# From Reactive to Proactive Global Autoscaling

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, IT [2]INRIA, FR

**Abstract**

Modern Cloud autoscaling techniques scale microservice replicas to adapt to their inbound traffic. We recently theorised and evaluated a scaling technique called architecture-level, global scaling, able to scale the whole architecture w.r.t. an expected amount of traffic. Among its benefits, global scaling helps to prevent "domino" effects (cascading slowdowns) due to the unstructured scaling actions of microservice-level scaling.

The current notion of global scaling assumes a reactive behaviour w.r.t. traffic fluctuations, where the latter trigger the application of the related scaling plans.

Here, we challenge this reactive interpretation, and we propose a *proactive* extension of *global scaling* able to increase the efficiency of global scaling, by further lowering latency and message loss due to the runtime overhead of allocating virtual machines and replicas. Our contributions include the proposal of a platform for proactive global autoscaling and a preliminary benchmarking of the efficiency gain of proactive vs reactive global 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 [8] and autoscaling [11].

Although a powerful resource, autoscaling comes with its own challenges. As Ghandi et al. [6] put it:

> [...] while cloud computing offers flexible resource allocation, 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.

The common way to perform application autoscaling [11] focusses on the single component (e.g., a microservice) and applies an "horizontal" scaling, where the platform deploys new copies of the same component to withstand an increase of inbound requests to the component—and, vice versa, it shuts down redundant copies of the same microservice when idle, to save resources/money.

In a stand of work started in 2019 [3, 4, 2] we proposed the concept of architecture-level *global scaling* as an alternative to the application of the standard, localised (intended at the level of single component/microservice in an architecture) scaling. The motivation behind our proposal comes from the fact that localised scaling suffers from "domino" effects due to unstructured scaling actions that may cause cascading slowdowns or outages [7, 12]. Note that when discussing about scaling, we refer—except where otherwise specified—to the automatic trigger of the scaling strategy and we omit the prefix 'auto', without losing in specificity.

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 [3], we can compute deployment plans that:

- scale the whole architecture w.r.t. a considered, expected amount of inbound traffic;
- respect (if any) the constraints of resource allocation and dependency of the scaled microservices (e.g., to run 2 copies of $M_1$ we will have 2 pairs—4 in total—of $M_2$);
- 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.

Ideally, given a specification that accurately defines the constraints of resource allocation and dependency of the scaled microservices, the global approach would scale the system in similar ways as the local one minus the delays deriving from the domino effect.

## 2 From a Reactive to Proactive Global Scaling platform

While global scaling mitigates some problems of localised scaling, they both undergo inefficiencies due to their *reactive* nature. Indeed, both modalities trigger scaling as a reaction to fluctuations of inbound traffic. Unfortunately, in the always-available, responsive world of Cloud applications, the time spent in reacting to change goes to the detriment of customers, who can endure delays, downtimes, and transitory outages and receive a lower-than-expected level of service.

Researchers realised this problem early on [14], also due to how susceptible is local scaling to domino effects, and proposed ways to mix the reactive nature of local scaling with *proactive* elements, e.g., by forecasting the incoming workload [13].

Intuitively, also global scaling shall endure some performance inefficiencies due to its reaction overhead, which one could mitigate by switching to a proactive paradigm.

In this section, we provide two contributions. The first regards the presentation of a platform that DevOps exploit to perform *proactive global scaling*. In doing so, we do not start from scratch, and we build on previous work on global scaling, proposing a minimal modification able to capture proactive scaling and reusing most elements of the existing reactive global scaling platform from [3, 4, 2]. The second regards the empirical benchmarking of the inefficiencies of reactive global scaling w.r.t. an Oracle—i.e., an ideal omniscient predictor—that can proactively trigger the global scaling of the architecture to face future traffic.

We start from the description of the platform, whose elements we represent in Fig. 1 with continuous-line borders and mark with a dotted border the new element, Proactive Module, that supports the application of proactive scaling behaviours (we stripe the background of the Monitor component because we also slightly extend its behaviour in the new, proactive version of the platform, discussed when we describe the Proactive Module).

For completeness, we briefly describe all elements of the platform, although dedicating more space to the component for proaction.

In Fig. 1, we find two kinds of elements. The components found in the "cloud" are the elements of a given microservices architecture, labelled $G, M_1, M_2, M_3$, which we want to scale. The elements outside the cloud belong to the global scaling platform. In the following, we concentrate on the latter and use the architecture components in examples.

In the figure, we also show three kinds of flows. The one represented by continuous-line arrows $\rightarrow$ showing the traffic addressed to the microservice architecture. The one indicated by the dashed-line arrows $\dashrightarrow$ regards the runtime execution of global scaling. The thick arrow $\Leftarrow$ shows the compilation time of deployment plans.

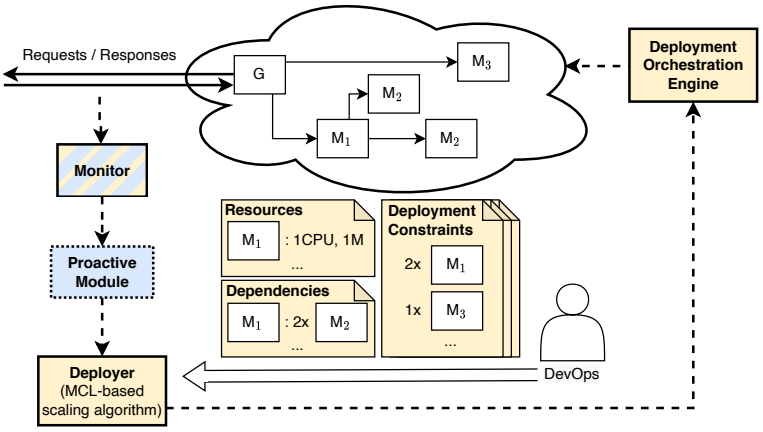


Figure 1: Representation of the Proactive Global Scaling platform.

We comment the components in Fig. 1 in anticlockwise order, leaving the new, proactive one for last.

**Deployment Orchestration Engine** The Deployment Orchestration Engine is an orthogonal component to the platform. The only requirement is that it should provide a programming interface for the application of deployment plans. Docker and Kubernetes are candidates for this role.

**Deployer** The Deployer—which implements the Maximum Computational Load (MCL) scaling algorithm and the deployment strategy proposed in [2]—regards two flows.

The first one, represented by $\Leftarrow$ , regards the computation of the deployment plans, applied by the scaling flow $\dashrightarrow$ . 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 respect 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—from now on, called *deltas*.

As previously mentioned, the second flow is that of the runtime scaling $\dashrightarrow$ , run alongside the inbound traffic $\rightarrow$ . In this case, the Deployer acts as service that other components can 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** The last component of the original platform for reactive global scaling is the Monitor, which observes the traffic reaching the deployed architecture and possibly triggers the application of deltas.

In its original formulation, the monitor tracks the traffic flowing on the architecture within prefixed ticks (e.g., a tick can be 10 seconds), it 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 size.

**Proactive Module** The last component, which we add to support proactive deployment, is the Proactive Module. The new component breaks the pre-existing, direct connection between the Monitor and the Deployer, taking the place of the former in determining the triggering of the Deployer.

The interesting point of our minimal change is that, in case we want to disable any proactive behaviour of the platform, we just need to set the Proactive Module in forward mode, so that it would pass the reactive triggers coming from the Monitor to the Deployer.

Instead, when active, the Proactive Module can act independently of the traffic, i.e., it can choose to follow or ignore the triggering of the Monitor. In Fig. 1, we striped the background of the Monitor to indicate that we slightly extended the behaviour of the Monitor for the proactive version of the global scaling platform, so that, besides the triggering signals, it also provides to the Proactive Module the information on the tracked traffic, which the latter can choose to consider when performing its task (e.g., to estimate the inbound traffic at a given time).

## 2.1 Benchmarking the performance gap between Reactive and Proactive Global Scaling

The benchmarking operations are carried out considering the Email Pipeline Processing System presented in [2]. In particular, the system is composed of 12 types of microservices, each one having its own load balancer. The latter is used to distribute requests over a set of instances that are incremented/decremented at need.

We perform our experiments to highlight the performance gap between Reactive and Proactive Global Scaling by modeling the system and scaling approaches via the ABS [1] programming language and executing it with the Erlang Backend (ABS is not directly executable, but it is compiled into Erlang). We take the system modeling done in [2] and we extend it with the Proactive Module described before: in the Reactive scaling approach the proactive power of such a module is set to 0, while in the Proactive one it acts as an oracle. Besides the adding of the Proactive Module, we also modify the way in which the system Monitor keeps track of the inbound workload for scaling operations: in [2] they take the average of messages arrived in a tick, while here we take the maximum. For example, let's consider a *tick* set to 10 seconds, we divide the *tick* in $X$ samples, we keep track of the number of inbound messages arrived for each sample and finally we consider the maximum value of them for computing scaling operations. Our approach turns out to be more conservative and less sensitive to irregular inboud workloads. We decide to run our experiments using a part of an IMAPS email traffic similar to that in [10] (accounting for the fact that here email attachments are also considered). We implement such inbound workloads by means of an *email generating service*.

In our experiments, we focus on: $(i)$ latency, $(ii)$ message loss, $(iii)$ costs and $(iv)$ deployed instances. Since by construction our Oracle never loses any message and the amount of cost and deployed instances is the same but simply shifted by 1 *tick*, the most interesting aspect deployment orchestration, scale configuration, global adaptation latency, thus here we only show the latency comparison results. We consider the latency as the average time for completely processing an email that enters the system. As can be seen from Figure 2, the Oracle is almost perfect but it does not have 0 latency (for example see $t_{12} - t_{14}$) as one might expect. This is caused by the fact that our Oracle is not capable of foreseeing the email structure, e.g. it does not know the actual number of attachments, and this may cause a small overloading of some services. Moreover, the Oracle can anticipate of 1 *tick* the scaling operations, but it does not take into account the amount of time that a virtual machine needs to be ready to work, i.e. the startup time, causing a small delay in the adaptation.

To give an intuition of the the performance gap between the Reactive global scaling and the Oracle, we compute the area under their latencies (AUC) by using the composite trapezoidal rule. The results we get, $AUC_{global} = 31s$ and $AUC_{oracle} = 0.4s$, indicate that the limitations of our Oracle are negligible. Moreover, such $AUCs$ suggest that a generic proactive global scaling approach can bring a significant improvement of the performances.

## 2.2 Discussion & Conclusion

We present a comparison between a Reactive global Scaling and an ideal (Oracle) Proactive Scaling approach. As shown by our empirical results the Oracle outclass the reactive approach thanks to its (almost) perfect proactiveness. However, implementing an Oracle like the one used here in real world scenarios is, obviously, not feasible. Nevertheless, the significant gap between the Reactive and Proactive paradigm performances suggests that even if we do not have a perfect Proactive Module for workload prediction we can still obtain significant improvements.

Concretely, the proactive module can be implemented in different ways; for instance, we see the usage of AI-based solutions for workload prediction promising [9, 15]. On one side, we would like to further our research to empirically evaluate the difference between local and global scaling (both reactive and proactive), possibly using open source microservice benchmarks [5]. Of course, since real-world implementations of the proactive module can fail to predict the actual traffic shape, another direction for future work is the integration between the reactive and proactive approaches —for example, by monitoring how close the prediction is of the current workload and set thresholds to switch the control from the proactive to the reactive mode.

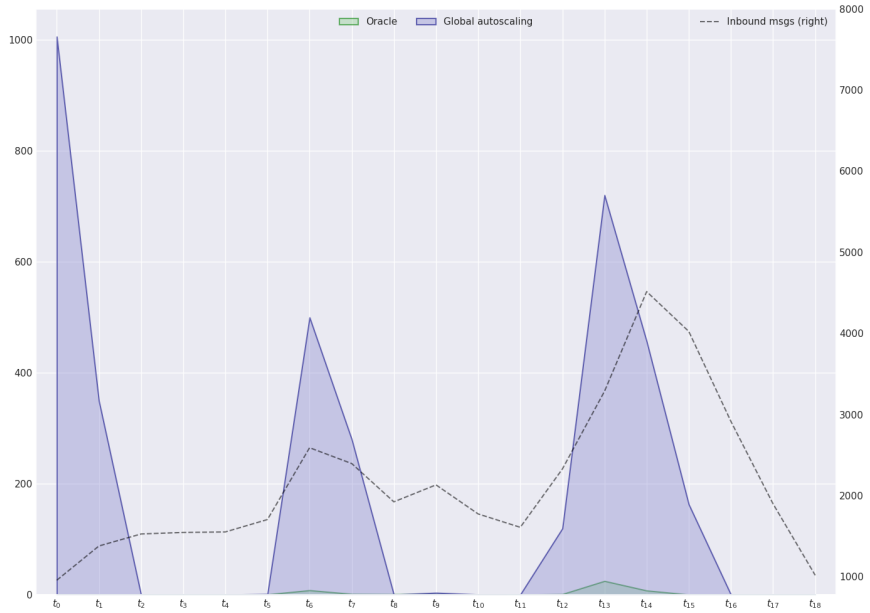


Figure 2: Latency(ms) Comparison.

# References


[1] Abs. ABS documentation. http://docs.abs-models.org/.

[2] L. Bacchiani, M. Bravetti, S. Giallorenzo, J. Mauro, I. Talevi, and G. Zavattaro. Microservice dynamic architecture-level deployment orchestration. In F. Damiani and O. Dardha, editors, *Coordination Models and Languages - 23st IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Proceedings*, Lecture Notes in Computer Science. Springer, 2021.

[3] M. Bravetti, S. Giallorenzo, J. Mauro, I. Talevi, and G. Zavattaro. Optimal and automated deployment for microservices. In *Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, pages 351–368. Springer, 2019.

[4] M. Bravetti, S. Giallorenzo, J. Mauro, I. Talevi, and G. Zavattaro. A formal approach to microservice architecture deployment. In *Microservices, Science and Engineering*, pages 183–208. Springer, 2020.

[5] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[6] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. Adaptive, model-driven autoscaling for cloud applications. In *11th ICAC*, pages 57–64, 2014.

[7] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[8] J. Humble and D. Farley. Reliable software releases through build, test, and deployment automation. *Anatomy of Deployment Pipeline*, 2010.

[9] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. V. Papadopoulos, B. Ghit, D. H. J. Epema, and A. Iosup. An experimental performance evaluation of autoscaling policies for complex workflows. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, pages 75–86, 2017.

[10] M. Karamollahi and C. Williamson. Characterization of IMAPS email traffic. In *27th IEEE MASCOTS, Rennes, France, October 21-25, 2019*, pages 214–220. IEEE Computer Society, 2019.

[11] 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*, 12(4):559–592, 2014.

[12] S. Mccombs. Outages? downtime? https://sethmccombs.github.io/work/2018/12/03/Outages.html, 2018. Accessed on Mar. 2022.

[13] C. Qu, R. N. Calheiros, and R. Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR)*, 51(4):1–33, 2018.

[14] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507. Ieee, 2011.

[15] L. Versluis, M. Neacşu, and A. Iosup. A trace-based performance study of autoscaling workloads of workflows in datacenters. In *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2018, Washington D.C., USA, May 1-4, 2018*, pages 1–10, 2018.