliably estimate the accuracy of the predicted scaling configuration w.r.t. traffic fluctuations. Hence, we introduce a new, stable estimation, rooted in the workload measure defined below.

Our idea is to use the Maximum Computation Load (MCL) from [9], which measures the capacity of a system configuration to handle a given workload. Using the MCL, we 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 estimate both over- and under-scaling of proactive global scaling, given by the distance between the MCL (of the scaling configuration) induced by the actual traffic.

Our estimation considers statically-defined scores for each architectural reconfiguration increment (allocating new system resources, i.e., service instances and bindings [9]), called $\Delta scale$. Hence, each $\Delta scale$ has associated a score s, computed on the basis of the increment in system MCL (i.e., the maximum supported workload for a given inbound traffic). Following [9], we have $i \in [1, 4]$ different $\Delta scale_i$ plans, which are applied sequentially (in the exceptional case $\Delta scale_4$ is not enough, we restart from $\Delta scale_1$, see [9]). For each $\Delta scale_i$ we have a differential system MCL increment of: $\Delta MCL_1 = 60$ for $\Delta scale_1$ and $\Delta MCL_i = 90$ for $\Delta scale_i$ 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 $\sum_{i=1}^{4} s_i = 1$.

Then, for each time unit t, we compute our estimation following these 3 steps.

In step 1, we calculate, for each index i, the absolute value $|diff_i|$ of the difference between the $\Delta scale_i$ of the predicted workload and the observed one at time t. Then, we compute a weight $w \in [0, 1]$ that we later use to



(a) Latency (b) Deployed microservices

Fig. 6. Comparison between hybrid and pure proactive (analytics-based) global scaling, on the outliers test set.

combine the predicted workload and the measured one. Since $|\mathit{diff}_i| > 1$ only happens in exceptional cases (when $\Delta scale_4$ is not enough), we compute $w = min\left(\sum_{i=1}^{4} s_i \cdot |\mathit{diff}_i|, 1\right)$.

We keep track of the values w 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 $(1, w_{t-2}), (2, w_{t-1})$ are included in h only if the system was already running at those times.

In step 2, we compute the overall weight $\mathit{overall\_w} = \frac{\sum_{(i,w)\in h} w \cdot i}{\sum_{(i,\_)\in h} i}$ of t. In particular, $w \cdot i$ means that the most recent w is the most influential one in the sum. The overall weight indicates the distance between the measured workload and the predicted one. Specifically, the closer the overall weight is to 1 the more distant the prediction is from the actual workload.

In step 3, we linearly combine the predicted and the measured workload through $\mathit{overall\_w}$ to estimate the workload used by the global scaler to compute the current system configuration: $\mathit{workload\_estimation} = (\mathit{overall\_w} \cdot \mathit{workload\_measured}) + ((1 - \mathit{overall\_w}) \cdot \mathit{workload\_predicted})$.

Benchmarking the Performance of Proactive-Reactive Global Scaling In the following, we benchmark our hybrid global scaler in two ways. First, we compare the hybrid scaler against the pure proactive global scaler from Section 3. Second, we compare our hybrid scaler with an alternative implementation from the literature [4]. In both benchmarks we re-use the same highly-volatile traffic used in Section 3.

Proactive-Reactive Global Scaling vs Pure Proactive Global Scaling Similarly to what done in Section 3 , we report only latency (Fig. 6a) and number of deployed microservices (Fig. 6b), which are proxies for message loss and costs.

From Fig. 6a, hybrid local scaling rapidly recovers from wrong predictions, while pure proactive scaling neglects unexpected traffic fluctuations. This is visible, e.g., in the interval 11–13, where the pure proactive scaler expects fewer requests and endures high latency. Also the hybrid scaler initially undergoes high

latency, but, detecting the diverge with the predictions, it assumes a reactive stance and quickly adapts. Note that the latency of the hybrid global scaler in the timespan 18–19 is "good". Indeed, while the workload drops between 15–17, the pure proactive scaler allocates a high number of microservices (cf. Fig. 6b), wasting a lot of resources. Contrarily, the hybrid scaler (reacting to the unforeseen change) trades some minor latency off resource savings.

Alternative Hybridisation Techniques Many hybrid local scaling techniques, see Section 5, use local metrics (CPU, memory) that cannot directly translate into global scaling ones. This is because we would need a global measure out of the local ones, however none of them provide a method to obtain this aggregate global measure (understandably, because they are interested in local scaling). Therefore, for the aim of comparison with our hybrid global scaler, we cannot translate the local scaling algorithm used in such techniques into a global one.

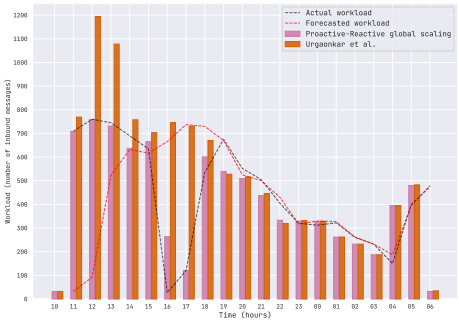


Fig. 7. Comparison of the workload obtained (in our hybrid approach and that of [4]) by mixing the actual and forecasted one.

We are instead able to compare our algorithm with the global one of [4], which, like us, computes the target workload (used for scaling) in terms of received requests per time unit. We implement the algorithm proposed in [4] into an alternative hybrid global scaler and benchmark it using the outliers test set. We report in Fig. 7: the workload of the actual traffic, of the forecasted traffic, and the target workload of our hybrid scaler and that of [4].

As shown in Fig. 7, both techniques adjust underestimations, i.e., they do not let the system degrade its level of service. However, the alternative implementation is not able to adjust overestimated predictions (range 15–18), which end up wasting resources (and money)—a shortcoming reported in [4]. Besides this qualitative trait, quantitatively, our mixing approach is more accurate than that of [4]. In range 11–14 of Fig. 7 our scaler approximates the actual workload on the system. The algorithm of [4] overcompensates the innacurate prediction with the peaks at 12–13.

5 Related Work

Global scaling The strand of work closest to ours [5,7,9] introduced a kind of reactive global scaling that eliminates domino effects by design; building upon this we propose our proactive-reactive solution. More distant work on global scaling focus on smoothing, rather than removing, the domino effect (called "bottleneck shifts") as, e.g.: [6], which however only considers reactive scaling,

and [4], which we already discussed in detail in Section 4. Besides [4], also [8] mixes reactive and proactive. The main differences of [8] with our work are: (*i*) the microservice architectures considered in [8] disregard fork-join patterns, i.e. those accomplishing a task via parallel execution of pipelines as in the use case we consider (where the analysis of an email requires analysing its subparts, i.e., attachments, text body, etc...); (*ii*) the global scaling algorithm of [8] suffers from the domino effect in that, when global adaptation is triggered, the maximum number of replicas for each microservice is established only from metrics obtained by its local monitoring.

Local scaling Reactive and proactive local-scaling proposals abound in the literature (both pure and mixed) [15,16]. Recent examples of pure reactive local scaling include Bayesian Optimisation techniques [17] and Fuzzy Logic [18]. Researchers already followed this path for the case of local scaling—due in part to how susceptible local scaling is to domino effects [19]—, also proposing ways to mix the reactive nature of local autoscaling with *proactive* elements, e.g., by forecasting the incoming workload [16]. The proactive mode involves adopting future workload prediction techniques to create early scaling mechanisms. The prediction of future system load is usually addressed with probabilistic modelling frameworks or time series analysis techniques. Mathematical modelling of processes entails the usage of techniques such as Markov chains [20], model-checking or probabilistic time automata [21,22], enabling the analysis and anticipation of system behaviours. Time series analysis is a data-oriented technique that involves extracting relevant information from the behaviour of the studied system. The most commonly used techniques include machine-learning algorithms, such as k-means [23], neural networks [24,25]—which we also use to maximize accuracy and precision of our predictor.

Previous work also presented approaches that mixed reactive-proactive scaling at the local level. However, a significant difference between our hybrid global scaling approach and the local ones is that the latter, for the most part, exploit local metrics of the virtual machine hosting services (CPU, memory). As argued in Section 4, looking at local metrics is fine as long as we aim at local replications, however these metrics are not enough to account for functional dependencies between system requests and local requests to single microservices (which allow us to eliminate the domino effect). Thus, we cannot directly compare with the literature on hybrid local scaling and we only draw a coarse comparison. Previous work also presented local-scaling hybrid reactive-proactive systems. These improve system behaviour and effectively deal with unexpected traffic fluctuations, while simultaneously benefiting from the analytics-based proactive and reactive power of the system [24,26]. Industry-wise, the main platforms delivering Cloud services, e.g., Amazon and Google, offer integrated solutions for reactive local scaling of resources based on user thresholds or rules for adapting to the workload [27,28]. Recently, these platforms have introduced predictive capabilities in their systems [29], exploiting gathered historical information for automatic adaptation and orchestrating between reactive and proactive modalities. Following the hybrid approach, we developed our platform to accommodate both

reactive and proactive global modes and showed a possible implementation (see Section 4). Specifically, we favour analytics-based proactive scaling and pass control to the reactive modality when traffic fluctuations exceed some set accuracy threshold.

6 Conclusion and Future Work

We proposed a platform that can host both reactive and proactive global scaling and compared the analytics-based proactive and proactive-reactive scaling.

Low-hanging fruits from this work include both the introduction and refinement of analytics-based prediction and hybridisation 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).

Another direction towards using data analytics to help global scaling is helping DevOps in compiling the deployment constraints of the scaling plans (cf. Fig. 1). In this case, monitors would track how requests hop among the microservices of the observed architecture, and data-analytics techniques would provide hints for DevOps to quantify the multiplicative deployment factors among the microservices.

References

1. J. Humble and D. Farley, "Reliable software releases through build, test, and deployment automation," *Anatomy of Deployment Pipeline*, 2010.
2. T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of grid computing*, vol. 12, no. 4, pp. 559–592, 2014.
3. A. Gandhi, P. Dube, A. Karve, *et al.*, "Adaptive, model-driven autoscaling for cloud applications," in *11th International Conference on Autonomic Computing (ICAC 14)*, pp. 57–64, 2014.
4. B. Urgaonkar, P. J. Shenoy, A. Chandra, *et al.*, "Agile dynamic provisioning of multi-tier internet applications," *ACM Trans. Auton. Adapt. Syst.*, vol. 3, no. 1, pp. 1–39, 2008.
5. M. Bravetti, S. Giallorenzo, J. Mauro, *et al.*, "Optimal and automated deployment for microservices," in *FASE 2019*, pp. 351–368, Springer, 2019.
6. A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *2019 IEEE ICDCS*, pp. 1994–2004, IEEE, 2019.
7. M. Bravetti, S. Giallorenzo, J. Mauro, *et al.*, "A formal approach to microservice architecture deployment," in *Microservices, Science and Engineering*, pp. 183–208, Springer, 2020.
8. F. Rossi, V. Cardellini, and F. L. Presti, "Hierarchical scaling of microservices in kubernetes," in *ACSOS*, pp. 28–37, IEEE, 2020.
9. L. Bacchiani, M. Bravetti, S. Giallorenzo, *et al.*, "Microservice dynamic architecture-level deployment orchestration," in *COORDINATION 2021*, LNCS, Springer, 2021.
10. J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, *et al.*, "Serverless computing: One step forward, two steps back," in *CIDR 2019*, www.cidrdb.org, 2019.

11. J. D. Kelleher, B. Mac Namee, and A. D'arcy, *Fundamentals of machine learning for predictive data analytics: algorithms, worked examples, and case studies*. MIT press, 2020.
12. B. Klimt and Y. Yang, "The enron corpus: A new dataset for email classification research," in *Machine Learning: ECML 2004*, (Berlin), pp. 217–226, 2004.
13. L. Bacchiani, M. Bravetti, M. Gabbrielli, S. Giallorenzo, and S. P. Zingaro, "Repository of the datasets, testbed, and tests." github.com/LBacchiani/predictive-autoscaling, 2022.
14. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen, "Abs: A core language for abstract behavioral specification," in *FMCO 2010*, pp. 142–164, Springer, 2010.
15. Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, *et al.*, "Elasticity in cloud computing: state of the art and research challenges," *IEEE Trans. on Services Computing*, vol. 11, no. 2, pp. 430–447, 2017.
16. C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–33, 2018.
17. G. Yu, P. Chen, and Z. Zheng, "Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach," *IEEE Trans. Cloud Comp.*, 2020.
18. B. Liu, R. Buyya, and A. Nadjaran Toosi, "A fuzzy-based auto-scaler for web applications in cloud computing environments," in *ICSOC*, pp. 797–811, Springer, 2018.
19. N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *IEEE CLOUD 2011*, pp. 500–507, 2011.
20. G. A. Moreno, J. Cámara, D. Garlan, *et al.*, "Efficient decision-making under uncertainty for proactive self-adaptation," in *IEEE ICAC 2016*, pp. 147–156, 2016.
21. A. Naskos, E. Stachtiari, A. Gounaris, *et al.*, "Dependable horizontal scaling based on probabilistic model checking," in *IEEE/ACM CCGRID 2015*, pp. 31–40, 2015.
22. G. A. Moreno, J. Cámara, D. Garlan, *et al.*, "Proactive self-adaptation under uncertainty: a probabilistic model checking approach," in *ACM ESEC/FSE 2015*, pp. 1–12, 2015.
23. S. Dutta, S. Gera, A. Verma, *et al.*, "Smartscale: Automatic application scaling in enterprise clouds," in *2012 IEEE Fifth International Conference on Cloud Computing*, pp. 221–228, 2012.
24. N. Marie-Magdelaine and T. Ahmed, "Proactive autoscaling for cloud-native applications using machine learning," in *GLOBECOM 2020*, pp. 1–7, 2020.
25. J. Park, B. Choi, C. Lee, and D. Han, "Graf: a graph neural network based proactive resource allocation framework for slo-oriented microservices," pp. 154–167, 2021.
26. A. Bauer, V. Lesch, L. Versluis, A. Ilyushkin, N. Herbst, and S. Kounev, "Chamulteon: Coordinated auto-scaling of micro-services," in *ICDCS*, pp. 2015–2025, IEEE, 2019.
27. Amazon, "AWS Auto Scaling." aws.amazon.com/autoscaling, 2022.
28. Microsoft, "Overview of autoscale in Microsoft Azure." docs.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-overview, 2022.
29. Google, "Scaling based on predictions." cloud.google.com/compute/docs/autoscaler/predictive-autoscaling, 2022.