Benchmark Evaluation of Scale-Up and Scale-Out Techniques on a Functional Programming Based Microservice

Arnold Attila Higyed

Department of Computer and Information Technology
Politehnica University Timisoara
Timisoara, 300223, Romania
higyedarnold@gmail.com

June 8, 2020

***Abstract*— When data grows too large, the tendency is to either scale-out, by adding nodes to the system or to scale-up, by adding resources to a single node. It is well known that these techniques come with different complexities and bottlenecks, changing significantly the architecture, the API and the business logic of the whole application. This study proposes to capture how these scaling methods perform under different loads, and to define a common boundary between the mentioned scaling practices. It is important to establish whether or not these techniques should be applied separately or altogether, in which for the last case is essential to elaborate a concrete configuration on how the scaling entities should be managed and how the resource allocation should be done. Therefore, this work relies on a microservice architecture, where one service is taken and it is being deployed under different scaling contexts in an isolated environment. Therefore, the resulting measurements and comparisons can give us a valuable insight on how future microservices should be designed and deployed from the very beginning.**

***Keywords*— horizontal scaling, vertical scaling, functional programming, constraints programming, microservice**

I. Introduction

In the cloud computing era, the microservice architecture has gained a widespread popularity in the software development community. It is quickly being adapted as a best practice for creating enterprise applications [1]. Under the microservices architecture model, a traditional monolithic application is dissociated into several smaller, self-contained component services. The most notable benefits of such a decomposition includes enhanced deployability, fault-tolerance and scalability. Companies are increasingly taking advantage of the benefits and are moving their infrastructure to the cloud for reduced operational costs [1].

The workload of internet applications varies dynamically over multiple periods of time. A period of a sudden increase in workload can lead to an overload, when the volume of the requests exceeds the capacity of the available resources. The length of overload periods increases proportionally with the overhead of resource provisioning. During overload periods, the response times may grow to very high levels, where the service is degraded or totally denied. In production environments, such behavior is harmful for the reputation of the internet application providers and might lead to unsatisfied customers [2][3]. As overload periods might appear, there are cases of underload periods as well, when the service demands are low and the power consumption of the hosting infrastructure is high. These periods raise costs since inefficient power utilization given the fact that the scaling techniques are rigid. Elastic scaling would be an optimal method of using the allocated resources [4].

Traditional methods for scaling can generally be categorized into vertical or horizontal scaling with the more popular approach being horizontal scaling [5]. This scaling technique involves replicating a microservice onto other machines to achieve high availability. By replicating a microservice onto another machine, the resource allocation doubles. This approach, however, is greedy and presumes there is no shortage of hardware resources. Furthermore, horizontal scaling creates additional overhead on each replicated machine and is confronted with bandwidth limitations on network and disk I/O, respective hardware limitations on socket connections [1]. From a service point of view, disadvantages might appear if the microservice involves state or a context. There are cases when a microservice is further indivisible by

design but still includes multiple tasks that are related to the it. Without the benefits of vertical scaling, the horizontal model would face difficulties to keep up with the demands, since task decomposition lies in the microservice itself.

Vertical scaling, on the other hand, aims to maximize the utilization of resources on a single machine by providing the microservice more resources such as memory and computing cores. Unfortunately, this method is also limited as a single machine does not necessarily possess enough resources. Upgrading the machines to meet demands quickly becomes more expensive than purchasing additional commodity machines [1]. This model also supposes a more rigid scaling approach, lacking the dynamic behavior of computational resource addition and removal [6].

To compare the two approaches a microservice is being implemented and tested under various scenarios. The given microservice is part of a planning application and it fulfils the most resource demanding role: solving a given planification problem. The task includes a given search space, in which the service must find a solution that satisfies the request by means of validity and preferences. Vertical scaling is achieved by the service architecture by allowing a flexible resource allocation and execution context configuration, while the horizontal scaling is obtained by means of deployment inside a virtualized environment such as Docker. A load testing process is then applied to capture the relevant quality of service criteria [3].

The problem statement for this paper is divided into two main research questions, which will be used as a foundation for the work:

- How do the two scaling techniques perform? Which is better and more load-tolerant, given the fact that each case receives the same amount of allocated resources and requests? The requests are supposed to be equal in count or to be equivalent in complexity.
- How do the two methods perform together? Should they be used separately or altogether to enhance service performance? Is there a know-how for the configuration of the two and what is a good practice when it comes to thread and instance mapping to physical resources?

II. Methodology

A. About the microservice

The implemented microservice serves as the solver module of a planning application. From a high-level perspective, the goal of the solver is to solve one or more problems. A problem consists of distributing operations on a given time interval respecting some constraints. The returned solution must ensure an optimum combination of the variance domain, fact that is guided by the search criteria during the evaluation process. The solution contains the operations distributed in time and performed by an eligible resource, and an appreciation factor called cost. The cost reflects how well did the solution perform taking into consideration the specified criteria.

The solver contains a basic model which is capable of distributing operations in time for one day only, providing the best solution by the defined criteria. It is an exemplative model, having a lot of limitations when considering on what time interval can the distribution extend.

B. Scaling

The microservice is called planr and it is designed to support both horizontal and vertical scaling. The first is achieved in the deployment process, while the second by its architecture.

Horizontal scaling is done by declaring multiple planr instances, placing a load balancer in the front of them called Nginx. Nginx intercepts the requests and forwards each of them to the next microservice instance, distributing the traffic sequentially. From a Docker perspective, Nginx is exposed under port 8900, while the planr instances are hidden and accessible only by the load balancer.

Vertical scaling is achieved by the actor model. During the initialization, the SolverActor gets created, specifying the number of instances and the dimension of the thread pool on which the actor is placed. The SolverActor is published under the actor system that is defined by the Play2 web service API. This actor system follows the service life-cycle and restarts automatically when the service restarts [7]. If one of the SolverActor

instances crashes for some reason, it is automatically restarted by the actor system. The actor system is fault-tolerant and successfully implements the 'Let it crash' model [7]. The deployment for the vertical scaling is almost the same, declaring only one microservice. In the end it resumes that the two techniques are achieved by varying the number of instances, actors and allocated resources. The request and actor message distribution are handled by the same Round-robin fashion either by the load balancer, either by the actor system. While the horizontal scaling contains multiple planr instances, each having one actor, the vertical scaling contains only one instance and multiple actors. The allocated resources are the same for both, the distribution being done only in different stages of the build and deployment process.

C. Resource limitation

The machine on which the Docker environment is running has at its disposal 12 logical processors (Intel i7-9750H 2.60 GHz, 6 cores, L1 cache 384 KB, L2 cache 1.5 MB, L3 cache 12 MB) and 16 GB of RAM at 2667 MHz. Under the Resources option, 10 cores and 10 GB of RAM are allocated to the Docker application. From these, 8 cores and 8 GB of RAM are distributed for the planr application infrastructure, the rest is left for the Nginx and for the Docker internal management. In the case of horizontal scaling, each planr instance of the total 4 receives 2 cores and 2 GB of memory, while in the case of vertical scaling the whole resource amount is allocated to a single planr instance. On an even smaller magnitude, each planr instance consists of a Docker container that runs inside a minimal Linux kernel, the JVM and the application jar. The jar is limited to the one fourth of the container memory resources, meaning that it has 512 MB, respective 2 GB of memory. The count of the actors for the horizontal scaling is 1, while for the vertical scaling is 4, assuring that the other half of the threads are consumed by the Linux kernel and the JVM. These scaling parameters and resource limitations form the base of the testing scenarios.

D. Test scenarios

The initial tests are based on three different scaling scenarios:

Table 1: Initial scaling scenarios

(C – Containers, Co – Cores, M – Memory, A – Akka Actors, H – JVM Heap)

/	C	Co/C	M/C (GB)	A/C	H/C (GB)
planr 4.0 (pure vertical)	1	8	8	4	2
planr 2.0 (hybrid)	2	4	4	2	1
planr 1.0 (pure horizontal)	4	2	2	1	0.5

Further scaling scenarios are derived from planr 1.0. The difference is that each new planr version contains different scale-up parameters. The obtained versions are:

Table 2: Further scaling scenarios

(C – Containers, Co – Cores, M – Memory, A – Akka Actors, H – JVM Heap)

/	C	Co/C	M/C (GB)	A/C	H/C (GB)
planr 1.2	4	2	2	2	0.5
planr 1.4	4	2	2	4	0.5
planr 1.8	4	2	2	8	0.5
planr 1.16	4	2	2	16	0.5

The performance tests are defined in the planr-gatling project. Taking one base Problem, three request derivations were obtained for each scaling scenario:

Table 3: Requests for the performance tests

(R – POST requests, P – Problems, S – Solutions)

/	R	P/R	S/R
Request 4.0	1	4	4
Request 2.0	2	2	2
Request 1.0	4	1	1

All the three request versions expand to a total number of 4 dayFrames. The same results are returned for day 1, 2, 3 and 4, the difference being how the requests with the problems are composed.

III. RESULTS

Each testing was carried out respecting the same procedure, meaning that no background tasks were executing on the host system, having only the Docker environment and Gatling framework

running. Before each test, a warmup was executed, guaranteeing that the initialization process does not influence the average response times. The testing scenarios are 30 seconds long and they have general assertions that check if the whole execution is under 60 seconds and none of the response times exceeds 10 seconds. In case of an assertion violation, a test is failed, but the benchmark information is still available. After the warmup, 7 tests are executed sequentially, each test increasing the number of the injected users per second. While at the beginning 20 users fire in parallel one of the request versions, at the end there are 50. Like this, the results can intercept when the scaling scenario starts to be uncapable to handle the load. After each test execution the Gatling framework outputs a report as a result. The results are composed of the average response times, other charts being available only in the thesis.

A. *Initial outcomes*

The initial test scenarios are depicted in Table 1:

Table 4: Initial test scenarios

/	planr 1.0	planr 2.0	planr 4.0
Scenario 1	Request 1.0	Request 2.0	Request 4.0
Scenario 2	Request 1.0	Request 1.0	Request 1.0
Scenario 3	Request 4.0	Request 4.0	Request 4.0

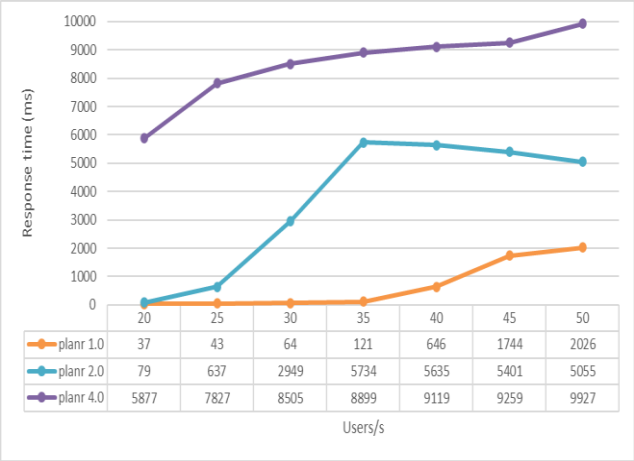

Figure 1: Scenario 1 - Average response time

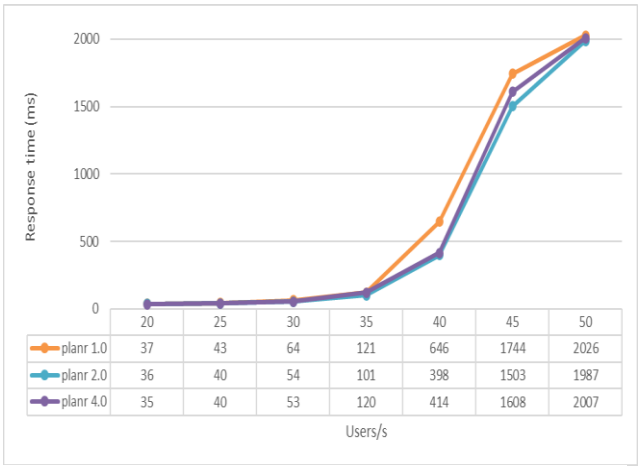

Figure 2: Scenario 2 - Average response time

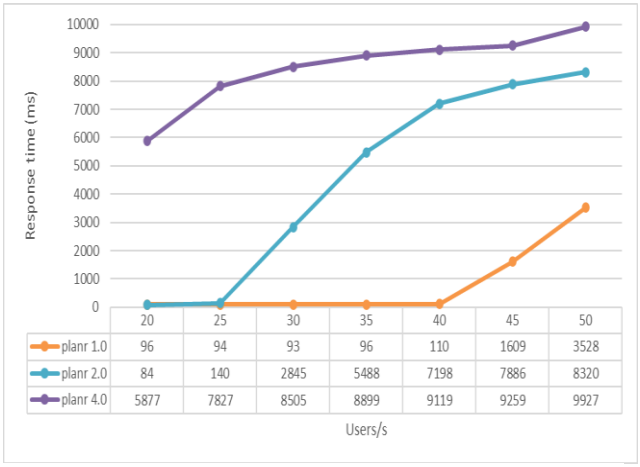

Figure 3: Scenario 3 - Average response time

Overall planr 1.0, the pure horizontal model provides the best results in terms of average response times. The hybrid model classes itself between the two, proving that the vertical scaling technique by itself is inefficient.

B. *Further improvements*

Further test scenarios are depicted in Table 2:

Table 5: Further test scenarios

/	planr 1.2	planr 1.4	planr 1.8	planr 1.16
Scenario 4	Request 1.0	Request 1.0	Request 1.0	Request 1.0
Scenario 5	Request 4.0	Request 4.0	Request 4.0	Request 4.0

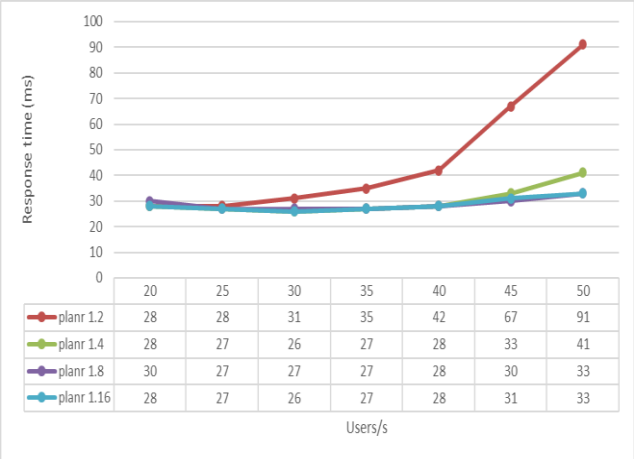

Figure 4: Scenario 4 - Average response time

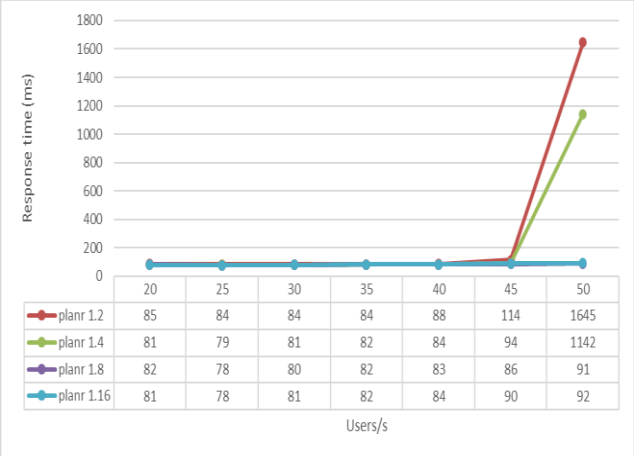

Figure 5: Scenario 5 - Average response time

IV. Conclusions

The benchmark results bring the following achievements:

- Pure horizontal scaling provides better performance than a pure vertical one;
- The combination of scale-out and scale-up techniques maximizes efficiency to cope with overload periods;

The results from section B show that the more actors are added, the better is the outcome. Even adding more actors has a limitation, since to each logical core there is one or more threads mapped. As it can be seen, the difference between planr 1.8 and planr 1.16 is insignificant, obtaining no further improvements after a while. Entities declared for scale-up methods, such as the Akka actors, easily handle a configuration of many to one with the physical resources, the limit still being unknown. Future work consists of a better definition of these entities configuration.

This paper is a complementary value to the current state of the art explored in the thesis and also serves as a comparison between two technologies when it comes to scaling and resource allocation, namely Akka and Docker. The submitted work is meant to respond to those who question the power of functional programming and its capabilities to switch execution contexts and to manipulate threads. Rather considering that one technique should exclude the other, the direction I encourage and emphasize is how these techniques should be practiced together and used efficiently to obtain the best results.

References

[1] - Wong, Jonathon Paul. Hyscale: Hybrid scaling of dockerized microservices architectures. Diss. 2019.
[2] - Dawoud, Wesam, Ibrahim Takouna, and Christoph Meinel. "Scalability and performance management of internet applications in the cloud." Communication Infrastructures for Cloud Computing. IGI Global, 2014. 434-464.
[3] - Ahmad, Bilal. Coordinating vertical and horizontal scaling for achieving differentiated QoS. MS thesis. 2016.
[4] - Kumar, Ranjan, and G. Sahoo. "Characteristics Based Scale-Out vs. Scale-Up for Green Cloud Computing."
[5] - Sevilla, Michael. "A Short Comparison of Scale-up and Scale-out."
[6] - Sevilla, Michael, et al. "A framework for an in-depth comparison of scale-up and scale-out." Proceedings of the 2013 international workshop on data-intensive scalable computing systems. 2013.
[7] - Hunt, John. "Play framework." A Beginner's Guide to Scala, Object Orientation and Functional Programming. Springer, Cham, 2