Towards Quantifiable Boundaries for Elastic Horizontal Scaling of Microservices

Manuel Ramírez López
Zurich University of Applied Sciences
School of Engineering, Service Prototyping Lab
(blog.zhaw.ch/icclab/)
Winterthur, Switzerland
ramz@zhaw.ch

Josef Spillner
Zurich University of Applied Sciences
School of Engineering, Service Prototyping Lab
(blog.zhaw.ch/icclab/)
Winterthur, Switzerland
josef.spillner@zhaw.ch

ABSTRACT

One of the most useful features of a microservices architecture is its versatility to scale horizontally. However, not all services scale in or out uniformly. The performance of an application composed of microservices depends largely on a suitable combination of replica count and resource capacity. In practice, this implies limitations to the efficiency of autoscalers which often overscale based on an isolated consideration of single service metrics. Consequently, application providers pay more than necessary despite zero gain in overall performance. Solving this issue requires an application-specific determination of scaling limits due to the general infeasibility of an application-agnostic solution. In this paper, we study microservices scalability, the auto-scaling of containers as microservice implementations and the relation between the number of replicas and the resulting application task performance. We contribute a replica count determination solution with a mathematical approach. Furthermore, we offer a calibration software tool which places scalability boundaries into declarative composition descriptions of applications ready to be consumed by cloud platforms.

CCS CONCEPTS

• **Networks** → **Cloud computing**; • **Software and its engineering** → *Automatic programming*; • **Computing methodologies** → Parallel algorithms;

KEYWORDS

microservices; scalability; replication; optimization

ACM Reference format:
Manuel Ramírez López and Josef Spillner. 2017. Towards Quantifiable Boundaries for Elastic Horizontal Scaling of Microservices. In *Proceedings of UCC'17: 10th International Conference on Utility and Cloud Computing Companion, Austin, Texas, USA, December 5–8, 2017 (UCC'17 Companion),* 6 pages.
https://doi.org/10.1145/3147234.3148111

1 INTRODUCTION

Composite microservice architectures are among the most popular architectural underpinnings for cloud-native applications [11]. Nowadays, a lot of companies decompose their monolithic applications into coupled, often containerized, sets of microservices. New software is being designed and developed following this architecture as well, benefiting from its advantages [6]. Fast horizontal scalability is one of the most important strengths of a microservices architecture. In this scaling model, each microservice can be scaled out by creating new instances which are placed separately according to the associated load, in contrast to monolithic applications. Another strength is the design for failure by strict separation of stateful and stateless services in which the stateless ones can be respawned at any time without having to consider violations of data characteristics such as availability or consistency [5, 7]. State is either kept in carefully designed and implemented stateful microservices or in centralized services outside of the application scope which are dynamically bound through service brokers [1].

While applications can be scaled vertically by allocating more resources to a running instance of a microservice, the dominant mechanism found in production cloud computing environments remains the mentioned horizontal scaling in which replicas of the same service are added and removed on demand. The separated scaling behavior provides many possible combinations of microservices concerning the number of replicas of each service, particularly for large applications. Formally, n service types s_j ($1 \leq j \leq n$) represented in a dependency-spanning type graph ($TG := \{s_1, \ldots, s_n\}$) are leading to m possible instance graphs ($IG_k := \{s_1 \times i_{k1}, \ldots, s_n \times i_{kn}\}; 1 \leq k \leq m$) in which each service type s_j is instantiated i_{kj} times. Evidently, for any specific application task triggered through a service call or a complex transaction, not all of these combinations are optimal in regard to a specific characteristic or constraints such as performance and cost. Often it is non-trivial to know in advance which combination is best for a given target of performance and an associated set of resources and how to scale out and in if the demand on services increases or decreases. Because of the inherent uncertainty, most application platforms provide automation in the form of horizontal auto-scaling features [9]. These reactive autoscalers help to adjust the performance of a set of microservice instances automatically depending on some metrics like processor load, memory utilization or network traffic.

The difficulty with this approach is twofold. First, the application provider will have to manually tweak several combinations of metrics to determine empirically a suitable configuration of initial

This research has been funded by the Swiss Commission for Technology and Innovation (CTI) in project ARKIS/18992.1 and MOSAIC/19333.1. It has also been supported by a Microsoft Azure Research Award and a Google Cloud credit.

UCC'17 Companion, December 5–8, 2017, Austin, Texas, USA

ACM ISBN ISBN 978-1-4503-5149-2/17/12…$15.00
https://doi.org/10.1145/3147234.3148111

scaling factors and metric thresholds. Second, most auto-scaling systems are simplistic in the sense of treating all microservices equally without the consideration of dependencies or specificities including resource quotas [4, 10].

The contribution of this paper is to mathematically express boundaries on how compositions of microservices (auto-)scale depending on whether they are stateful or stateless, reducing m possible instance graphs to a single optimal one, IG_o, per pre-defined task-specific workload. The paper recalls the limits of auto-scaling and demonstrates a unique solution to scale a microservices architecture within well-defined boundaries in which each single scale-out decision contributes positively to the overall performance. The theoretic part of the contribution is a mathematical method which finds the optimal scale combination for an application architecture depending on specified requirements on performance, price and available resources. The practical part of the contribution is a software which calibrates the scaling behavior by combinatorial tests followed by an injection of the desirable scaling rules into a format understood by platform-level microservice scalers such as Docker-Compose, Kubernetes or OpenShift for containerized composite applications.

This paper is organized as follows. The next section explains the motivation behind our research work and identifies the research question. Subsequently, a mathematical formula to determine scaling factors i_{kj} and an algorithm to instantiate the formula are presented and evaluated in Sections 3 & 4. The penultimate section describes limitations and future work and the last one concludes with a summary and an outlook on practical use.

2 MOTIVATION AND PROBLEM STATEMENT

The expected or desired behavior when scaling a microservice is a directly proportional relationship between the number of replicas and the performance. We consider a system with three-dimensional scaling on the X axis (horizontal duplication), Y axis (functional decomposition) and Z axis (data partitioning) according to the scaling cube. When a replica is added to L existing replicas ($L > 0$), the performance should grow proportionally to $\frac{L+1}{L}$. The actual behavior on service hosting platforms is usually more similar to the representation in Fig. 1 where at some point the gradient of the performance line is decreasing. Depending on the circumstances, it can fall to zero. It can even become negative, resulting in worse performance with more replicas, and furthermore it can be limited on the X axis by instance quotas. A first explanation of the decline would be the limited parallelization benefit described by Amdahl's law [2]. The behavior could also occur, for example, because beyond this point the replicas are stealing resources from other microservices which are on the critical performance path in resource-constrained environments [12]. Despite seemingly infinite resources in public clouds, both instance and resource quotas are commonly faced by application providers [13]. Another reason is the ineffective scaling of microservices which are not the bottleneck of the application. In these cases, the scaling leads to just wasting resources (in commercial environments, this means wasting money) to instantiate more replicas without yielding better performance.

Therefore, not always is scaling out a single overloaded microservice the solution to solve a performance problem. It is crucial to know the best combination of microservice instances for the current workload of an application and which combination is the best one when needing to scale out or in cost-effectively and cost-predictably.

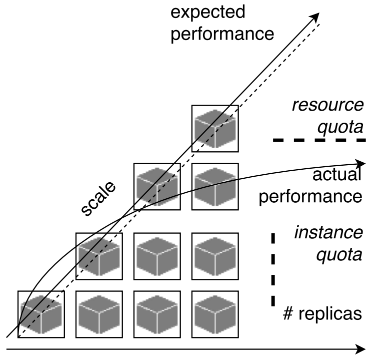

Figure 1: Number of replicas vs performance

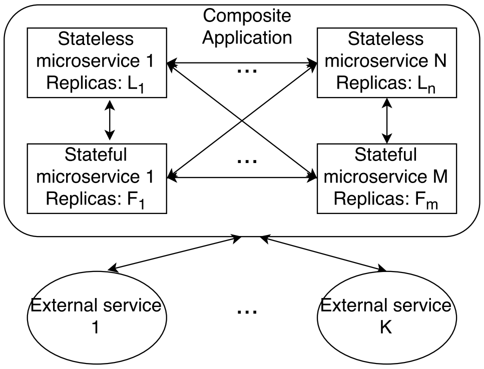

Figure 2: Microservices architecture of a single application

Fig. 2 expresses in a simple diagram how a microservice architecture with $N + M$ microservices and K external services, each one with a different number of replicas, is composed. To scale an application out effectively, one must first locate the bottleneck and scale it in a bounded way. When the bottleneck is in a third-party service outside of the scope of scaling, for instance a hosted database service (DBaaS), this dependency constraint must likewise be detected, for instance through formalized dependency operators [8].

Following the motivation to resolve the mentioned problems to ease application onboarding into cloud environments, we have identified the following two initial questions which trigger the solution approach by providing clarity about the current state of technology in scaling.

- Assuming an implementation of microservices as containers, do all containers scale with comparable characteristics? Are there identifiable patterns concerning stateful microservices?
- Does auto-scaling solve the mentioned scaling problems for complex applications? What happens if the bottleneck is in microservices which do not auto-scale at all or not well due to missing rules, or in third-party services whose scaling behavior cannot be influenced?

Once the knowledge about scaling is consolidated, the solution approach leads to the answer of the central research question:

- Can the best combination of replicas for a given application and workload be calculated for performance-critical and cost-constrained settings?

Our method to find the answer is based on a formalization of the application, a task and workload for it, its environment and a number of scaling constraints. Using a formalized mathematical model, matrix calculations across all combinations of scaling factors i_{kj} are performed and result vectors with optimal scale factors $\{i_{o1}, \ldots, i_{on}\}$ for each target optimum instance graph IG_o are obtained. Beyond the calculation, our solution approach includes the automatic configuration of composite microservice applications for targeted deployments into contemporary cloud platforms.

3 FORMALIZATION OF APPLICATION SCALABILITY

3.1 Concepts and Nomenclature

We define some important concepts and nomenclature which are needed first to encompass the research question. Specifically, we define:

(1) Node: A single machine, usually a virtual machine in the cloud, exposing finite compute, storage and networking resources.
(2) Cluster: A programmable set of nodes which appears like a single machine to an application.
(3) Application: A set of composite microservices.
(4) Maximum of replicas: Maximum number of replicas of each microservice which a cluster supports through its cumulative resources.
(5) Combination: Number of replicas for each microservice running in the cluster as constituents of the application composition. Let m be the number of microservices, with a maximum of replicas $0 < n_1, n_2, \ldots, n_m$ for each service. This gives a total of $n_1 \times n_2 \times \cdots \times n_m$ possible combinations.
(6) Experiment: Set of sequential or parallel requests to an application that simulates a typical use case, resulting in a unique workload.
(7) Makespan (μ): Time in seconds which a combination needs to finish an experiment.
(8) Performance (p): The inverse of the makespan.
(9) Cost of a microservice (κ): Number indicating the resource cost of running a single replica of this microservice in the cluster. For instance, the number of CPU cycles a replica consumes.
(10) Cost: Total resource cost of an experiment, formalized as $\sum_{i=1}^{m} \kappa_i \times n_i$.
(11) Maximum of cost (max_κ): The maximum admissible cost.
(12) Minimum of performance (min_p): The minimum performance that the scenario permits.
(13) Maximum of makespan (max_μ): The maximum value of makespan (experiment duration) that the scenario permits; the inverse of min_p. Note: $p > min_p \iff \mu < max_\mu$ holds. Therefore satisfying the performance constraints is equivalent to satisfying the makespan constraints.

3.2 Research Question Examined

With these concrete definitions, we reformulate the central research question to the more concise form: What is the best combination of replicas for a given application on a given cluster with a given workload? What, eventually, can be even considered a best combination? There is no deterministic answer to this question as it depends on the use cases, in other words, specific application tasks and associated workloads. For simplicity, we define best as either of the three: minimum makespan μ, minimum cost κ, and maximum weighted utility defined by $\frac{1}{\lambda_1\mu+\lambda_2\kappa}; \mu, \kappa, \lambda > 0$. Our approach thus answers the central research question narrowly for the following scenarios: What is the most economical combination satisfying minimum performance constraints? What is the fastest combination satisfying maximum price constraints? What is a sensible trade-off with high compromise utility involving both metrics? The approach to find these answers is visualized in Fig. 3. Up to three optimal instance graphs IG_o emerge from a microservice type graph TG for a given workload.

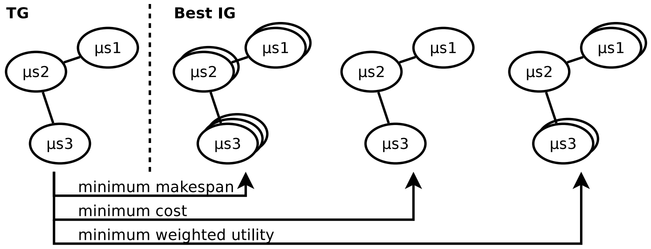

Figure 3: Solution approach to the research question

3.3 Mathematical Model and Formula

3.3.1 Fastest and cheapest combination. Let c be the cluster with unspecified number of nodes. Let $0 < m$ be the number of microservices and $n_1, n_2, \ldots, n_m$ the maximum number of replicas for each microservice in c. The first step is to define an experiment, e, the use case with the workload expected for the application. For the experiment e in the cluster c, for each combination $comb_i$ where $comb_i = i = (r_1, r_2, \ldots, r_m)$ with $0 < r_1, r_2, \ldots, r_m <= n_1, n_2, \ldots, n_m$ there is a resulting makespan μ_i, i.e., $e(c, comb_i) \rightarrow \mu_i$. We then create an (m)-dimensional matrix M_e with the makespan for all the combinations. We call this matrix the makespan matrix or matrix of performance. The following representation shows the first two out of (m) dimensions, corresponding to systematic combinations of instance counts for the first two microservices in a composition.

$$M_e = M(e)_{n_1 \times \cdots \times n_m} = \begin{pmatrix} \mu_{1,1,\ldots} & \mu_{1,2,\ldots} & \cdots & \mu_{1,n_2,\ldots} \\ \mu_{2,1,\ldots} & \mu_{2,2,\ldots} & \cdots & \mu_{2,n_2,\ldots} \\ \cdots & \cdots & \cdots & \cdots \\ \mu_{n_1,1,\ldots} & \mu_{n_1,2,\ldots} & \cdots & \mu_{n_1,n_2,\ldots} \end{pmatrix}$$

In analogy, the cost of a combination is $cost(comb_i, prices) = \sum_{s=0}^{m} r_s \times prices_s$ where $prices$ is a tuple with the operating cost of the microservices (κ). Only the resource cost is considered in this formula because the monetary cost is typically offset by a free tier in commercial environments and other discount schemes which hide the actual cost. With e, M_e and $cost$ there are three parameters to influence the solution. Now, we define the fastest solution as:

$$fastest(M_e) = i \mid \min_{\forall m_i \in M_e} m_i$$

A solution is trivially the fastest one if a certain combination of instances requires the least time to complete a fixed workload. Often, but not always, this implies a high number of replicas. Obviously, the cheapest solution in contrast corresponds to the minimum number of microservices, i.e, $1, 1, \ldots, 1$. Extending these formulas with the constraints of performance and cost yields: Let I be the set of indices of the makespan combination matrix M_e.

$$\begin{aligned} &fastest(M_e, prices, max_\mu, max_\kappa) \\ &\quad = i \mid \min_{\forall i \in I}\{m_i \in M_e \mid m_i < max_\mu, \\ &\qquad cost(i, prices) < max_\kappa\} \end{aligned} \tag{1}$$

$$\begin{aligned} &cheapest(M_e, prices, max_\mu, max_\kappa) \\ &\quad = i \mid \min_{\forall i \in I}\{cost(i, prices) \mid M_e \ni m_i < max_\mu, \\ &\qquad cost(i, prices) < max_\kappa\} \end{aligned} \tag{2}$$

3.3.2 Rate for almost-optimal combinations. With the previous formulas one obtains the fastest or the cheapest solution which satisfies all constraints, but they calculate the solution quite rigidly with respect to the policies. A more practical method is to consider almost-optimal combinations. They occur if the second-fastest solution is fast enough but quantitatively a lot cheaper, or cheap enough but a lot faster, respectively. With the previous formulas these solutions will be skipped. Therefore, we have added the rate parameter. With it we will obtain the solutions ordered by the corresponding other policy which we have chosen as long as they are close enough, or inside the rate, to be a valid solution. In case the strict solution is desired, this parameter must just be set to 1.0.

First, let us define the next orders:

$$\begin{aligned} \prec_{cost} := \forall i, j \in I, i < j \iff & cost(i, prices) < cost(j, prices) \\ & \text{or } (cost(i, prices) = cost(j, prices) \text{ and } m_i < m_j) \end{aligned}$$

$$\begin{aligned} \prec_{perf} := \forall i, j \in I, i < j \iff & m_i < m_j \\ & \text{or } (m_i = m_j \text{ and } cost(i, prices) < cost(j, prices)) \end{aligned}$$

Using these orders we define the improved formulas:

$$\begin{aligned} &fastest\_rate(M_e, prices, max_\mu, max_\kappa, rate) \\ &\quad = i \mid \min_{\forall i \in I}(i \mid \frac{m_i}{m_k} \leq rate, \prec_{cost}) \\ &\quad \text{where } k = fastest(M_e, prices, max_\mu, max_\kappa) \end{aligned} \tag{3}$$

$$\begin{aligned} &cheapest\_rate(M_e, prices, max_\mu, max_\kappa, rate) \\ &\quad = i \mid \min_{\forall i \in I}(i \mid \frac{cost(i, prices)}{cost(k, prices)} \leq rate, \prec_{perf}) \\ &\quad \text{where } k = cheapest(M_e, prices, max_\mu, max_\kappa) \end{aligned} \tag{4}$$

4 PRACTICAL EXAMPLES AND EVALUATION

4.1 Experiments Setup

With the following experiments, we encompass all the cases studied until now in this paper. The scenario application under observation implements a microservices architecture with one stateful and one stateless microservice as containers. We will scale the stateless service along the X axis and the stateful one along both the Z and X axes (as shown in Fig. 4). For each service we state if it makes sense to use auto-scaling or not. We apply the bounded scaling formula to find the best solution.

The example is inspired by the design of online document management applications where the stateful microservice is a MongoDB database of documents in which each document belongs to a user (tenant). Connected to it is the stateless microservice as a CRUD layer for documents, with the option to search patterns in the documents as well. This service provides a REST API implemented in Python. The data used in these experiments is an array of JSON structures which represent the documents and can be generated using a proportioned script.

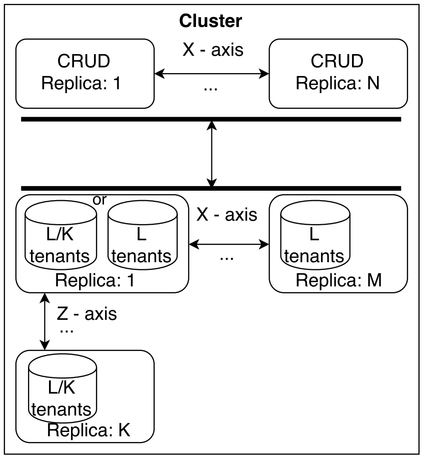

Figure 4: Scaling axes and instance counts of the scenario application composed of database and CRUD/search containers

4.2 Experiments Approach and Open Science Notebook

For general reproducibility and recomputability of our results, we have created simultaneously to the experiments an open science notebook[1]. It describes the details of hardware and software used in each experiments, the prototypical software implementation, the datasets and the instructions to reproduce each experiment with the respective reference results.

4.3 Implementation of the Formula

There are two steps required to obtain the solution which are linked by a performance matrix. A third step is required to automate

[1] Open Science Notebook: http://osf.io/6gup8/

the configuration of the application orchestrators and scalers. We have implemented three linkable software prototypes in Python to automate the entire process of boundary detection.

Step 1. The first step involves the generation of the performance matrix by measuring combinatorial benchmarks. These benchmarks need to be implemented a-priori. The script runs the experiment for each scaling combination automatically if it is possible or waits for the user if it detects a manual scaling requirement. The result is the matrix filled for use with the formula.

Step 2. In the second step, the formula implementation receives the performance matrix as well as the cost of the microservices, the constraints of performance and cost, the rate (1.0 by default) and a policy (fastest or cheapest). It returns the best combination for these parameters.

Step 3. The replica count for each microservice type is annotated on the type graph *TG* whose representation is a Docker Compose file or a set of Kubernetes deployment files. The implementation manages both formats through auto-detection. In the case of Kubernetes, an exemplary configuration excerpt in JSON format is shown in Listing 1. The number of replicas is given as placeholder REPLICAS in the listing.

Listing 1: Replica count in Kubernetes

```
{
 "kind": "Deployment",
 "apiVersion": "extensions/v1beta1",
 "metadata": {
  "name": "MICROSERVICE",
 },
 "spec": {
  "replicas": REPLICAS,
   "spec": {
    "containers": [
     {
      "name": "MICROSERVICEIMPL",
      "image": "NAMESPACE/CONTAINER:1.1",
...
```

4.4 Experiments Results

We have deployed the scenario service on a 6-node Kubernetes cluster on the Google Cloud Platform. In the same geographic zone, we have further deployed a 3-node Kubernetes cluster to run the experiment code. The execution is initiated by a set of requests to the REST API implemented in the stateless microservices. The dominant observation is that the stateless microservice can just scale on the X axis and the stateful microservice on the X and Z axes. By examining both cases, we obtain different matrices. In these matrices, the number of replicas of the stateful microservice is represented in the rows and the number of replicas of the stateless one in the columns. For instance, if M is the matrix then if $a_{ij} \in M$, a_{ij} is the performance of the experiment for i replicas of the stateful and j replicas of the stateless service.

Scaling on the X axis. We scaled MongoDB on the X axis using the Kubernetes features StatefulSet and StorageClass. This combination creates a cluster of MongoDB instances where each node maintains a full replica of the database and only one of them acts as master. The experiment consists of 10 million document insertions. It represents a typical use case for document batch processing in the cloud. For this use case, the following is the partially obtained matrix M_1. All values μ_{ij} are specified in seconds.

$$M_1 = \begin{pmatrix} 3379.0 & * & 2960.5 & * & 3501.6 & * & 3040.5 \\ * & * & * & * & * & * & * \\ 3263.0 & * & 3602.2 & * & 3365.9 & * & 3263.4 \end{pmatrix} \quad (5)$$

There is no improvement of the performance when the stateful microservice is scaled out ($\Delta_{max} = 641.72$). This observation can be explained because the MongoDB cluster is intended to improve high availability, resilience and distribution of the database in different geographic zones, instead of performance [3]. One can use the auto-scaling option of Kubernetes for adding a maximum scaling factor to avoid scaling in vain when the bottleneck is in the other (stateful) microservice. The experiments numbered {1,. . . ,8} in the open science notebook are related to this example.

Scaling on the Z axis. We scaled the MongoDB microservice on the Z axis by separating the tenants in the replicas of this microservice. This approach results in having L tenants and K replicas of the service, each one having L/K tenants. In the experiment, we set $L = 100$ and $K \in \{1, 2\}$. Running the experiment for 30000 document insertions we obtained the next matrix M_2. The experiments numbered {9,. . . ,20} in the open science notebook are related to this example.

$$M_2 = \begin{pmatrix} 89.2 & * & 45.5 & * & 43.8 & * & 41.9 & * & 42.1 & * & 40.5 \\ 71.7 & * & 48.1 & * & 40.1 & * & 35.9 & * & 36.1 & * & 36.4 \end{pmatrix} \quad (6)$$

Having the performance matrices as input, the formula to determine the best combination is applied. It is parameterized with the matrix M_2 with the tuple of prices: (1/4,1/12). In the most expensive combination $c_1 = (2, 6)$ the cost is 1, i.e, $cost(c_1, prices) = 1$. Table 1 collects the best solutions for the different options of the formula. The example reveals the fastest and cheapest option which satisfies the performance and cost constraints: $(2, 3)$ and $(1, 3)$. Adding a small rate, we can find another quite similar solution for the policy parameter and improving considerably the other one. For example, for the fastest with-rate option, the cost is reduced to 77% while increasing the makespan to only 104% (lines 3 and 5 in the table).

The calculated number of replicas is then injected into the static deployment configuration (implemented for Kubernetes descriptors) or into the dynamic scaling manager, both of which requires a labelling of services as stateful or stateless (using Kubernetes object labels).

5 LIMITATIONS AND FUTURE WORK

Depending of the workload of the application we choose a combination of scaling factors. We can simulate the workload we expect with an experiment although this combination is fixed to this specific workload. But, usually, the workload of an application is variant. Hence, a new research question becomes evident: Is it possible to know which is the best combination for each workload? One option is to perform a number of experiments corresponding to different

Table 1: Optimal instance count results

Title	Policy	max_μ	max_κ	Rate	#S-ful	#S-less	Cost	Makespan
baseline	fastest	X	X	X	2	7	0.83	35.92
baseline	cheapest	X	X	X	1	1	0.33	89.16
with C	fastest	45.0	0.8	X	2	5	0.75	40.07
with C	cheapest	45.0	0.8	X	1	5	0.5	43.79
with C&R	fastest	45.0	0.8	1.06	1	7	0.58	41.88
with C&R	cheapest	45.0	0.8	1.2	1	7	0.58	41.88

Legend: S-less = stateless, S-ful = stateful, C = constraints, R = rate

workloads the application handles. Yet, encompassing the different possibilities is a lot work and in some cases not feasible. Another option is find a ratio relation between the number of replicas of the different microservice and scale the microservices according to this relation.

Not all microservices are functionally connected. We can represent a microservice architecture as a bi-directed disconnected graph, where the vertices are the microservices and the edges represent the connection between the microservices. The first step is then splitting the graph into multiple connected graphs without interconnections as exemplified in Fig. 5 with the microservices $A - G$.

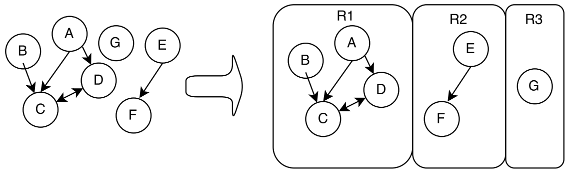

Figure 5: Graph decomposition for composite microservice dependencies

Future work will include the definition of a formula to find the relation for each of the bi-directed connected graphs which compose the microservice architecture. This relation will be defined for different reasons: performance motives, in this case, the dependency between the microservice marks the number of replicas of each one, and for external constraints, for example, each replica of the main microservice must possess a replica of a sidecar microservice. For a set of n connected graphs there will be $r_1, r_2, \ldots, r_n$ relations defining how the application scales.

6 SUMMARY AND CONCLUSIONS

With the versatility and the power a microservices architecture offer for application scalability, the necessity to control it and use it properly also emerges. We have explored this characteristic, studying the scaling of different microservices. We analyzed existing tools which help to scale as auto-scalers, reported on their limits and driven by the necessity to overcome it contributed a method to overcome them and a practical cloud platform tool which configures scaling of containers.

The use of persistent data marks a different scaling behavior. We have given examples to demonstrate the scaling without further quantification to achieve the answer to the clarity questions. This same characteristic affects auto-scalers which also have other limitations that lead them to scale services which do not need to scale further, answering the other clarity question. Trying to solve this issue, we created a method to know which one is the best combination of microservices in an application for given expectations and resources. For this purpose we tested the possible scaling factor combinations using an experiment which simulates typical loads. The workflow creates a matrix of performances which is further analyzed and filtered with the given requirements, using a formalized calculation, to obtain the solution closest to the expectations. Thus, the answer to the research question is systematically exposed.

We make our experiments, datasets and code available through an open science notebook and encourage further work reproducing or countering our results.

REFERENCES

[1] Doug Davis, Morgan Bauer, Matt McNeeney, Shannon Coen, Paul Morie, and Ville Aikas. 2017. Open Service Broker API v2.12. specification online at https://github.com/openservicebrokerapi/servicebroker/blob/v2.12/spec.md. (June 2017).

[2] Fernando Díaz del Río, Javier Salmerón-Garcia, and José Luis Sevillano. 2016. Extending Amdahl's Law for the Cloud Computing Era. *IEEE Computer* 49, 2 (2016), 14–22. https://doi.org/10.1109/MC.2016.49

[3] Sandeep Dinesh. 2017. Running MongoDB on Kubernetes with StatefulSets. online: http://blog.kubernetes.io/2017/01/running-mongodb-on-kubernetes-with-statefulsets.html. (January 2017).

[4] Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Herbst, Alessandro Vittorio Papadopoulos, Bogdan Ghit, Dick H. J. Epema, and Alexandru Iosup. 2017. 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.* 75–86. https://doi.org/10.1145/3030207.3030214

[5] Martin Kleppmann. 2015. A Critique of the CAP Theorem. *CoRR* abs/1509.05393 (2015). http://arxiv.org/abs/1509.05393

[6] Nane Kratzke and Peter-Christian Quint. 2017. Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. *Journal of Systems and Software* 126 (2017), 1–16. https://doi.org/10.1016/j.jss.2017.01.001

[7] Wubin Li, Petter Svärd, Johan Tordsson, and Erik Elmroth. 2013. Cost-Optimal Cloud Service Placement under Dynamic Pricing Schemes. In *IEEE/ACM 6th International Conference on Utility and Cloud Computing, UCC 2013, Dresden, Germany, December 9-12, 2013.* 187–194. https://doi.org/10.1109/UCC.2013.42

[8] Mats Neovius, Luigia Petre, and Kaisa Sere. 2015. A Theory of Service Dependency. In *Proceedings 17th International Workshop on Refinement, Refine@FM 2015, Oslo, Norway, 22nd June 2015.* 112–128. https://doi.org/10.4204/EPTCS.209.9

[9] Juan Marcelo Pintos, Carlos Nunez Castillo, and Fabio López-Pires. 2016. Evaluation and comparison framework for platform as a service providers. In *XLII Latin American Computing Conference, CLEI 2016, Valparaíso, Chile, October 10-14, 2016.* 1–11. https://doi.org/10.1109/CLEI.2016.7833384

[10] T. Vondra, Jan Sedivý, and J. M. Castro. 2017. Modifying CloudSim to accurately simulate interactive services for cloud autoscaling. *Concurrency and Computation: Practice and Experience* 29, 10 (2017). https://doi.org/10.1002/cpe.3983

[11] Hulya Vural, Murat Koyuncu, and Sinem Guney. 2017. A Systematic Literature Review on Microservices. In *Computational Science and Its Applications - ICCSA 2017 - 17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part VI.* 203–217. https://doi.org/10.1007/978-3-319-62407-5_14

[12] Ke Wang, Xiaobing Zhou, Kan Qiao, Michael Lang, Benjamin McClelland, and Ioan Raicu. 2015. Towards Scalable Distributed Workload Manager with Monitoring-Based Weakly Consistent Resource Stealing. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2015, Portland, OR, USA, June 15-19, 2015.* 219–222. https://doi.org/10.1145/2749246.2749249

[13] Qian Zhu and Gagan Agrawal. 2012. Resource Provisioning with Budget Constraints for Adaptive Applications in Cloud Environments. *IEEE Trans. Services Computing* 5, 4 (2012), 497–511. https://doi.org/10.1109/TSC.2011.61