Politehnica University Timișoara
Faculty of Automation and Computers
Department of Computers and Information Technology

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

Master Thesis

Candidate:
Arnold Attila HIGYED

Supervisor:
Lect.dr.eng. **Alexandru TOPÎRCEANU**

Timișoara
2020

Contents

List of figures

List of tables

1. Introduction

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. A service is valuable for the client until the requirements are met, scalability being one of its essential capabilities.

1.1. Problem statement

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. 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 web application providers and might lead to unsatisfied customers [2]. 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 because of inefficient power utilization, given the fact that the scaling techniques are rigid. Elastic scaling would be an optimal method of using the allocated resources. The testing and comparison of these various elastic distribution patterns are not the goal of this thesis [4].

Traditional methods for scaling can generally be categorized into vertical and horizontal scaling, with the more popular approach being horizontal scaling. This scaling technique involves replicating a microservice onto other machines to achieve high availability. By replicating a microservice onto another machine, the resource allocations double. This approach however is greedy and assumes 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, all being related to a common context. 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, not before requesting it.

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 or removal.

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 carried out to capture the relevant quality of service criteria.

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?

1.2. Motivation

Working in the cloud environment, I can state that horizontal scaling was the only approach attended by the company where I work. To obtain a better service availability, vertical scaling was never attempted because the complexity of concurrent code execution. To obtain a robust service architecture, functional programming principles had to be applied, but none possessed enough expertise to deal with such a challenge, until a new project demanded it. The project was composed of several microservices, from one of which quite complex and resource demanding. This particular microservice included task decomposition in itself, with several fallback cases and internal state. This meant that horizontal scaling would have created a design overhead and complicated maintenance, if implemented in such a way. The project was launched supporting both scaling techniques and was finalized two years later. In production, there was a lot of suspicion if vertical scaling did improve the whole process, and it was quite attempting to completely eliminate it and to compensate the horizontal one. From this standpoint, I decided to create a similar and much simpler microservice, that includes some demanding tasks, and to benchmark both of these scaling techniques. Functional programming proves quite difficult to learn since it’s steep learning curve, and it is expensive from management point of view. Looking for similar studies and papers on the internet, I found plenty that address this problem of scaling from different viewpoints. This thesis

represents the challenges and design decision that were overcome to respond to the stated research questions.

2. State of the art

The comparison between scale-out and scale-up techniques, and the combination of the two is addressed by several studies that can be divided in two main categories: related master and doctor theses, and related research articles. The following subsections present the most relevant and similar works of these groups, highlighting the proposed problems, questions and conclusions. The comparison of these two scaling strategies are presented in various forms and frameworks.

2.1. Related theses

HyScale: Hybrid Scaling of Dockerized Microservices Architectures [1]

This study leverages the high availability of horizontal scaling and the fine-grained resource control of vertical scaling. Two novel reactive hybrid autoscaling algorithms are presented and benchmarked against Google's popular Kubernetes horizontal algorithm. Results consists of up to 1.49x speedups in response times, 10 times fewer failed requests, and 35% increase in resource efficiencies. Further on a new predictive approach is also explored using neural networks and a custom online learning technique, indicating prediction errors as low as 3.77% [1]. Based on the experimental analysis, the author validates the efficiency of the two strategies combined [1].

Scalability and Performance Management of Internet Applications in the Cloud [2]

In this dissertation, two approaches are proposed to alleviate the impact of the resource provisioning overhead. The first approach embeds control theory to scale resources vertically and cope relatively fast with the workload. This technique assumes that the provider has knowledge and control over the platform running in the virtual machines. This approach is limited to Platform as a Service (PaaS) and Software as a Service (SaaS) models. The second method is more customer-oriented dealing with the horizontal scalability in an Infrastructure as a Service (IaaS) model. It addresses the trade-off problem between cost and performance with a multi-goal optimization solution. This approach proves to achieve the highest performance with the lowest increase in costs [2]. Moreover, it employs a proposed time series forecasting algorithm to scale the application proactively and avoid under-utilization periods. To mitigate the interference impact on the web application performance, a system is developed which identifies and eliminates the virtual machines suffering from performance interference. The developed system is a light-weight solution that does not imply provider involvement. The evaluation of the proposed techniques is ensured by a simulator called ScaleSim. The workload is generated from the access logs of the official website of 1998 World Cup. The results show that optimizing the scalability thresholds and adopting proactive scalability can mitigate 88% of the resources provisioning overhead impact with only a 9% increase in the cost [2].

Coordinating vertical and horizontal scaling for achieving differentiated QoS [3]

This thesis investigates how the problem of scaling and of dynamic resource allocation in a cloud infrastructure can be addressed, by designing and implementing an autonomic provisioning controller. The concept is based on elements from control theory and defines measures to coordinate real-time and non-real-time applications. It is shown that traditional approaches perform

elastic decisions either based on the monitoring of the resource usages, or based on the quality of service criteria [3]. Applications are identified as being different based on their service level agreement constraints, not all of them being considered as latency-critical applications. The implementation focuses on resource allocation coordination between real-time and non-real-time applications. It proposes a hybrid model that relies on both resource usage and QoS requirements in the same time, in order to drive the elasticity decisions. Horizontal and vertical scaling is orchestrated under the same framework, obtaining in the end the expected results [3].

2.2. Related papers

Characteristics Based Scale-Out vs. Scale-Up for Green Cloud Computing [4]

This paper proposes a scale-out and scale-up hybrid approach, which follows to lower the energy consumption of a cluster. An integrated mechanism is used between the servers and the jobs, consisting of combinations of optimization techniques. In the end a method for green cloud computing is elaborated, in which a job scheduler schedules jobs to different web services [4]. The differentiation is determined whether it's a larger or a smaller job and a decision is taken by the scheduler determining the distribution of the job to a scale-out, scale-up or to a hybrid scaling infrastructure. The designed model brings no degradation to the quality of service of the datacenters or clusters [4].

A Short Comparison of Scale-up and Scale-out [5]

This paper presents the initial steps that are taken into consideration in a direct scale-up vs. scale-out study. The hypothesis is that scale-up systems degrade differently than scale-out systems. Current distributed systems benchmarks and workloads are selected in an effort to compare the actual performance of the two counterparts. The conclusions are that narrow methodologies and outdated hardware may lead to miss crucial bottle-necks and overheads [5]. With the help of HiBench to select applications, and Phoenix to port them, it is showed how the proposed methodologies uncover important details about the interplay between core, data and memory. The biggest contribution made is the direct comparison between scale-out and scale-up systems, remaining fair, representative, and feasible [5].

A Framework for an In-depth Comparison of Scale-up and Scale-out [6]

A novel comparison framework is proposed in this study based on MapReduce that accounts for the application, its requirements and its input size, by considering input, software and hardware parameters. Part of this framework requires implementing scale-out properties on scale-up [6]. The complex trade-offs, interactions and dependencies of these properties are discussed for two specific case studies: word count and word sort. This work lays the foundation for future work in quantifying design decisions and in building a system that automatically compares architectures and selects the best one [6].

Scale-up x Scale-out: A Case Study using Nutch/Lucene

This paper investigates the behavior of two competing approaches to parallelism, scale-up and

scale-out in an emerging search application. The conclusions show that a scale-out strategy can be the key to good performance even on a scale-up machine [7]. Furthermore, scale-out solutions offer better price per performance ratio, although at an increase in management complexity. The results also reveal that running scale-out in a box gives better performance than using multi-threading [7].

Scale-up vs Scale-out for Hadoop: Time to rethink? [8]

The measurements as well as other recent work shows that the majority of real-world analytic jobs process less than 100GB of input, but popular infrastructures such as Hadoop/MapReduce were originally designed for peta scale processing [8]. The hypothesis is that a single scale-up server can process each of these jobs and do as well or better than a cluster in terms of performance, cost, power and server density. The paper presents an evaluation across 11 representative Hadoop jobs that shows scale-up to be competitive in all cases and significantly better in some cases, than scale-out. To achieve that performance, several modifications to the Hadoop runtime that target the scale-up configurations are described. These changes are transparent, do not require any changes to the application code, and do not compromise the scale-out performance. The results show the contrary to conventional wisdom, analytic jobs, in particular Hadoop and MapReduce jobs, are often better served by a scale-up server than a scale-out cluster [8].

Scale-Out vs. Scale-Up Techniques for Cloud Performance and Productivity [9]

This paper proposes an elastic cloud provisions upon user demand. Auto-scaling, scale-out, scale-up, or any mixture techniques are used to reconfigure the user cluster as workload changes. Three scaling strategies are evaluated to upgrade the performance, efficiency and productivity of elastic clouds like EC2, Rackspace [9]. The strengths and shortcomings of the three scaling strategies are revealed in the HiBench experiments [9]:

- Scale-out overhead is shown lower than in scale-up or mixed scaling clouds. Scale-out to a larger cluster of small nodes demonstrates high scalability.
- Scaling up and mixed scaling have high performance in using smaller clusters with a few powerful machine instances.
- With a mixed scaling mode, the cloud productivity is shown upgradable with higher flexibility in applications with performance/cost tradeoffs.

Cloud Performance Modeling and Benchmark Evaluation of Elastic Scaling Strategies [10]

This paper presents cloud performance models for evaluating Iaas, PaaS, SaaS, and mashup or hybrid clouds. The benchmark experiments are conducted mainly on IaaS cloud platforms over scale-out and scale-up workloads. Cloud benchmarking results are analyzed with the efficiency, elasticity, QoS, productivity, and scalability of cloud performance. Five cloud benchmarks are tested on Amazon EC2: namely YCSB, CloudSuite, HiBench, BenchClouds, and TPC W [10]. To satisfy production services, it is stated that the choice of scale-up or scale-out solutions should be based on the workload patterns and resources utilization rates required. Scaling out machine instances have much lower overhead than those experienced in scale-up experiments. However, scaling up is found more cost effective in sustaining from higher workload. The cloud productivity is greatly attributed to system elasticity, efficiency, QoS and scalability. Auto scaling is considered easy to implement with the flaw to over provision a node with resources. Lower resource utilization

rates may result from methods like auto scaling [10].

Scale up Vs. Scale out in Cloud Storage and Graph Processing Systems [11]

Deployers of cloud storage and iterative processing systems typically have to deal with either budget constraints or throughput requirements [11]. This paper examines the question of whether such cloud storage and iterative processing systems are more cost-efficient when scheduled on a scale-out cluster or a single scale-up machine. Two systems are evaluated: a distributed key-value store database such as Cassandra, and a distributed graph processing system like GraphLab. The research reveals scenarios where each option is preferable over the other. There are recommendations made by the authors for deployers of such systems to decide between scale-up and scale-out, as a function of their budget and throughput constraints. The conclusion states that there is a need for an adaptive scheduling in heterogeneous clusters containing scale-up and scale-out nodes [11].

3. Technical background

In this section we mention the technologies and important concepts with which the project implementation and testing was realized. Therefore, it is important to understand the reason behind each component and functionality in order to interpret the final results. Shortly, the programming language in which the microservice was written is Scala, providing support for functional programming. As build tool, sbt was preferred, since it comes with a lot of features and helpers for the Scala ecosystem. Play2 was used for the RESTful API, while Akka for the scale-up technique implemented with the actor model. Cats is a library which provides abstractions for functional programming. For the realization of the business logic, Google OR-Tools represented the most efficient choice, which is a fast and portable software suite for solving combinatorial optimization problems. Docker enabled to deploy the application in containers and last but not least, Gatling allowed to perform load testing on various scaling scenarios of the planning app.

3.1. Scala

The name Scala stands for "scalable language". The language is so named since it was designed to grow with the demand of its users. It is general-purpose programming language, which gives the ability to its users to write small scripts or to build large systems. Its strength as a language are relieved when designing large systems and frameworks of reusable components [12].

Scala is a pure object-oriented language in the sense that every value is an object. Types and behaviors of objects are described by classes and traits. Classes can be extended by subclassing, and by using a flexible mixin-based composition mechanism as a clean replacement for multiple inheritance [12].

Scala is also a functional language in the sense that every function is a value. Scala provides a lightweight syntax for defining anonymous functions, it supports higher-order functions, it allows functions to be nested, and it supports currying. Scala's case classes and its built-in support for pattern matching provide the functionality of algebraic types, which are used in many functional languages [12]. Singleton objects provide a convenient way to group functions that aren't members of a class. Furthermore, Scala's notion of pattern matching naturally extends to the processing of XML data with the help of right-ignoring sequence patterns, by way of general extension via extractor objects. In this context, for comprehensions are useful for formulating queries. These features make Scala ideal for developing applications like web services [13].

Scala's expressive type system enforces, at compile-time, that abstractions are used in a safe and coherent manner. In particular, the type system supports [13]:

- Generic classes;
- Variance annotations;
- Upper and lower type bounds;
- Inner classes and abstract type members as object members;
- Compound types;
- Explicitly typed self-references;
- Implicit parameters and conversions;
- Polymorphic methods;

Type inference means the user is not required to annotate code with redundant type information. In combination, these features provide a powerful basis for the safe reuse of programming abstractions and for the type-safe extension of software [13].

Scala is designed to interoperate well with the popular Java Runtime Environment (JRE). In

particular, the interaction with the mainstream object-oriented Java programming language is as seamless as possible. Newer Java features like SAMs, lambdas, annotations, and generics have direct analogues in Scala. Those Scala features without Java analogues, such as default and named parameters, compile as closely to Java as reasonably possible. Scala has the same compilation model as Java and allows access to thousands of existing high-quality libraries [13].

3.2. sbt

The simple build tool (sbt) is used for building Java and Scala projects. Its purpose is to allow users to skillfully perform the basics of building and packaging an application. It gives the developer the freedom to customize as need demands. sbt, at its core, provides a parallel execution engine and configuration system that allow its users to design an efficient and robust script to build the desired software. sbt aims to be consistent in the basic concepts in order to facilitate the creative process. sbt is a highly interactive tool, meant to be used during all stages of the development. It provides interactive help and autocomplete for most services and promotes a type of auto-discovery for builds [14]. Its main features are [15]:

- Little or no configuration required for simple projects;
- Scala-based build definition that can use the full flexibility of Scala code;
- Accurate incremental recompilation using information extracted from the compiler;
- Continuous compilation and testing with triggered execution;
- Packages and publishes jars;
- Generates documentation with scaladoc;
- Supports mixed Scala/Java projects;
- Supports testing with ScalaCheck, specs, and ScalaTest. JUnit is supported by a plugin;
- Starts the Scala REPL with project classes and dependencies on the classpath;
- Modularization supported with sub-projects;
- External project support;
- Parallel task execution, including parallel test execution;
- Library management support that include inline declarations, external Ivy or Maven configuration files, or manual management;

3.3. Play2

Play is a high-productivity framework for building a wide range of different types of web applications. These are applications that receive requests for data and functionality over HTTP(s). Play aims to make the construction of such applications simple, flexible and intuitive. It incorporates an integrated HTTP Server. It also incorporates a templating framework for the creation of websites and a RESTful web service API for the creation of microservices. It exploits the facilities within the Scala ecosystem, such as Akka, to ensure that the applications developed are scalable and perform well [16].

3.4. Akka

Akka is a Scala-based toolkit that simplifies developing concurrent distributed applications. Perfect for high-volume applications that need to scale rapidly, Akka is an efficient foundation for event-driven systems that want to scale elastically up and out on demand, both on multi-core processors and across server nodes. The framework has at its roots the actor model concept in order

to raise the abstraction level that decouples the business logic from the low-level constructs of threads, locks and non-blocking I/O. The framework provides the following features [17]:

- Concurrency;
- Scalability;
- Fault tolerance;
- Event-driven architecture;
- Transaction support;
- Location transparency;
- Scala/Java APIs;

3.5. Cats

Cats is a library which provides abstractions for functional programming in the Scala programming language. The name is a playful shortening of the word category. Scala supports both object-oriented and functional programming, and this is reflected in the hybrid approach of the standard library. Cats strives to provide functional programming abstractions that are core, binary compatible, modular, approachable and efficient. A broader goal of the mentioned library is to provide a foundation for an ecosystem of pure and typeful libraries [18].

3.6. OR-Tools

Google's Operations Research tools is an open source software for combinatorial optimization, which seeks to find the best solution to a problem out of a very large set of possible solutions. OR-Tools includes solvers for [19]:

- Constraint programming: a set of techniques for finding feasible solutions to a problem expressed as constraints;
- Linear and mixed-integer programming: the Glop linear optimizer finds the optimal value of a linear objective function, given a set of linear inequalities as constraints;
- Vehicle routing: a specialized library for identifying the best vehicle routes given in constraints;
- Graph algorithms: code for finding shortest paths in graphs, min and max cost flows, and linear sum assignments;

In most cases, problems like these have a vast number of possible solutions, too many for a computer to search them all. In order to overcome this, OR-Tools uses state-of-the-art algorithms to narrow down the variance domain of the problem, in order to find optimal or close to optimal solutions [19].

3.7. Docker

Docker is a software platform for building applications based on containers. Containers are small and lightweight execution environments that make shared use of the operating system kernel, but otherwise run in isolation from one another. While containers as a concept have been around for some time, Docker is an open source project launched in 2013. It helped to popularize the technology and drove the trend towards containerization and microservices in software development, that has come to be known as cloud-native development [20].

Docker brings the following advantages [20]:

- Portability: once a containerized application tested, it can be deployed to any other

system where Docker is running;

- Performance: virtual machines are an alternative to containers. The fact that containers do not contain an operating system, whereas virtual machines do, means that containers have much smaller footprints than virtual machines. As a result, containers are faster to create, and quicker to start;
- Agility: the portability and performance benefits offered by containers can help its users to make the development process more agile and responsive;
- Isolation: a container that contains an application also includes the relevant versions of any supporting software that the application requires. If other containers contain applications that require different versions of the same supporting software, that isn't a problem because the different containers are totally independent of one other;
- Scalability: when using multiple containers there are a range of management options for achieving horizontal scaling;

3.8. Gatling

Gatling is a highly capable load testing tool. It is designed for ease of use, maintainability and high performance. It takes advantage of its asynchronous architecture, which allows it to implement virtual users as messages instead of dedicated threads, making running numerous simultaneous virtual users less resource heavy than other solutions. Gatling tests are called simulations. They are written in Scala-files and each class represents its own load simulation. We can assign properties such as the URL, parameters and headers that are needed for the HTTP requests into variables using Gatling HTTP protocol builder [21].

4. Implementation

In the first part of this section the business logic is presented with the request and response API of the microservice. These are accompanied by the validation tests that also serve as examples. Further on the architecture of the microservice is detailed. The project structure explains each package and their scope. Next the build with sbt and the deployment with Docker are discussed, tearing down the whole process into small steps. Right after the scaling techniques are pointed out. These come accompanied by some resource limitation measures, since they are key to achieve an isolated environment, making each test execution encapsulated. Finally, the load test requests and scenarios are described. The proposed scenarios are benchmarked with different parameters, generating the final results.

4.1. Business logic

The implemented microservice serves as the solver module of a planning application. Mainly a general planning application is composed by the following parts:

- An input module, which allows the user to input its preferences. This is usually embedded in the frontend;
- A data loader or fetcher, which gathers all the relevant data to construct the search context from one database or more;
- A solver that is responsible for solving the search context guided by the given user preferences;
- An aggregator that communicates and orchestrates the data loader and the solver, performing the necessary transformations and conversions of data;
- An output module, which is embedded alongside with the input module, displaying the final results to the user;

These modules were identified as a result of the separation of concerns principle and as of need of scaling. Figure 1 helps for a better visualization of the described architecture. As it can be seen each service performs on its own a specific task, being independent from one another. This provides flexibility to the whole application, since individual parts can be scaled horizontally as the demand requires. Obtaining such separation also brings the benefit of not needing to scale the entire system if one element fails to cope with an increased load.

Further on the solver service is getting detailed as matter of its functionality. From the data flow perspective, the solver receives data that is already transformed, converted and processed by a higher level microservice, meaning that the only step that it's left is to solve the given search context. The search context naming is given by the fact that the data is composed of the user's preferences and the relevant information loaded from the database, forming a so-called context of the search. The communication between the microservices is done via a RESTful API. The solver module accepts a POST operation on the route '/solve' having as an input a Problems json and as an output a Solutions json. Figure 2 and Figure 3 shows the Problems, respective the Solutions API. On the right of each field there is a comment, that specifies if the field whether or not is optional. The solver engine supports an input with the lack of constraints and costs, solving a general planification problem. The searchInterval field helps to normalize the solutions costs when performing various scaling scenarios. For example, when a user wants to distribute some operations over one or ten days, then the solutions cost will vary in function of the distribution interval. The horizontal and vertical scaling methods must output the same results for the performance tests, whether the results are returned as a whole, or aggregated from parts. Nonetheless this field has no

essential role in the internal logic of the distribution engine, being introduced when the scaling methodologies were designed.

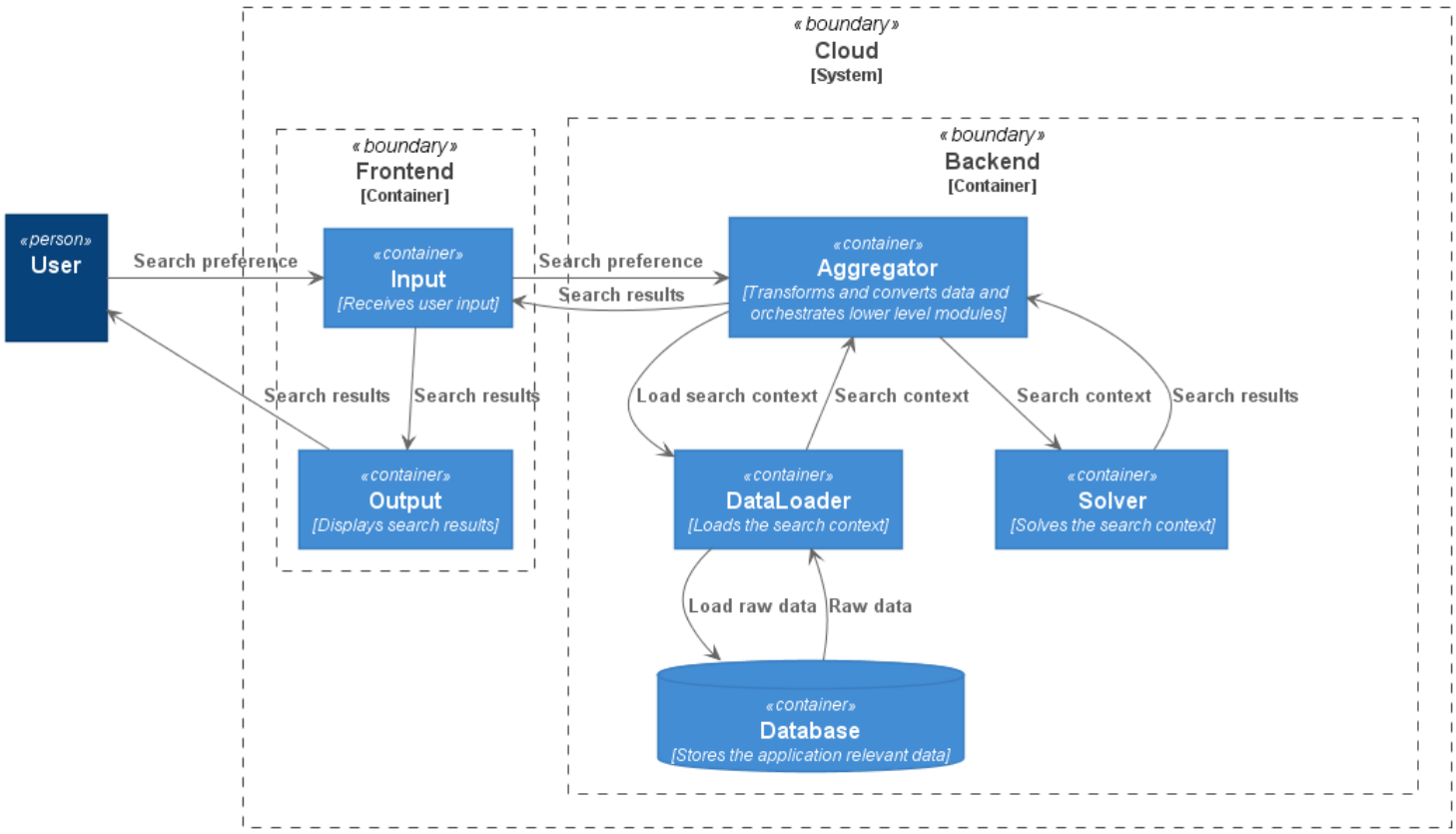


Figure 1: Planning application architecture

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 search 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 next paragraphs present in detail the input and output of the solver.

The Problems API contains 3 fields from a top down approach: a problem that describes a general problem independent of the duration timeline, an array of dayFrames that place the given problem in specific time intervals, and the mentioned searchInterval. The problem is composed of a key as tracing data of the request, an array of operations, an array of resources susceptible to perform these operations, and two optional fields, constraints, respective costs. An operation is defined by a unique key, a name, a fixed duration and some resource keys that might perform it. A resource has a unique key and a name. The solver supports three kind of costs and these are:

- asSoonAsPossible – seeks to provide a solution that starts as soon as possible relative to the search start time;
- asTightAsPossible – the solution with the minimum total duration is preferred, distributing the operations as tight as possible to each other;
- preferredTimeInterval - favorizes a solution that has its operations between a given time interval that is calculated for each day, since days might differ in length when in UTC distribution;

If no costs are defined by the user, then the search engine will declare the first valid variation of the search domain as the best solution, having a cost of zero. In terms of constraints, the solver implements six and these are:

- operationGrid – operations start and an hour in minutes must be divisible by the grid value;
- sameResource – two or more operations must be performed by the same resource;
- enforcedTimeInterval – all the operations must be in the given time interval, calculated for each day, as same as the preferredTimeInterval cost;
- operationsRelation – a relation must be respected between two operations. The relations can be the following: ends after end, ends after start, ends at end, ends at start, starts after end, starts after start, starts at end and starts after end;
- program – respect the working program of the day in which the solution is;
- disjunctive – the blocked intervals of a resource must be disjunctive with the solution interval of that resource, if chosen;

The last two constraints are enforced by default on the solver model, while the other four are exposed as optional constraints on the API. The difference between a cost and a constraint is that if a constraint is not respected, then the solution is dropped, while for a cost, the solution is just depreciated. A dayFrame contains the start and stop values, the program and the blocked intervals for all the resources of a day. Each dayFrame places the generic problem in different time contexts.

The outputted Solutions API can contain some or none solutions. A solution will contain a cost, a total duration of all the operations in minutes, a day and interval in minutes to specify on which day and in which time interval are the operations distributed and finally an array of operations, each having its time interval and resource defined.

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. The solver model and business logic correctness are assured by 33 validation tests. They cover sufficient input cases to approve the fairness of the model. The tests are divided into two groups, input validation tests and solver tests. The first group is focused on the Problems validator, which checks for erroneous and incorrect fields of the input, while the second group tests each cost and constraint of the model separately, assuming that when they are combined the outputted solutions are conform the expectations. These tests can be also taken as examples for elaborating more complicated requests.

```
{
  "problem": {                                  //
    "key": "String",                            //
    "operations": [{                            //
        "key": "String",                        //
        "name": "String",                       //
        "duration": "Long",                     //
        "resourceKeys": "[String]"              //
      }],
    "resources": [{                             //
        "key": "String",                        //
        "name": "String"                        //
      }],
    "costs": {                                  // Optional
      "asSoonAsPossible": "Boolean",            // Optional
      "asTightAsPossible": "Boolean",           // Optional
      "preferredTimeInterval": {                // Optional
        "startT": "Long",                       //
        "stopT": "Long"                         //
      }
    },
    "constraints": {                            // Optional
      "operationGrid": "Long",                  // Optional
      "sameResource": [[                        // Optional
          "String",                             //
          "String"                              //
        ]],
      "enforcedTimeInterval": {                 // Optional
        "startT": "Long",                       //
        "stopT": "Long"                         //
      },
      "operationsRelation": [{                  // Optional
          "opRelType": "OperationRelationType", //
          "opKey1": "String",                   //
          "opKey2": "String"                    //
        }]}
  },
  "dayFrames": [{                               //
      "day": {                                  //
        "startDt": "Long",                      //
        "stopDt": "Long"                        //
      },
      "program": {                              //
        "startT": "Long",                       //
        "stopT": "Long"                         //
      },
      "allocations": [{                         //
          "resourceKey": "String",              //
          "intervals": [{                       //
              "startT": "Long",                 //
              "stopT": "Long"                   //
            }]}]
  }],
  "searchInterval": "Long"                      // Optional
}
```

Figure 2: Problems API

```
{
  "solutions": [{                //
      "cost": "Double",          //
      "duration": "Long",        //
      "day": {                   //
        "startDt": "Long",       //
        "stopDt": "Long"         //
      },
      "interval": {              //
        "startDt": "Long",       //
        "stopDt": "Long"         //
      },
      "operations": [{           //
          "key": "String",       //
          "name": "String",      //
          "duration": "Long",    //
          "resource": {          //
            "key": "String",     //
            "name": "String"     //
          },
          "interval": {          //
            "startDt": "Long",   //
            "stopDt": "Long"     //
          }
        }]}]
}
```

Figure 3: Solutions API

4.2. Architecture

The project is named planr and it is composed of three subprojects: planr-api, planr-core and planr-gatling. The architecture of the solver microservice includes only the first two, the api and the core, while the last module contains the performance test definitions. The api contains the so-called case classes which are the input and output messages of the solver, described in the previous subsection, and the main and test service interfaces that are bound to the main controller to expose the '/solve' and the '/test' routes. As a request is captured by the controller, the implementations of the mentioned interfaces are invoked. While the TestService returns a plain 'OK!' message to the requester, the SolverService hands over the execution flow to the SolverActor, moment in which a context shift happens from the initial REST thread pool to the solver thread pool. This separation of dispatchers is when the vertical scale intervenes, allowing the developer to configure how much resources should be allocated to the segment in which the heavy computations are done. On the actor side first the solver model is built, creating the variance domain of the search space, the constraints and the cost criteria. Then the model is solved and the available solutions are extracted. The actor then responds to its sender with the best obtained result. The responses from all the actor instances are collected and served back to the initial requester as the final response. The architecture is depicted in Figure 4.

One reason why the Scala programming language was picked is due to the design too. The shift of execution contexts raises difficulties when working in an environment with mutable states and impure functions. Functional programming guards against bad practices that might occur when concurrency is involved. As a result, vertical scaling can be achieved in a safe and simple manner.

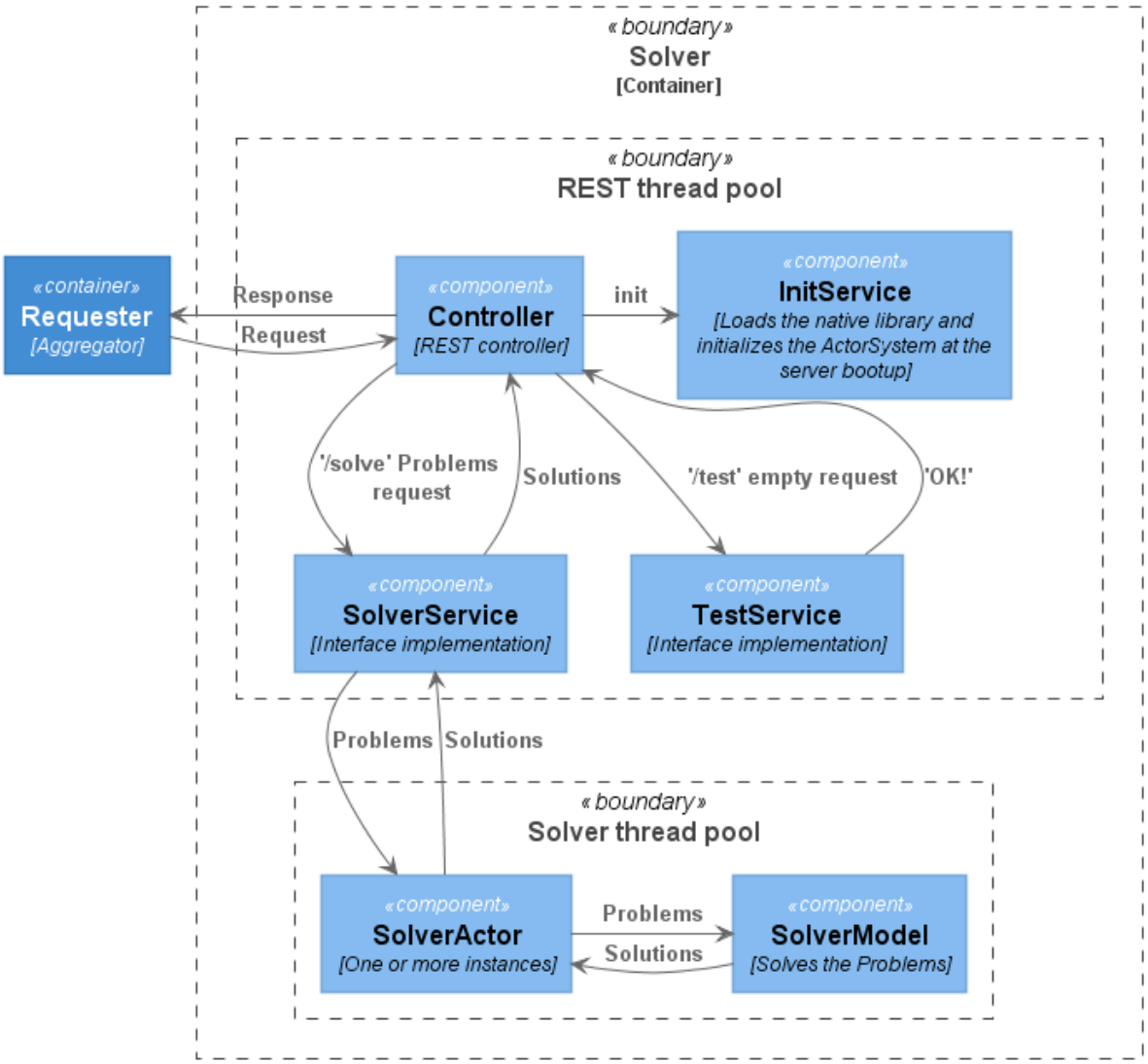
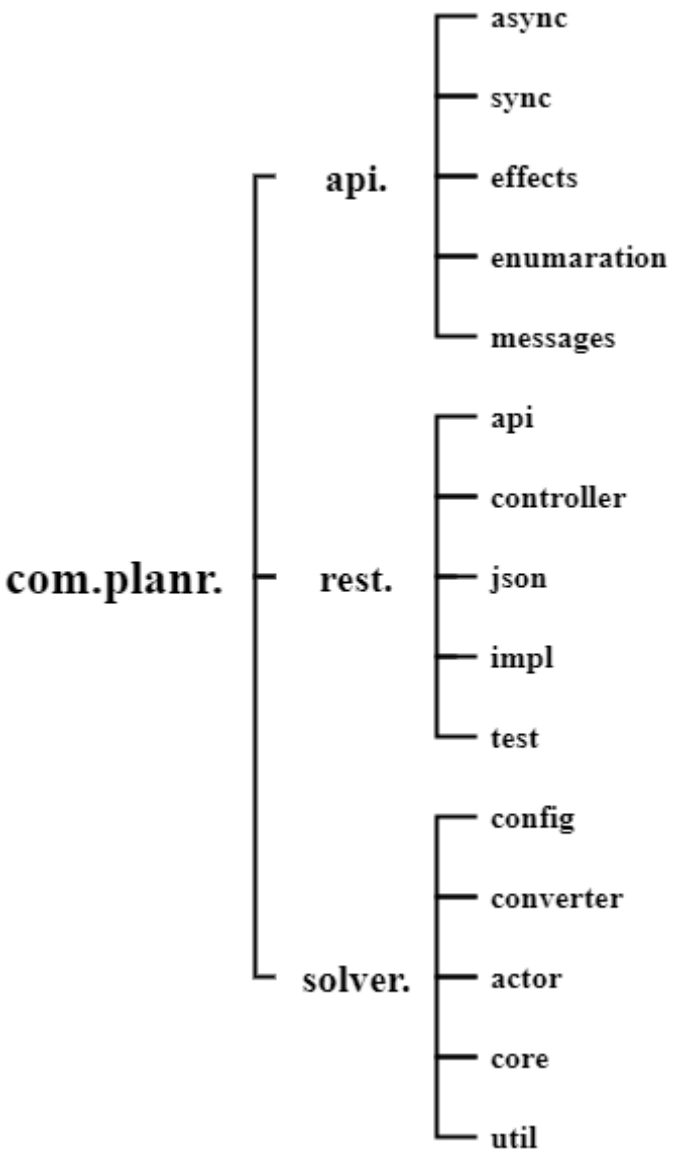


Figure 4: Solver architecture

4.3. Project structure

From a structure perspective, the project ends up having 15 different packages. The files containing classes, objects and enumerations are separated after their responsibilities and by their scopes. It is important to mention that the configurations of the Play2 HTTP server can be found in the conf folder, alongside with the logback preferences and the routes definition. The application.conf holds all the necessary properties for the server to bootup, like the scaling parameters for the actors, the published modules for the dependency injection framework and the timeouts. The rest of the files and folders are build and deployment related that is detailed in the upcoming subsection. Figure 5 represents a tree diagram of the package hierarchy of the project, each node containing the following entities:



- interfaces that are executed asynchronously: the TestService and the SolverService;
- interfaces that are executed synchronously: the InitService;
- type definitions, functional primitives and monad transformers;
- enumeration definitions that are used in the messages package;
- the Error, Problems and Solutions API alongside with the ErrorCodes and the input validation rules;

- the REST API and helpers;
- the controllers such as the TestController and the SolverController;
- serialization primitives and helper functions;
- the async and sync service implementations;
- validation tests;

- the solver model and the cost criteria configurations;
- the SolutionConverter that transforms the data obtained from the model into the Solutions API;
- the SolverActor;
- the definition of the variance domain and the solver model;
- the NativeLibLoader that is meant to load the OR-Tools native library;

Figure 5: Project package structure

4.4. Build and deployment

The build definition can be found in the build.sbt file. This consists of three subprojects, build dependencies, the build and the deployment tasks. This in turn uses settings and config files from the project folder and these are:

- build.properties – specifies the sbt version;
- CompilerSettings.scala – defines the Scala and the Java version of the project, and a list of custom compiler flags and plugins that are meant to avoid bad coding practices;
- plugins.sbt – a list of plugins for the jar assembly, jar unpackaging, Docker deployment, Scala formatting, Play2 server bootup and for the Gatling tests execution;

- PublishingSettings.scala – the developer credentials for publishing the project as a jar on Ivy or Maven;
- RuntimeConfig.scala – the JVM runtime configurations in development mode for debugging, memory limitation, error options and garbage collection policy;
- Settings.scala – general information about the project;
- version.sbt – project release version;

A script file called build.sh contains all the instructions necessary to obtain a planr application packaged in a Docker image. This executes the following commands:

- sbt build – cleans, compiles and test compiles the project. At the end unpacks the OR-Tools native library, that is packaged into a jar and deployed on the local maven repository;
- sbt docker – runs all the validation tests, merges the dependency tree of the planr-api and planr-core subprojects, and assembles the final runnable jar. Afterwards creates a Docker image file and deploys it to the local Docker. The image creation process involves copying the native library and the runnable jar, and configuring the entry point of the container and the JVM runtime options of the application;

With the Docker plugin, sbt successfully embeds the build in the deployment, making it concise and simple to follow, having only three tasks defined for the whole process.

4.5. Scaling and resource limitation

Planr is designed to support both horizontal and vertical scaling. The first is achieved in the deployment process, while the second by the microservice 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 one of the available microservice instances, distributing the traffic equally. From a Docker perspective, Nginx is exposed under port 8900, while the planr instances are hidden and accessible only by the load balancer. All the setup can be found in the nginx.conf and the docker-compose.yml config files.

Vertical scaling is achieved by the actor model. The InitService is initialized during the microservice bootup, since it's published as an eager singleton in the dependency injection framework. 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 application life-cycle and restarts automatically when the application restarts. 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 [16]. 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. Figure 6 and 7 shows the difference between the two deployment structures, each embedding its specific scaling technique.

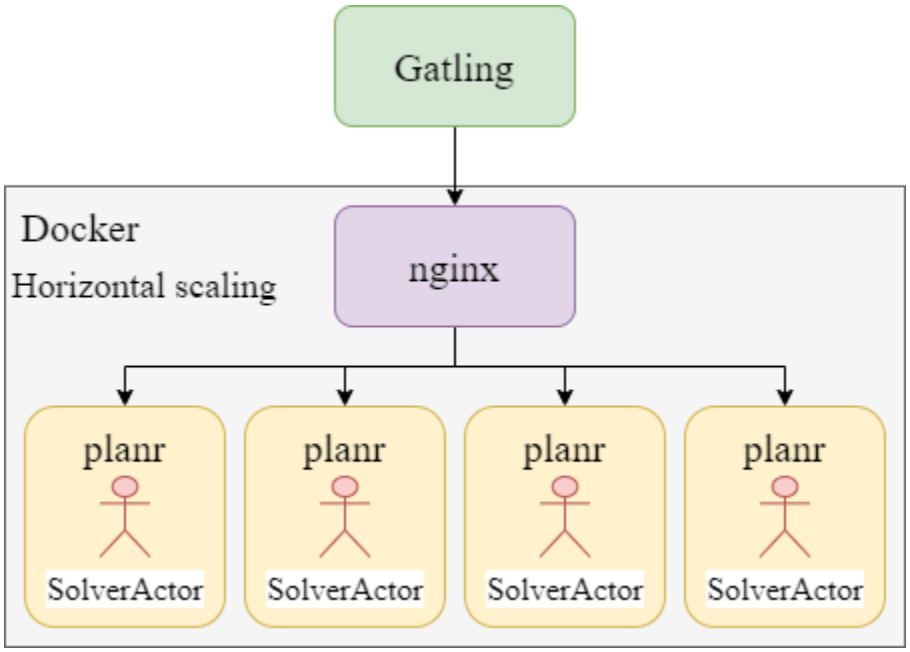


Figure 6: Horizontal scaling

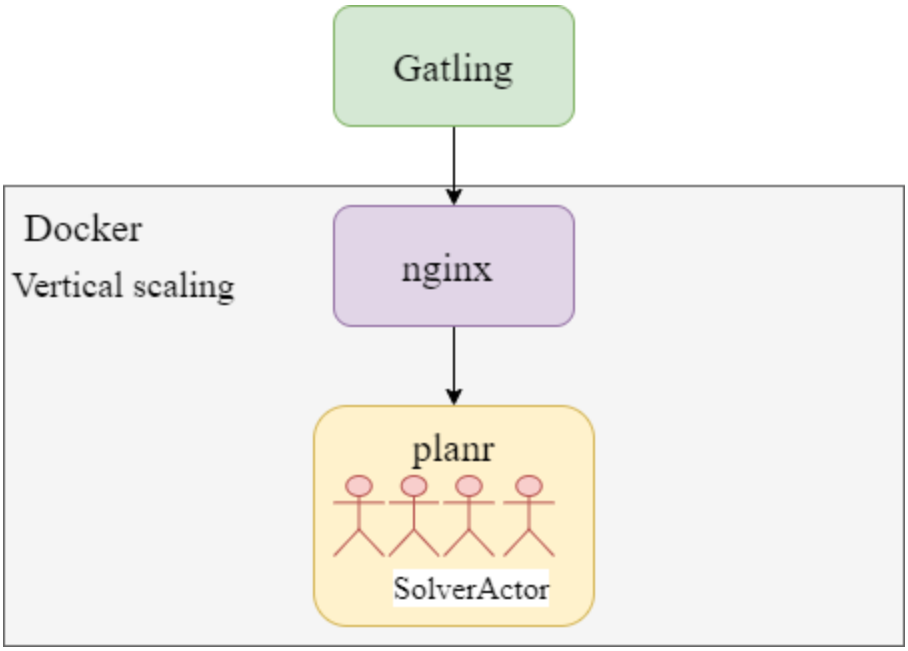


Figure 7: Vertical scaling

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 section, 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 containers and 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 in turn is limited to the one fourth of the container memory resources, meaning that it has 512 MB and 2 GB of memory for the two cases. 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 initial scaling parameters and resource limitations form the base of the testing scenarios.

4.6. Test scenarios

The tests are based on three different scaling infrastructures:

- Pure vertical, planr version 4.0 – 1 planr container with 8 cores and 8 GB of memory, having inside a planr application with 4 actors and a maximum heap of 2 GB;
- Hybrid, planr version 2.0 – 2 planr containers with 4 cores and 4 GB of memory each, one container having inside a planr application with 2 actors and a maximum heap of 1 GB;
- Pure horizontal, planr version 1.0 – 4 planr containers with 2 cores and 2 GB of memory each, one container having inside a planr application with 1 actor and a maximum heap of 0.5 GB;

Figure 8 briefly describes the previous points:

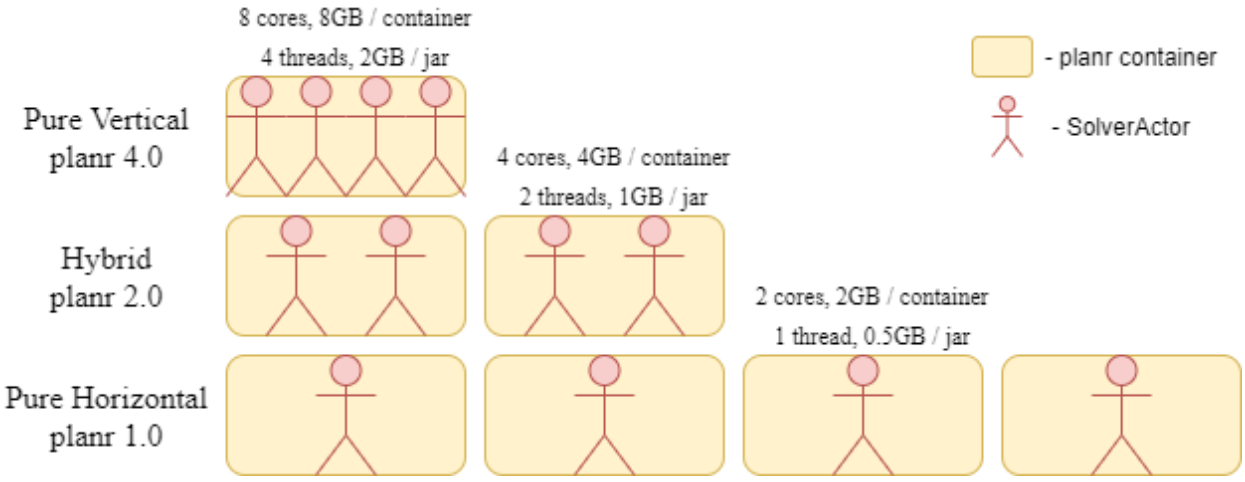


Figure 8: Scaling infrastructure

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

- Request version 4.0 – one request that contains the base Problem and a dayFrame of 4 days, expanding the request to 4 problems with 4 solutions;
- Request version 2.0 – two requests each containing the base Problem and a dayFrame of 2 days, expanding one request to 2 problems with 2 solutions;
- Request version 1.0 – four requests each containing the base Problem and a dayFrame of 1 day, expanding one request to 1 problem with 1 solution;

All the three request versions return the same results for day 1, 2, 3 and 4, the difference being how the requests with the problems are composed. Figure 9 serves for a better illustration:

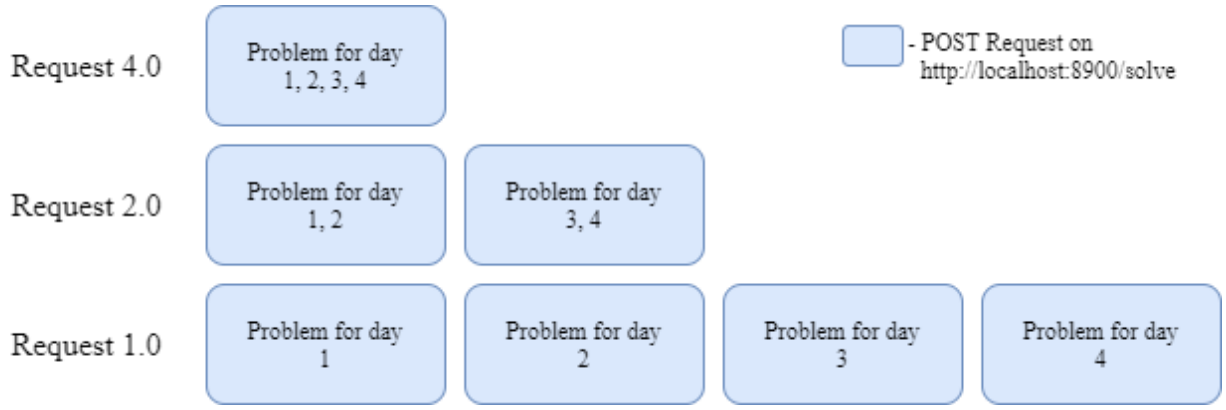


Figure 9: Test scenario requests

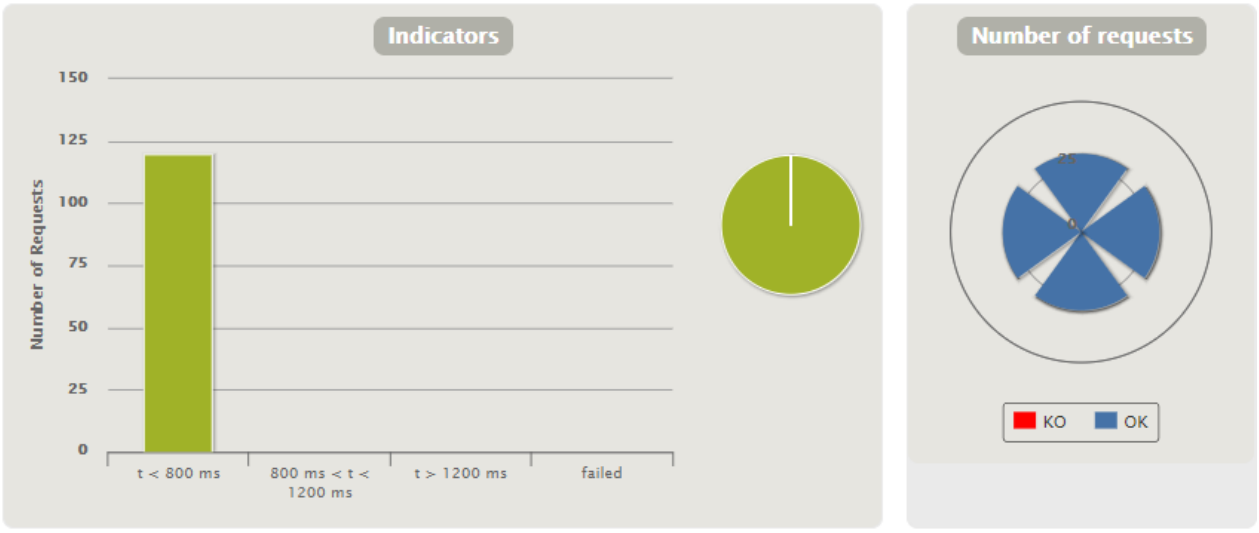
The base Problem is defined after the following properties:

- 7 operations of durations: 30, 20, 40, 35, 15, 45, 60 minutes and a group of resource keys of: 5, 6, 4, 4, 2, 1, 3;
- 18 candidate resources;
- 3 costs: asSoonAsPosssible, asTightAsPossible and preferredTimeInterval between 720 (12:00) and 1200 (20:00);
- 4 constraints: operationGrid of 10 minutes, sameResource for operation 1 and 7, and for operation 2 and 3, enforcedTimeInterval between 540 (9:00) and 1140 (19:00) and operationsRelation specifying the given layout;

The raw Problem definition can be found in the Appendix A section. The dayFrame constructs for each of the 4 days contains a program from 480 (8:00) to 1260 (21:00) with all the resources blocked between 780 (13:00) and 900 (15:00) with except of Resource 5.1, that is blocked between 780 (13:00) and 1020 (17:00). Other than that, just the day start and stop value changes in a dayFrame, since all time only defined fields are relative to a datetime field. The elaborated search context should assure that the constraint model has a wide and complicated search space, the variance of the domains submitting a considerable load on the system resources. The test combinations of the planr and request versions are presented alongside with their results in the Results section.

5. 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 the 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 informations are 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 infrastructure starts to be uncapable to handle the load. After each test execution the Gatling framework outputs a report as a result. Figure 10 is just a segment of such a report, containing the global and most relevant informations, like the average and maximum response times, respective the number of failed responses.



ASSERTIONS

Assertion	Status
Global: max of response time is less than or equal to 10000.0	OK

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	120	120	0	0%	4	15	19	83	94	611	705	55	108
Planr: /solve 1_1	30	30	0	0%	1	83	88	90	387	682	705	126	144
Planr: /solve 1_2	30	30	0	0%	1	15	18	20	30	406	558	37	97
Planr: /solve 1_3	30	30	0	0%	1	15	18	20	27	405	558	37	97
Planr: /solve 1_4	30	30	0	0%	1	15	18	20	28	33	34	19	4

Figure 10: Gatling report

5.1. Initial outcomes

The initial test scenarios are depicted in Table 1:

Table 1: 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

Each scenario result is composed of the following charts: the average response times and number of failed requests expressed in percentages. Other charts are available in the Appendix B section and these are the maximum response times, the number of total requests and the number of failed requests.

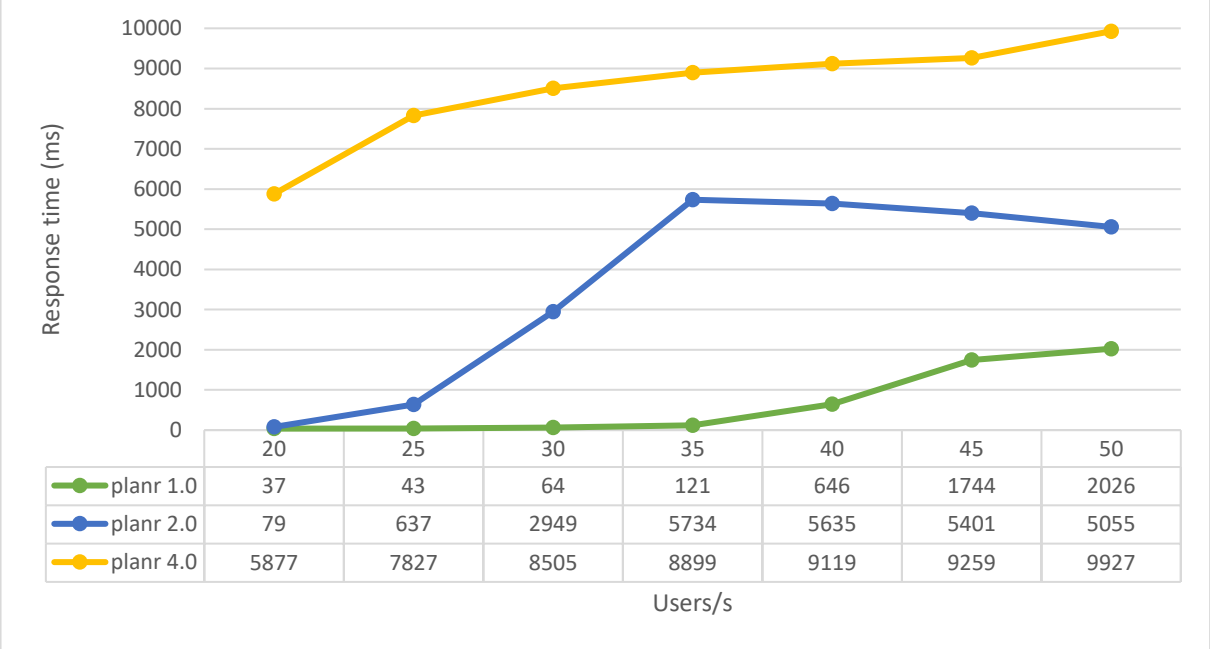
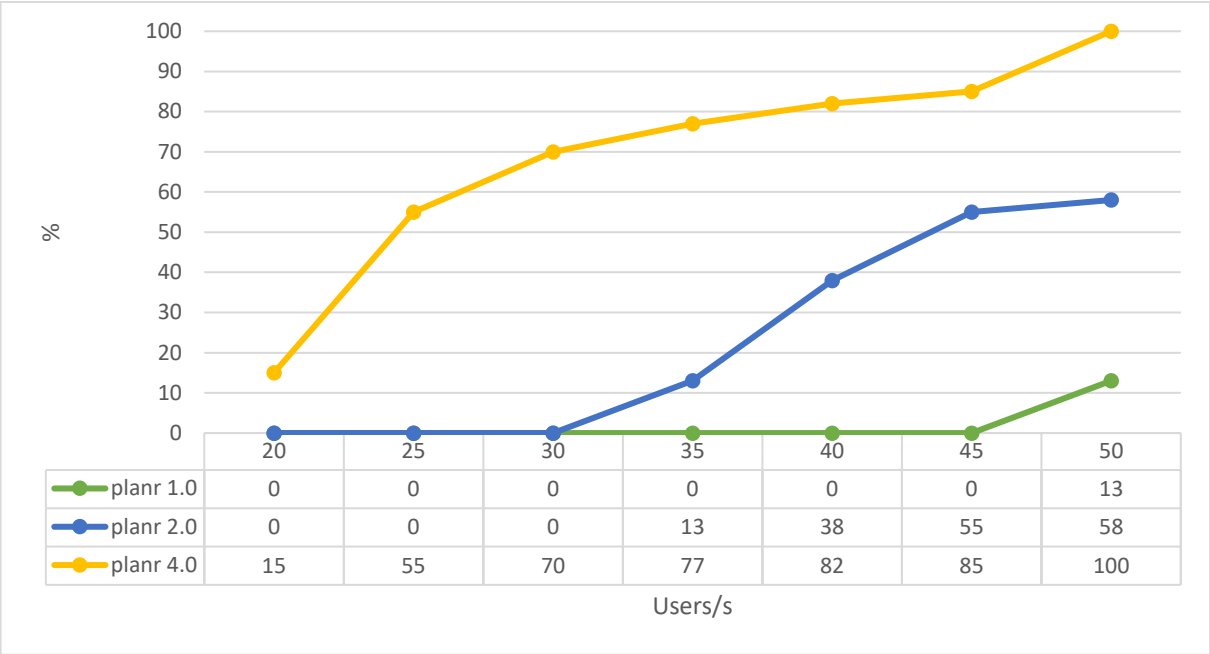
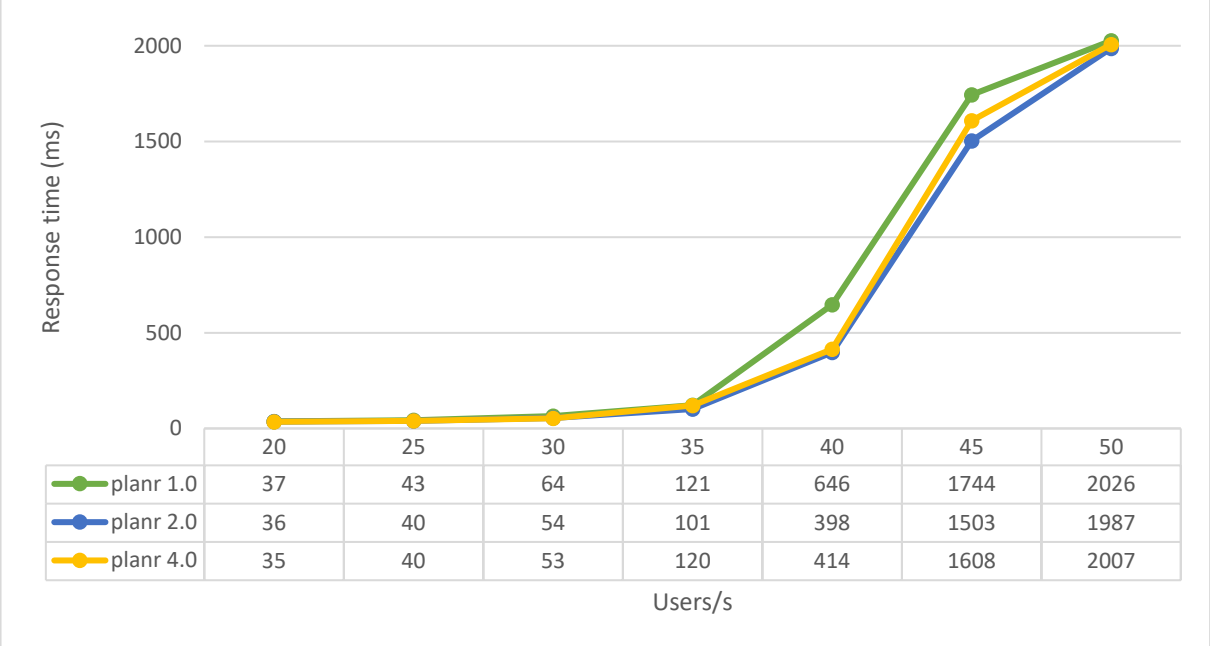


Figure 11: Scenario 1 - Average response time



	20	25	30	35	40	45	50
planr 1.0	0	0	0	0	0	0	13
planr 2.0	0	0	0	13	38	55	58
planr 4.0	15	55	70	77	82	85	100

Figure 12: Scenario 1 – Failed requests in percentage



	20	25	30	35	40	45	50
planr 1.0	37	43	64	121	646	1744	2026
planr 2.0	36	40	54	101	398	1503	1987
planr 4.0	35	40	53	120	414	1608	2007

Figure 13: Scenario 2 - Average response time

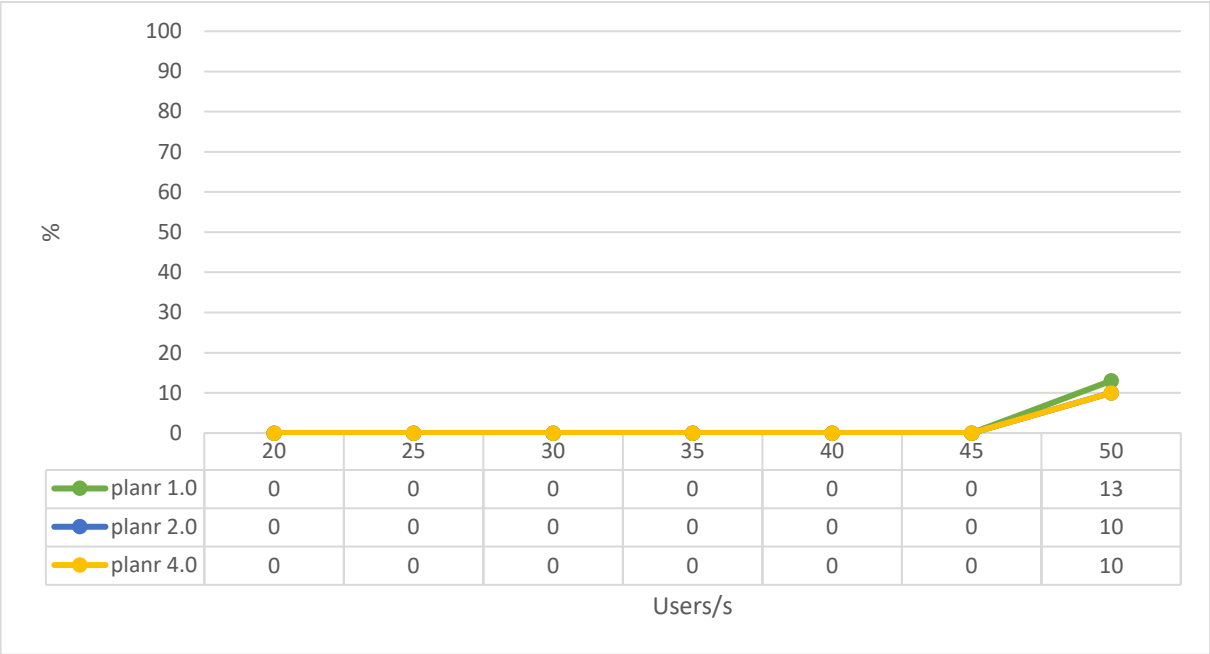


Figure 14: Scenario 2 - Failed requests in percentage

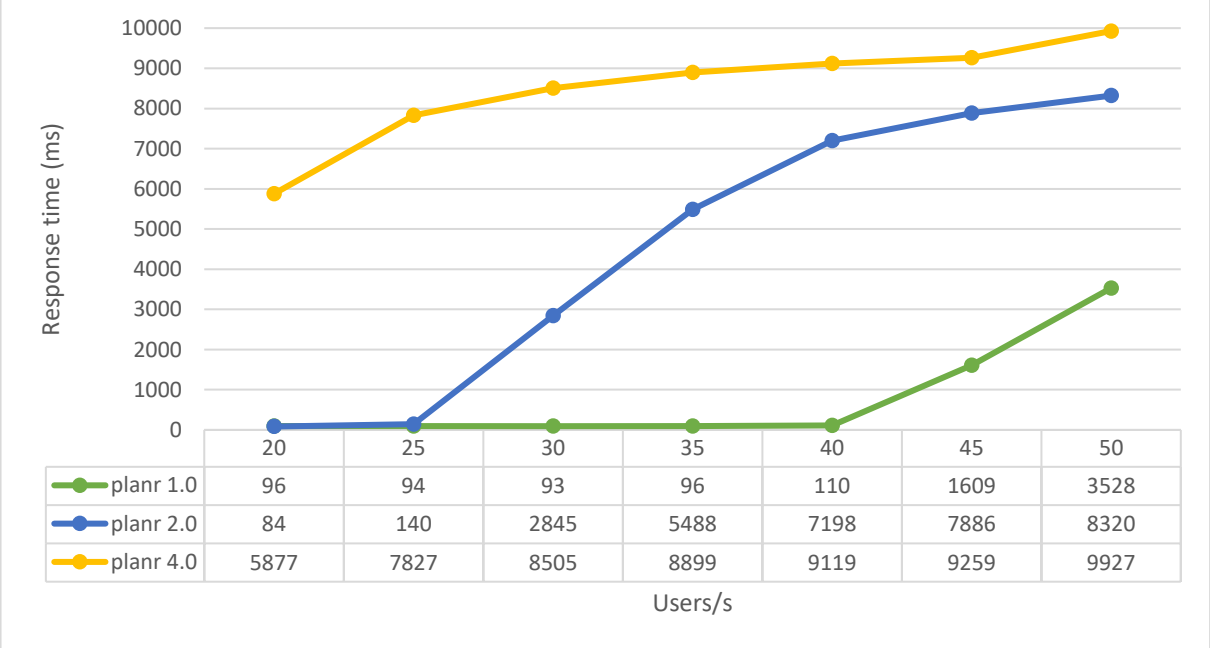


Figure 15: Scenario 3 - Average response time

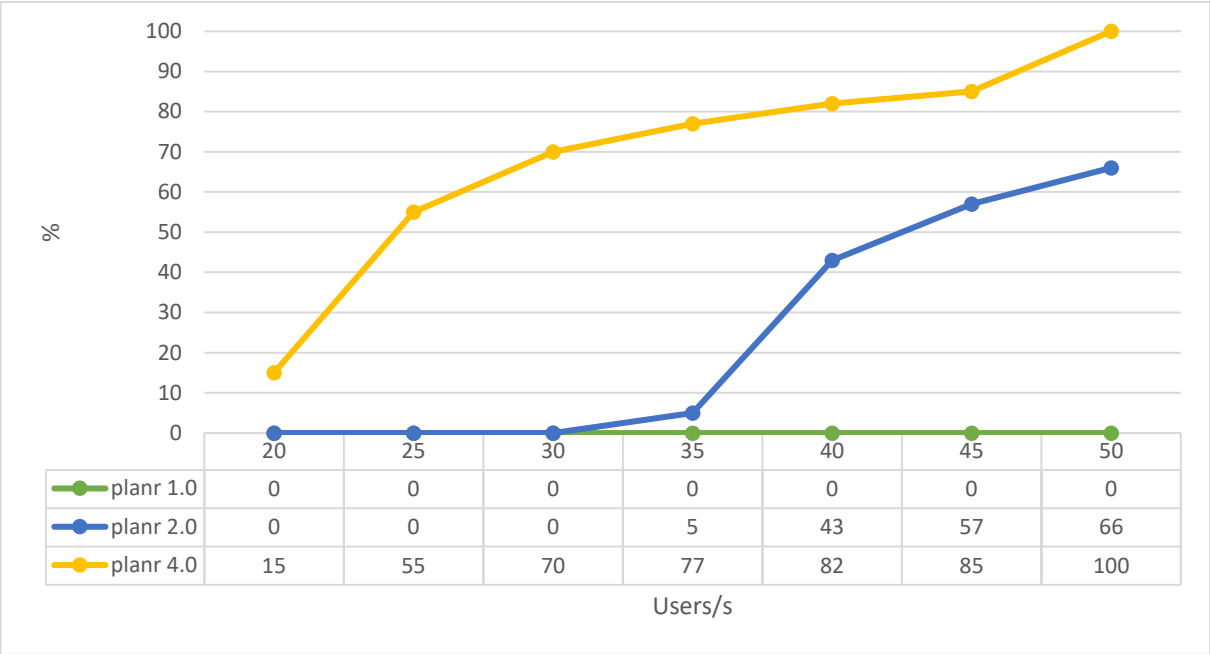


Figure 16: Scenario 3 - Failed requests in percentage

From Figure 11 to Figure 16 the results from the three scenarios can be seen. Overall planr 1.0, the pure horizontal model provides the best results in terms of average response times and number of failed requests in percentages. The other two scaling infrastructures can't keep up with the number of requests, the pure vertical model even failing right from the beginning. The hybrid model classes itself between the two, proving that the vertical scaling technique by itself is inefficient. The next subsection presents further benchmarks of the horizontal model combined with the vertical one.

5.2. Further improvements

Further test scenarios are depicted in Table 2:

Table 2: 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

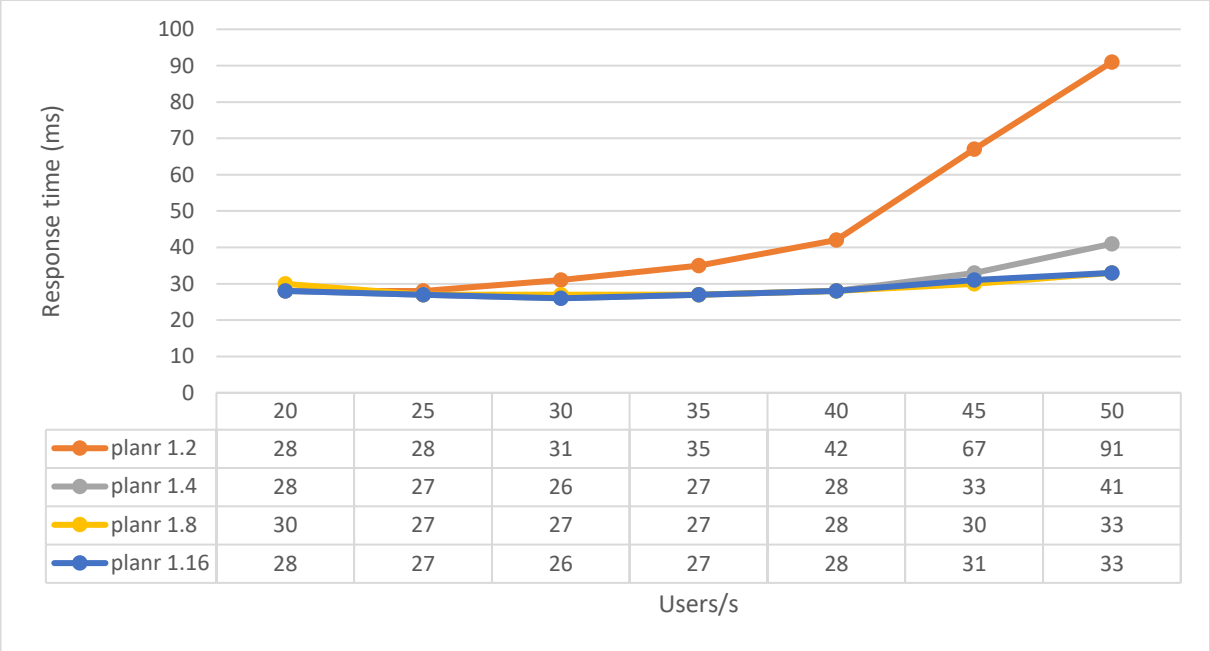
The new scenarios are derived from planr 1.0. The difference is that each new planr version contains vertical scaling with different parameters. The obtained versions are:

- planr version 1.2 – 4 planr containers with 2 cores and 2 GB of memory each, one container having inside a planr application with 2 actors and a maximum heap of 0.5

GB;

- planr version 1.4 – 4 planr containers with 2 cores and 2 GB of memory each, one container having inside a planr application with 4 actors and a maximum heap of 0.5 GB;
- planr version 1.8 – 4 planr containers with 2 cores and 2 GB of memory each, one container having inside a planr application with 8 actors and a maximum heap of 0.5 GB;
- planr version 1.16 – 4 planr containers with 2 cores and 2 GB of memory each, one container having inside a planr application with 16 actors and a maximum heap of 0.5 GB;

Each scenario results are composed by the same the charts as in the previous subsection.



	20	25	30	35	40	45	50
planr 1.2	28	28	31	35	42	67	91
planr 1.4	28	27	26	27	28	33	41
planr 1.8	30	27	27	27	28	30	33
planr 1.16	28	27	26	27	28	31	33

Figure 17: Scenario 4 - Average response time

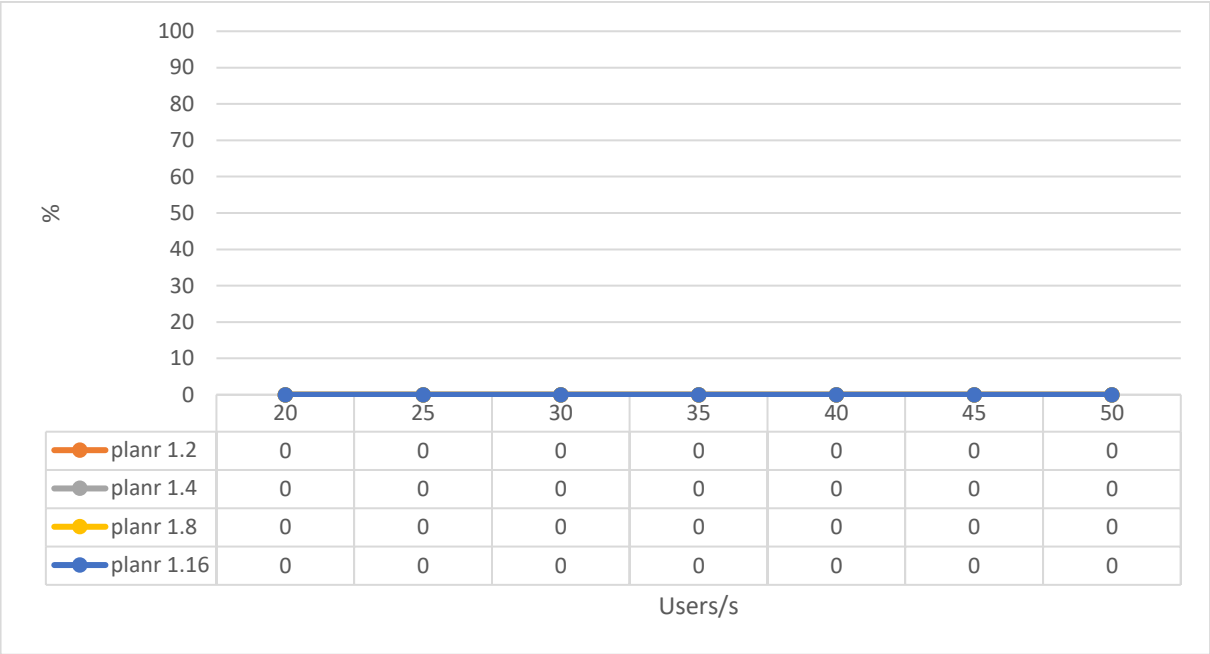


Figure 18: Scenario 4 - Failed requests in percentage

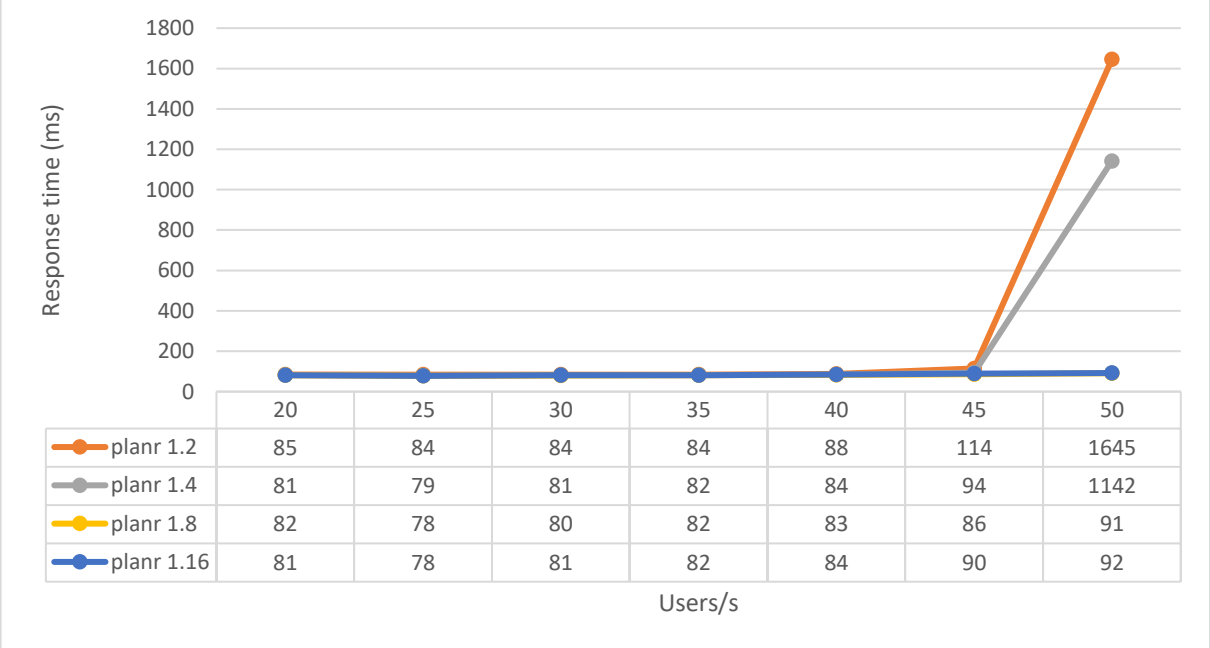


Figure 19: Scenario 5 - Average response time

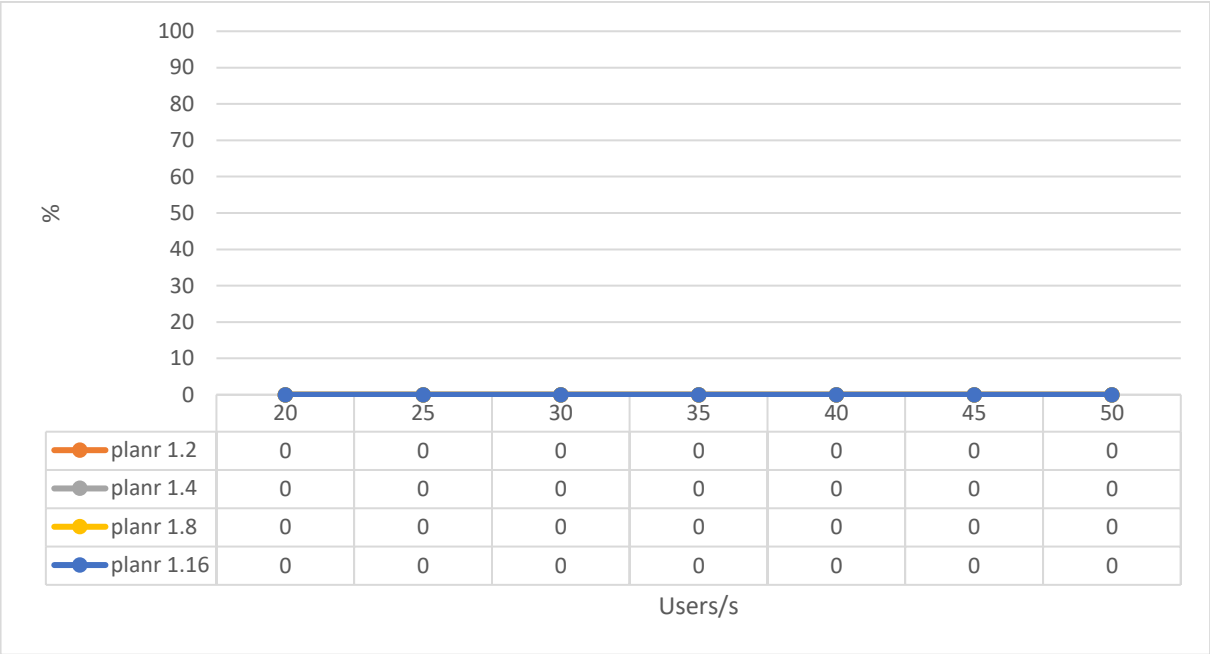


Figure 20: Scenario 5 - Failed requests in percentage

The results 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. Compared to the initial planr 1.0, the results are far better. The combination of the horizontal and vertical scaling provides the best use for heavy loads.

6. Conclusions

6.1. Achievements

The benchmark results bring the following achievements:

- The initial test scenarios confirm that horizontal strategy provides significantly better results than the vertical one. With the same resource pool, as cores and as memory, horizontal scaling proves to be more resilient and load resistant then its counterpart. The hybrid version was introduced to assure a more granular switching, in order to not miss out any relevant configuration details.
- The improved test scenarios are a superset of the pure horizontal strategy, and they demonstrate that the techniques applied together can further improve the results. The combination of the two maximize out the utilization of the allocated resources. The virtual entities mapping over the physical resources easily supports a multiple of 4, but a concrete definition is still undefined.

All the conclusions are valid only for the described context of frameworks and environment, since in some cases presented in the state of the art section there might be differences in performance between the applied strategies (like in Hadoop/MapReduce jobs). The process of elaborating the test scenarios and implementing the microservice also contributed with an architecture based on functional programming principles, a generic planification model written in terms of constraint programming, and offered some clear insights on how the design of web services is affected by external factors like the deployment.

6.2. State of the art comparison

This paper is a complementary value to the current state of the art and also serves as a comparison between two technologies when it comes to scaling and resource allocation, namely Akka and Docker. The fact that scale-out strategies outperforms its scale-up counterpart is already demonstrated in numerous works. Each of them brings different approaches, in some exceptional cases proving the contrary, but these are rare and apply on a very narrow use case. Scale-up techniques were recently gaining ground since hardware improvements, but they were not improved to substitute the scale-out techniques. The current field of research in this domain is already assuming that these strategies are applied together, in a hybrid model and the focus is currently shifted towards more intelligent decision capable load balancer. Not one paper introduces the concept of customized scaling models with a load balancer able to analyze a requested job and able to distribute it to the best fitting scaling model. Methods, such as neural networks and control theory are being explored in this direction. Elastic scaling methods are also debated and benchmarked for optimal resource utilization and cost minimization.

In the end, 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.

6.3. Future work

The mapping between virtual scale-up entities, such as the Akka actors and the physical

resources, mainly the CPU cores is still unclear. The relation is undefined because a correct response would include exploring how task scheduling and context switching between threads is performed on new generation hardware. Future work consists of a better understanding of the processes and hardware underneath the scale-up techniques. For better design and deployment in the cloud, web services would greatly benefit from a well-defined thread to core relation definition.

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] - Michael, Maged, et al. "Scale-up x scale-out: A case study using nutch/lucene." 2007 IEEE International Parallel and Distributed Processing Symposium. IEEE, 2007.

[8] - Appuswamy, Raja, et al. "Scale-up vs scale-out for hadoop: Time to rethink?." Proceedings of the 4th annual Symposium on Cloud Computing. 2013.

[9] - Hwang, Kai, Yue Shi, and Xiaoying Bai. "Scale-out vs. scale-up techniques for cloud performance and productivity." 2014 IEEE 6th International Conference on Cloud Computing Technology and Science. IEEE, 2014.

[10] - Hwang, Kai, et al. "Cloud performance modeling with benchmark evaluation of elastic scaling strategies." IEEE Transactions on parallel and distributed systems 27.1 (2015): 130-143.

[11] - Wang, Wenting, Le Xu, and Indranil Gupta. "Scale Up vs. scale out in cloud storage and graph processing systems." 2015 IEEE International Conference on Cloud Engineering. IEEE, 2015.

[12] - Odersky, Martin, Lex Spoon, and Bill Venners. Programming in scala. Artima Inc, 2008.

[13] - Scala documentation: https://www.scala-lang.org/

[14] - Suereth, Josh, and Matthew Farwell. SBT in Action: The simple Scala build tool. Manning Publications Co., 2015.

[15] - sbt documentation: https://www.scala-sbt.org/1.x/docs/

[16] - Hunt, John. "Play framework." A Beginner's Guide to Scala, Object Orientation and Functional Programming. Springer, Cham, 2018. 431-446.

[17] - Roestenburg, Raymond, Rob Bakker, and Rob Williams. Akka in action. Manning Publications Co., 2015.

[18] - Cats documentation: https://typelevel.org/cats/

[19] - Google OR-Tools documentation: https://developers.google.com/optimization/introduction/overview

[20] - Anderson, Charles. "Docker [software engineering]." IEEE Software 32.3 (2015): 102-c3.

[21] - Gatling documentation: https://gatling.io/docs/current

Appendix

A – Problems json


```
{
  "problem": {
    "key": "Performance4",
    "operations": [
      {
        "key": "Operation1",
        "name": "Operation 1",
        "duration": 30,
        "resourceKeys": [
          "Resource1.1",
          "Resource1.2",
          "Resource1.3",
          "Resource1.4",
          "Resource1.5"
        ]
      },
      {
        "key": "Operation2",
        "name": "Operation 2",
        "duration": 20,
        "resourceKeys": [
          "Resource2.1",
          "Resource2.2",
          "Resource2.3",
          "Resource2.4",
          "Resource2.5",
          "Resource2.6"
        ]
      },
      {
        "key": "Operation3",
        "name": "Operation 3",
        "duration": 40,
        "resourceKeys": [
          "Resource2.3",
          "Resource2.4",
          "Resource2.5",
          "Resource2.6"
        ]
      },
      {
```

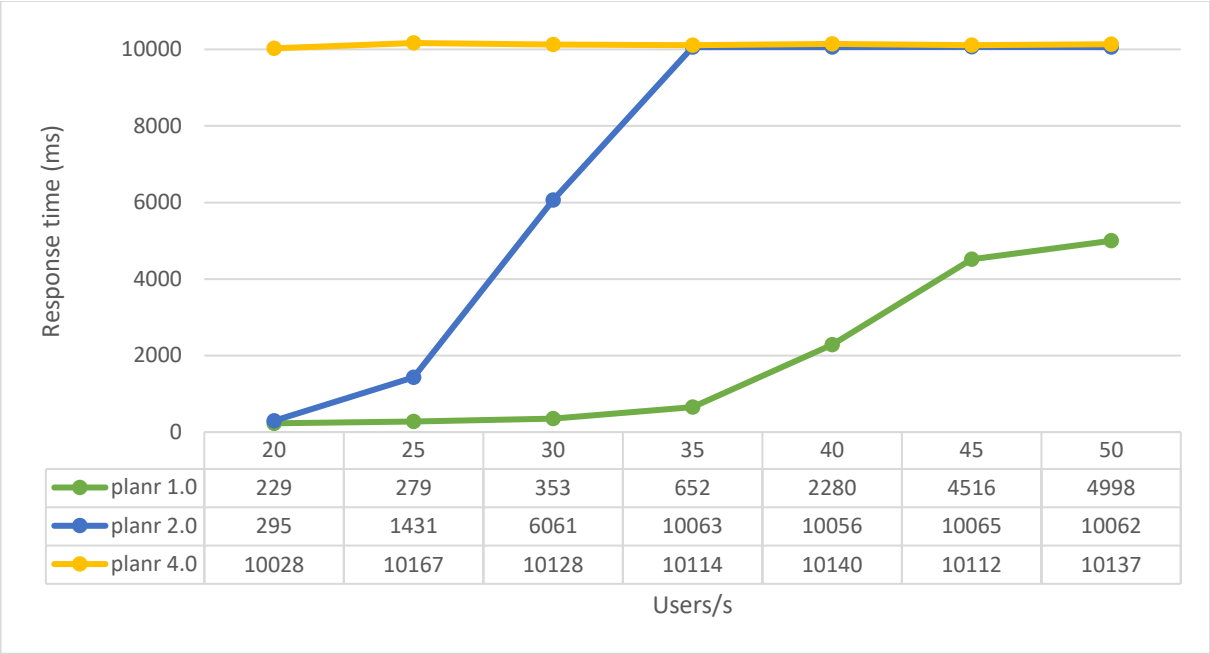
```
      "key": "Operation4",
      "name": "Operation 4",
      "duration": 35,
      "resourceKeys": [
        "Resource3.1",
        "Resource3.2",
        "Resource3.3",
        "Resource3.4"
      ]
    },
    {
      "key": "Operation5",
      "name": "Operation 5",
      "duration": 15,
      "resourceKeys": [
        "Resource4.1",
        "Resource4.2"
      ]
    },
    {
      "key": "Operation6",
      "name": "Operation 6",
      "duration": 45,
      "resourceKeys": [
        "Resource5.1"
      ]
    },
    {
      "key": "Operation7",
      "name": "Operation 7",
      "duration": 60,
      "resourceKeys": [
        "Resource1.2",
        "Resource1.4",
        "Resource1.5"
      ]
    }
  ],
  "resources": [
    {
      "key": "Resource1.1",
      "name": "Resource 1.1"
    },
    {
      "key": "Resource1.2",
      "name": "Resource 1.2"
```

```
88      },
89      {
90        "key": "Resource1.3",
91        "name": "Resource 1.3"
92      },
93      {
94        "key": "Resource1.4",
95        "name": "Resource 1.4"
96      },
97      {
98        "key": "Resource1.5",
99        "name": "Resource 1.5"
100     },
101     {
102       "key": "Resource2.1",
103       "name": "Resource 2.1"
104     },
105     {
106       "key": "Resource2.2",
107       "name": "Resource 2.2"
108     },
109     {
110       "key": "Resource2.3",
111       "name": "Resource 2.3"
112     },
113     {
114       "key": "Resource2.4",
115       "name": "Resource 2.4"
116     },
117     {
118       "key": "Resource2.5",
119       "name": "Resource 2.5"
120     },
121     {
122       "key": "Resource2.6",
123       "name": "Resource 2.6"
124     },
125     {
126       "key": "Resource3.1",
127       "name": "Resource 3.1"
128     },
129     {
130       "key": "Resource3.2",
131       "name": "Resource 3.2"
132     },
133     {
```

```
      "key": "Resource3.3",
      "name": "Resource 3.3"
    },
    {
      "key": "Resource3.4",
      "name": "Resource 3.4"
    },
    {
      "key": "Resource4.1",
      "name": "Resource 4.1"
    },
    {
      "key": "Resource4.2",
      "name": "Resource 4.2"
    },
    {
      "key": "Resource5.1",
      "name": "Resource 5.1"
    }
  ],
  "costs": {
    "asSoonAsPossible": true,
    "asTightAsPossible": true,
    "preferredTimeInterval": {
      "startT": 720,
      "stopT": 1200
    }
  },
  "constraints": {
    "operationGrid": 10,
    "sameResource": [
      [
        "Operation1",
        "Operation7"
      ],
      [
        "Operation2",
        "Operation3"
      ]
    ],
    "enforcedTimeInterval": {
      "startT": 540,
      "stopT": 1140
    },
    "operationsRelation": [
      {
```

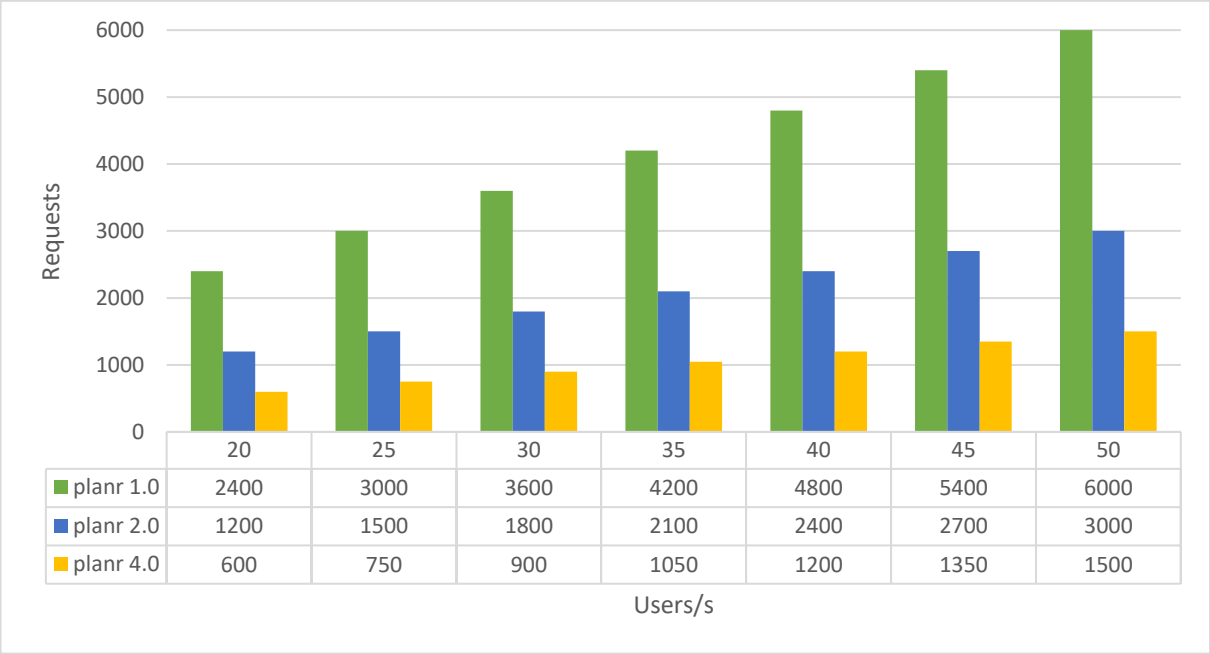
```
        "opRelType": "STARTS_AFTER_END",
        "opKey1": "Operation2",
        "opKey2": "Operation1"
      },
      {
        "opRelType": "STARTS_AT_END",
        "opKey1": "Operation3",
        "opKey2": "Operation2"
      },
      {
        "opRelType": "STARTS_AFTER_END",
        "opKey1": "Operation4",
        "opKey2": "Operation3"
      },
      {
        "opRelType": "STARTS_AFTER_START",
        "opKey1": "Operation5",
        "opKey2": "Operation4"
      },
      {
        "opRelType": "STARTS_AT_START",
        "opKey1": "Operation6",
        "opKey2": "Operation4"
      },
      {
        "opRelType": "STARTS_AFTER_END",
        "opKey1": "Operation7",
        "opKey2": "Operation4"
      },
      {
        "opRelType": "STARTS_AFTER_END",
        "opKey1": "Operation7",
        "opKey2": "Operation5"
      },
      {
        "opRelType": "STARTS_AFTER_END",
        "opKey1": "Operation7",
        "opKey2": "Operation6"
      }
    ]
  }
 }
}
```

B – Additional measurements



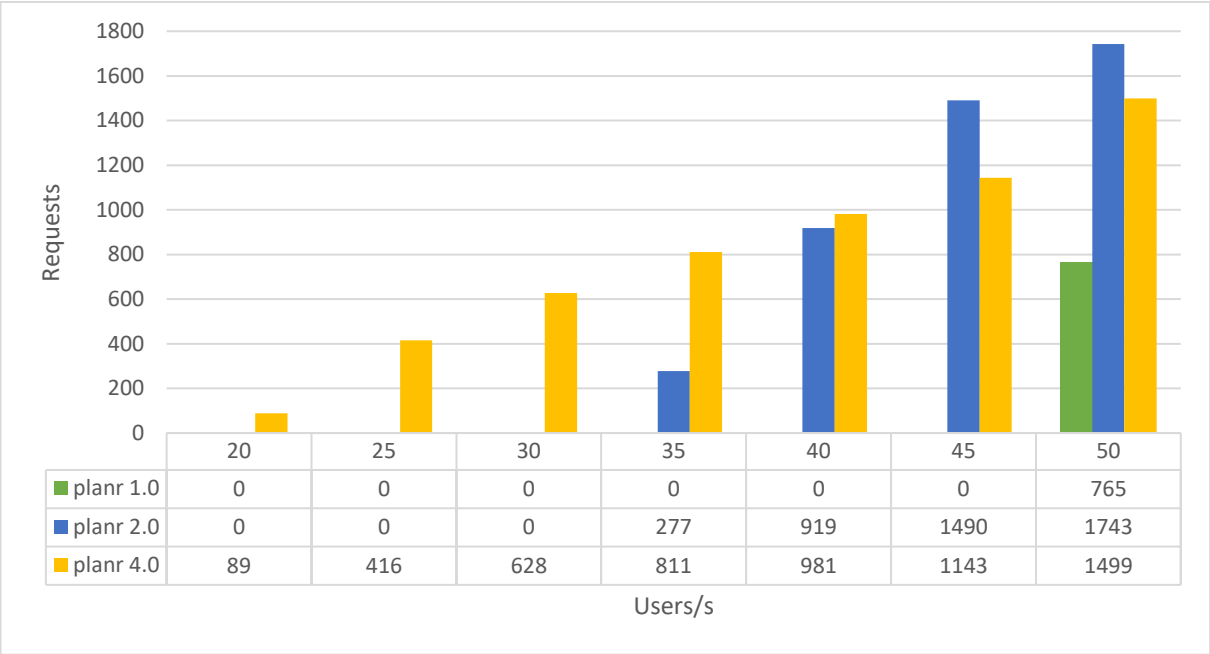
	20	25	30	35	40	45	50
planr 1.0	229	279	353	652	2280	4516	4998
planr 2.0	295	1431	6061	10063	10056	10065	10062
planr 4.0	10028	10167	10128	10114	10140	10112	10137

Figure 21: Scenario 1 - Maximum response time



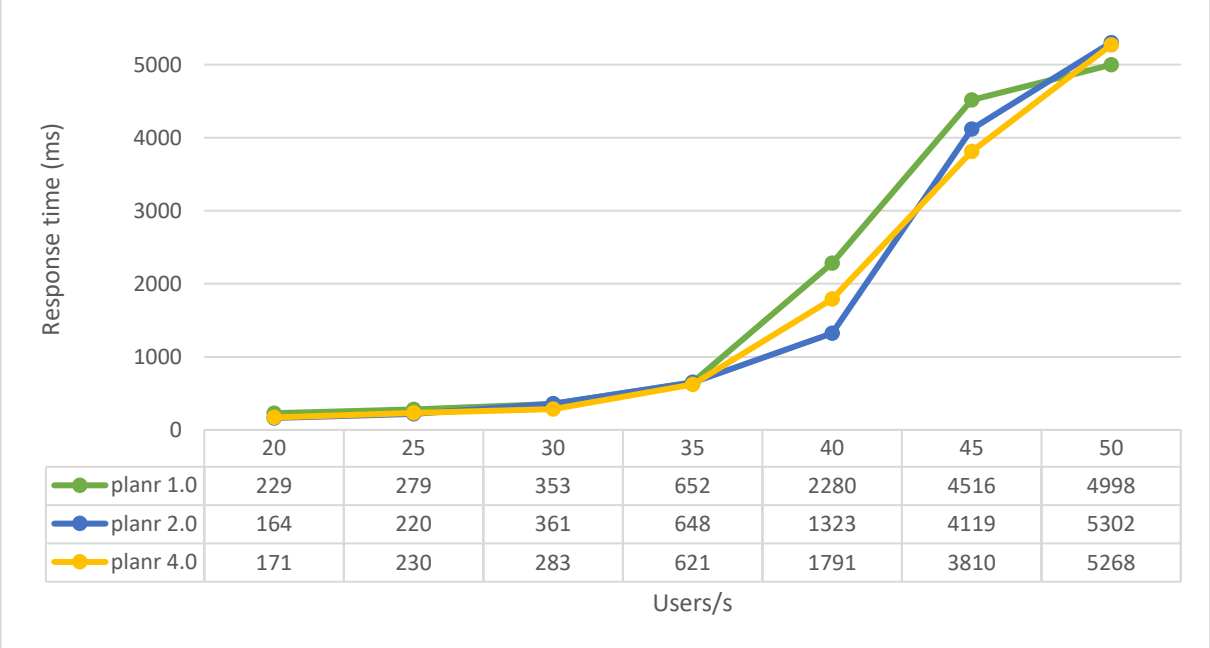
	20	25	30	35	40	45	50
planr 1.0	2400	3000	3600	4200	4800	5400	6000
planr 2.0	1200	1500	1800	2100	2400	2700	3000
planr 4.0	600	750	900	1050	1200	1350	1500

Figure 22: Scenario 1 - Total requests



	20	25	30	35	40	45	50
planr 1.0	0	0	0	0	0	0	765
planr 2.0	0	0	0	277	919	1490	1743
planr 4.0	89	416	628	811	981	1143	1499

Figure 23: Scenario 1 - Failed requests



	20	25	30	35	40	45	50
planr 1.0	229	279	353	652	2280	4516	4998
planr 2.0	164	220	361	648	1323	4119	5302
planr 4.0	171	230	283	621	1791	3810	5268

Figure 24: Scenario 2 - Maximum response time

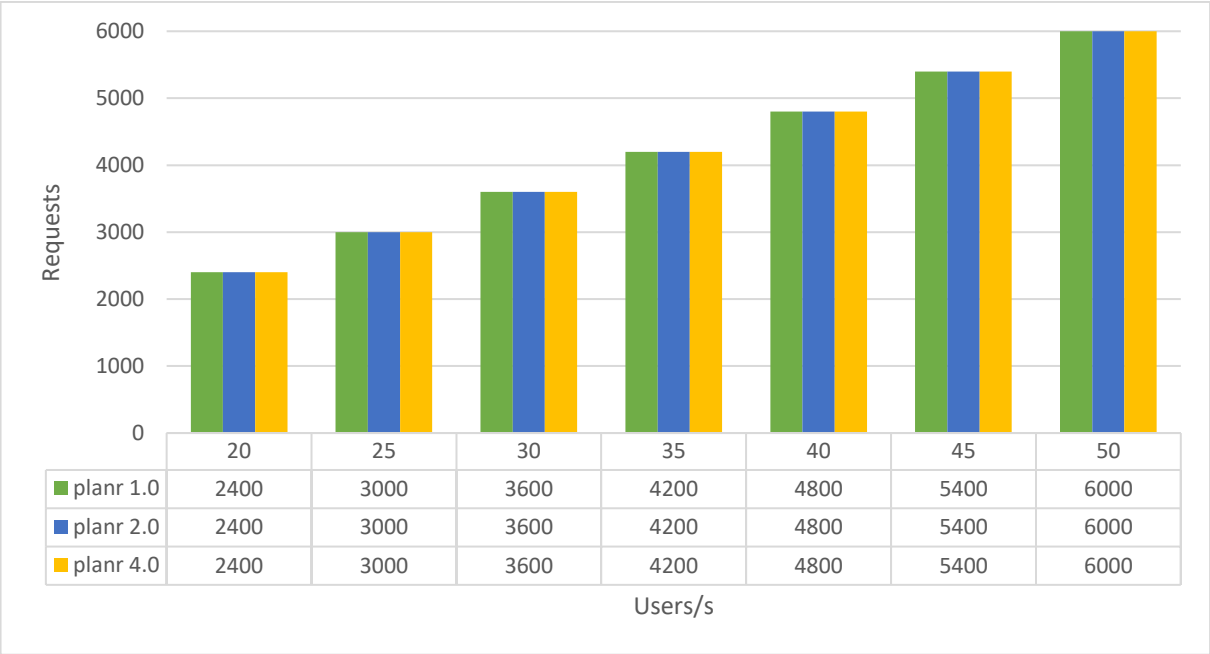


Figure 25: Scenario 2 - Total requests

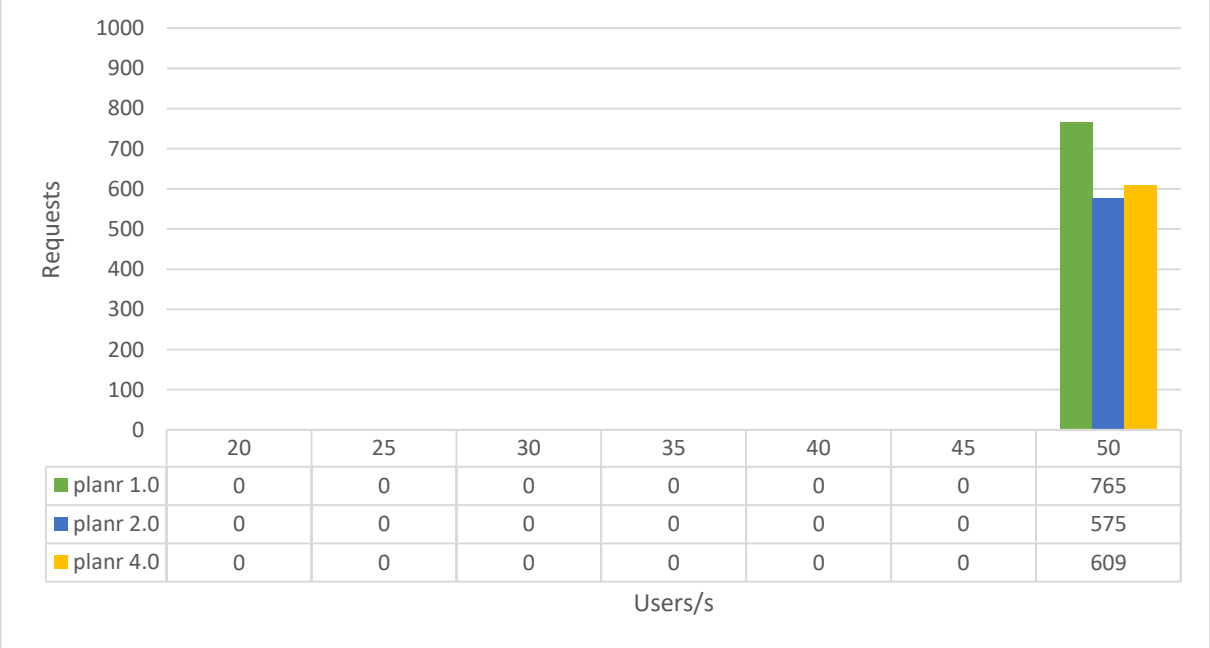
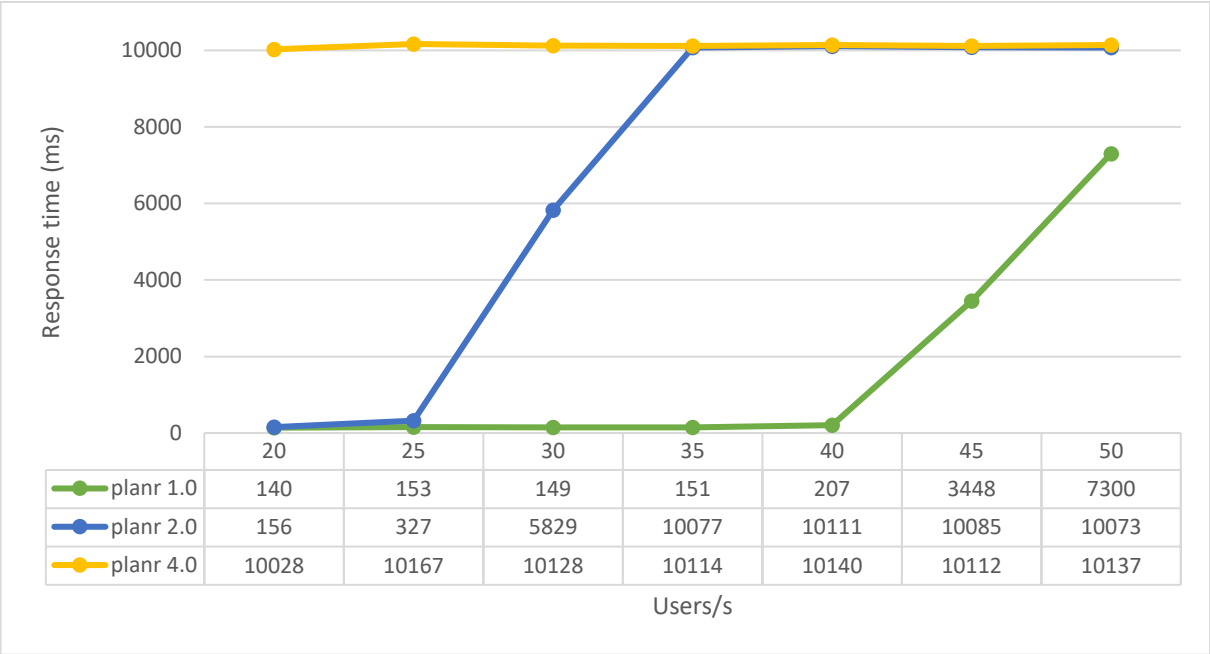


Figure 26: Scenario 2 - Failed requests

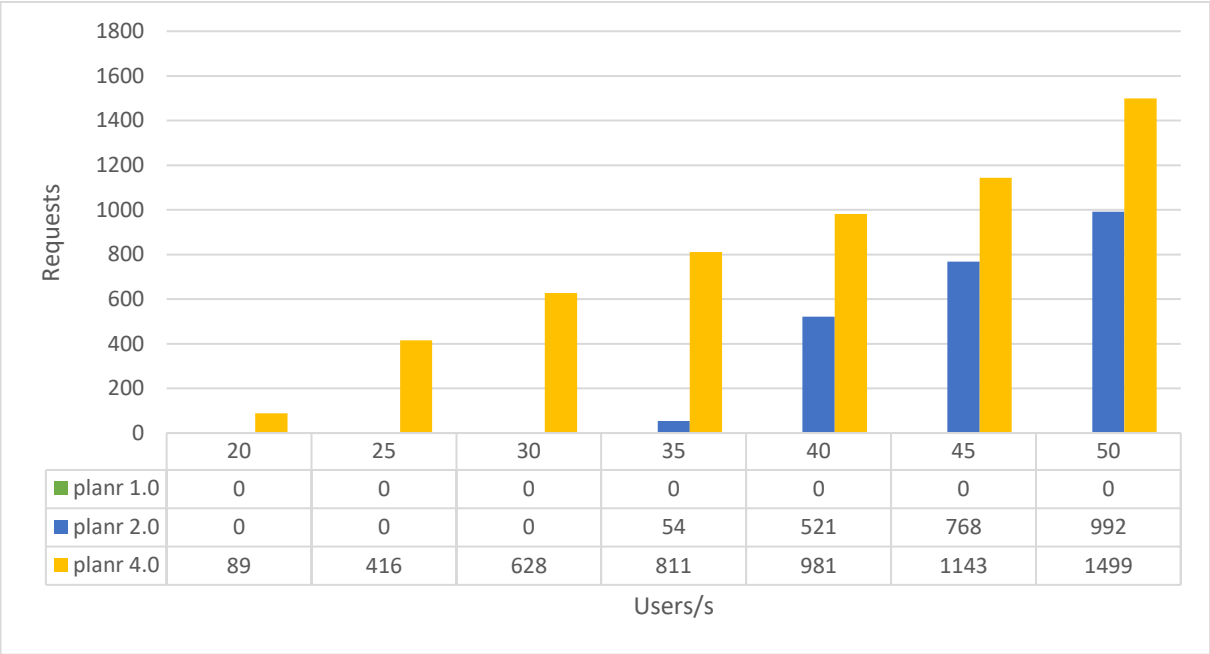


	20	25	30	35	40	45	50
planr 1.0	140	153	149	151	207	3448	7300
planr 2.0	156	327	5829	10077	10111	10085	10073
planr 4.0	10028	10167	10128	10114	10140	10112	10137

Figure 27: Scenario 3 - Maximum response time

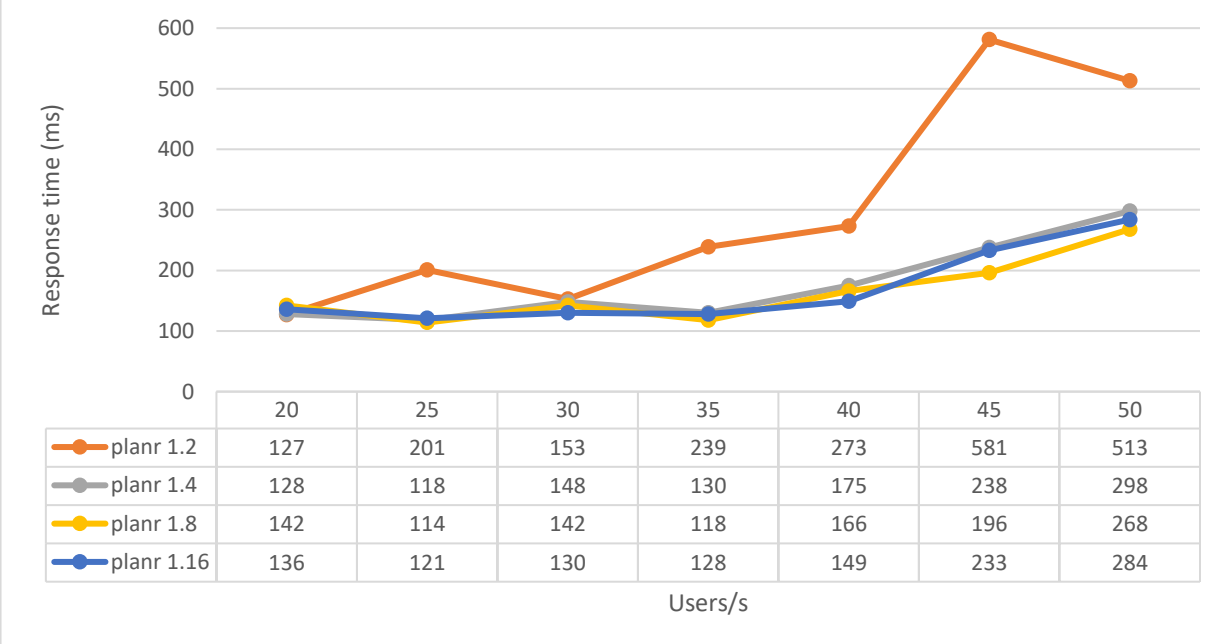
	20	25	30	35	40	45	50
planr 1.0	600	750	900	1050	1200	1350	1500
planr 2.0	600	750	900	1050	1200	1350	1500
planr 4.0	600	750	900	1050	1200	1350	1500

Figure 28: Scenario 3 - Total requests



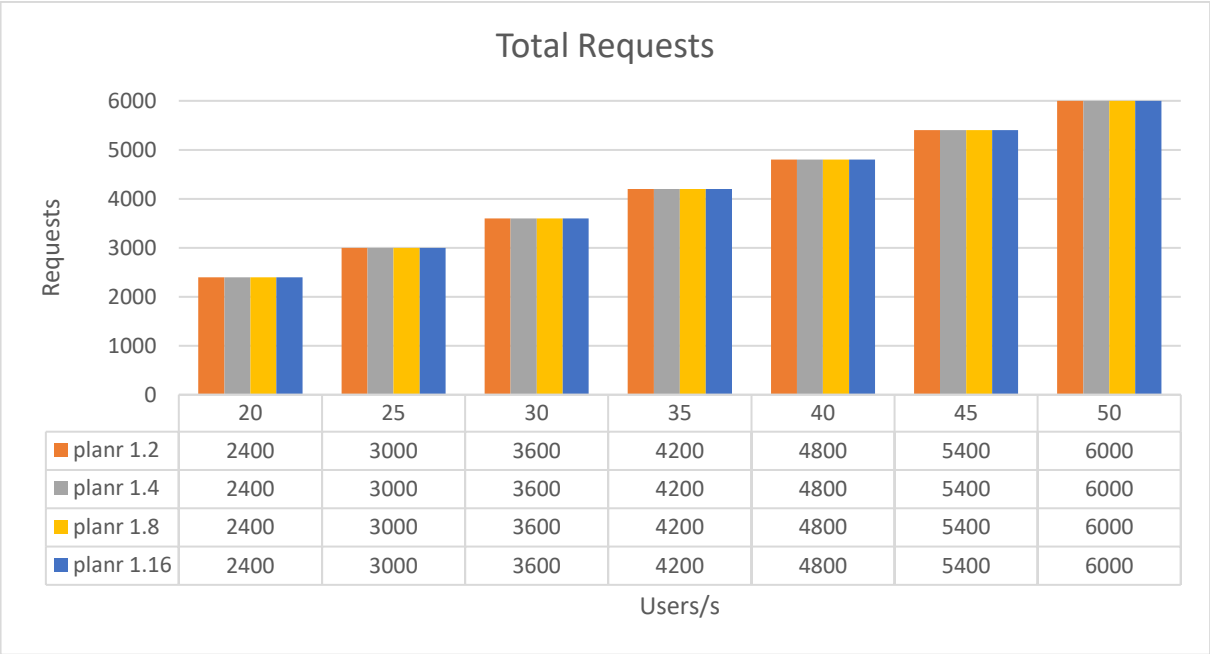
	20	25	30	35	40	45	50
planr 1.0	0	0	0	0	0	0	0
planr 2.0	0	0	0	54	521	768	992
planr 4.0	89	416	628	811	981	1143	1499

Figure 29: Scenario 3 - Failed requests



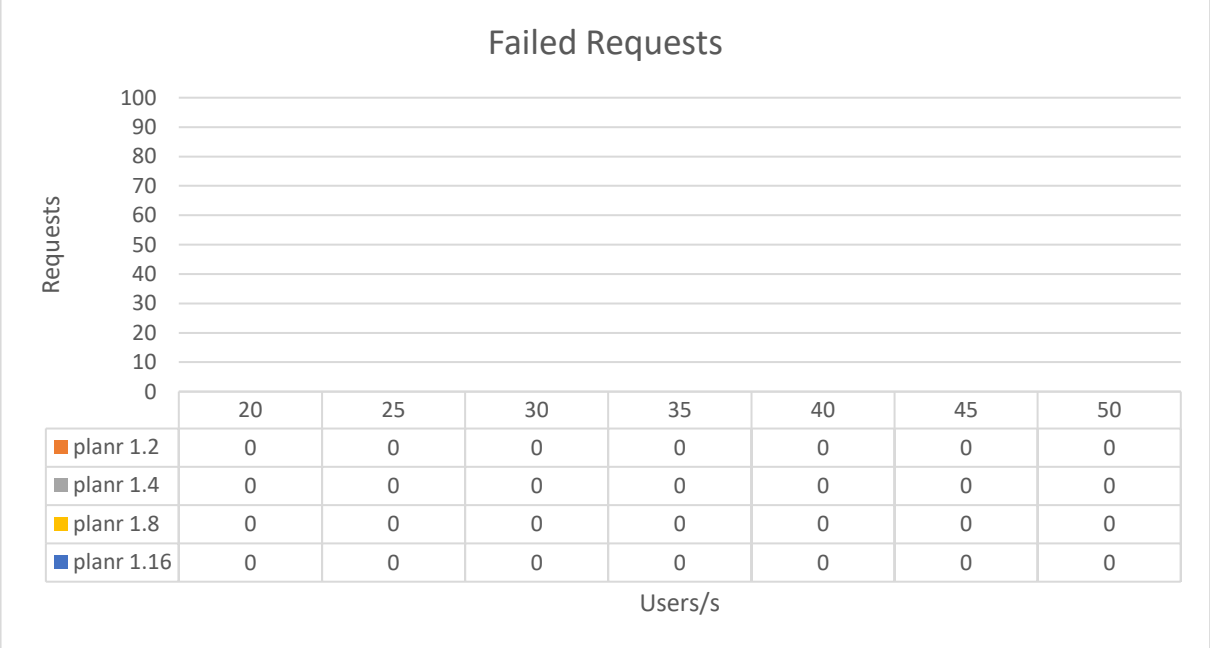
	20	25	30	35	40	45	50
planr 1.2	127	201	153	239	273	581	513
planr 1.4	128	118	148	130	175	238	298
planr 1.8	142	114	142	118	166	196	268
planr 1.16	136	121	130	128	149	233	284

Figure 30: Scenario 4 - Maximum response time



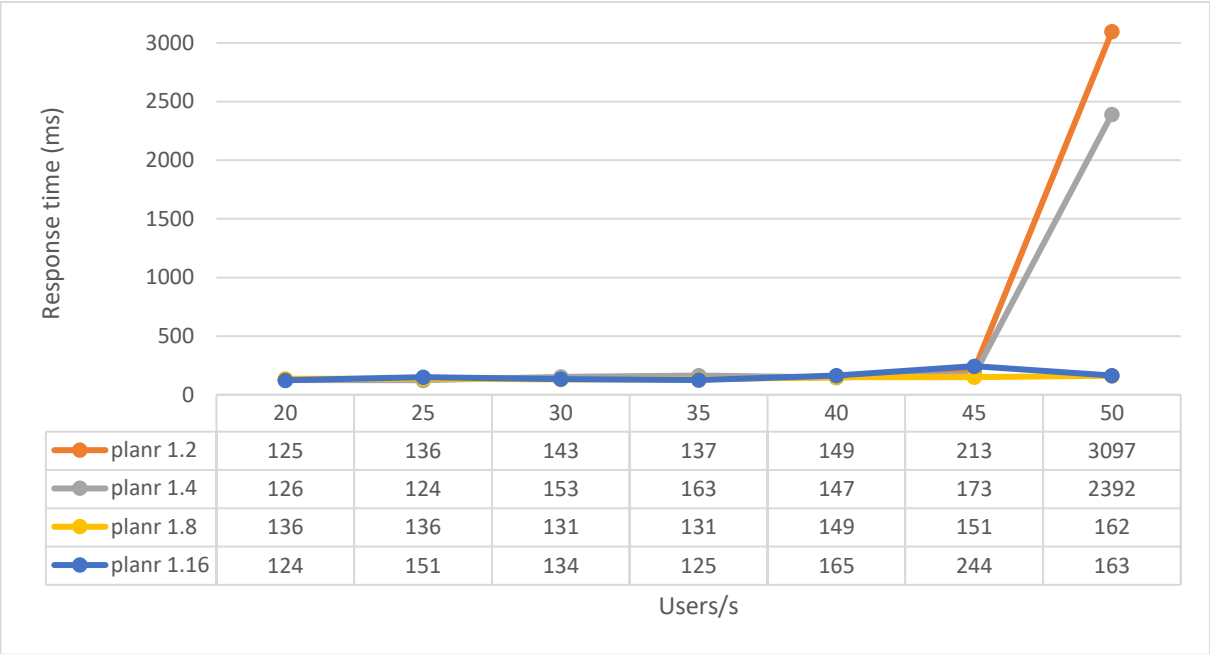
	20	25	30	35	40	45	50
planr 1.2	2400	3000	3600	4200	4800	5400	6000
planr 1.4	2400	3000	3600	4200	4800	5400	6000
planr 1.8	2400	3000	3600	4200	4800	5400	6000
planr 1.16	2400	3000	3600	4200	4800	5400	6000

Figure 31: Scenario 4 - Total requests



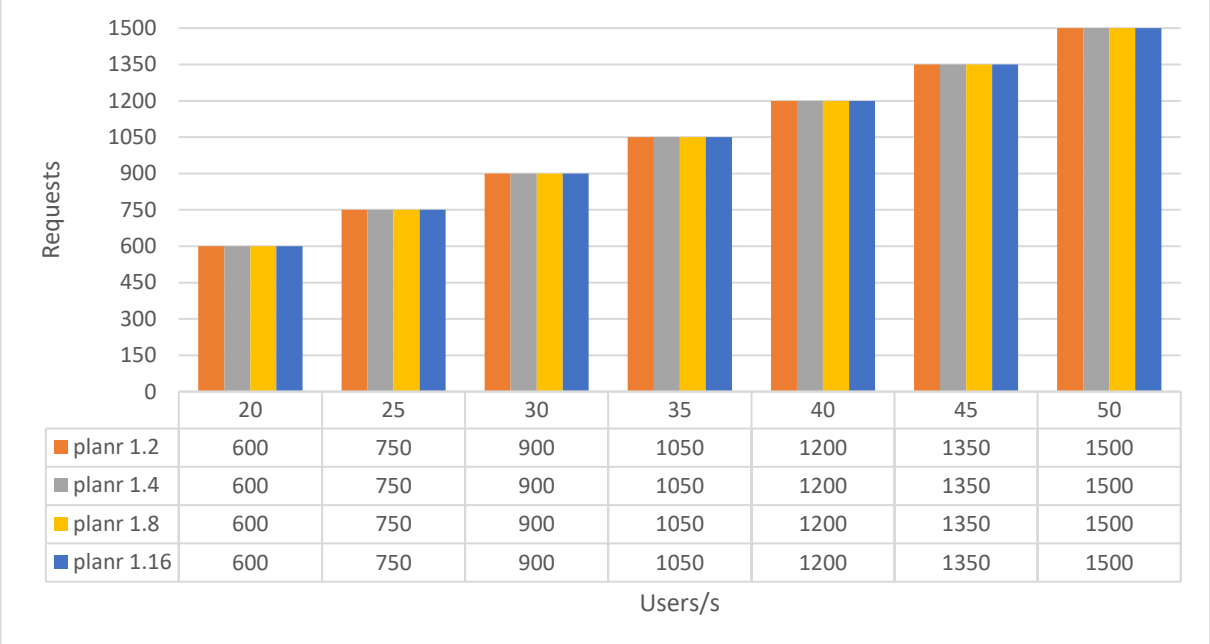
	20	25	30	35	40	45	50
planr 1.2	0	0	0	0	0	0	0
planr 1.4	0	0	0	0	0	0	0
planr 1.8	0	0	0	0	0	0	0
planr 1.16	0	0	0	0	0	0	0

Figure 32: Scenario 4 - Failed requests



	20	25	30	35	40	45	50
planr 1.2	125	136	143	137	149	213	3097
planr 1.4	126	124	153	163	147	173	2392
planr 1.8	136	136	131	131	149	151	162
planr 1.16	124	151	134	125	165	244	163

Figure 33: Scenario 5 - Maximum response time



	20	25	30	35	40	45	50
planr 1.2	600	750	900	1050	1200	1350	1500
planr 1.4	600	750	900	1050	1200	1350	1500
planr 1.8	600	750	900	1050	1200	1350	1500
planr 1.16	600	750	900	1050	1200	1350	1500

Figure 34: Scenario 5 - Total requests

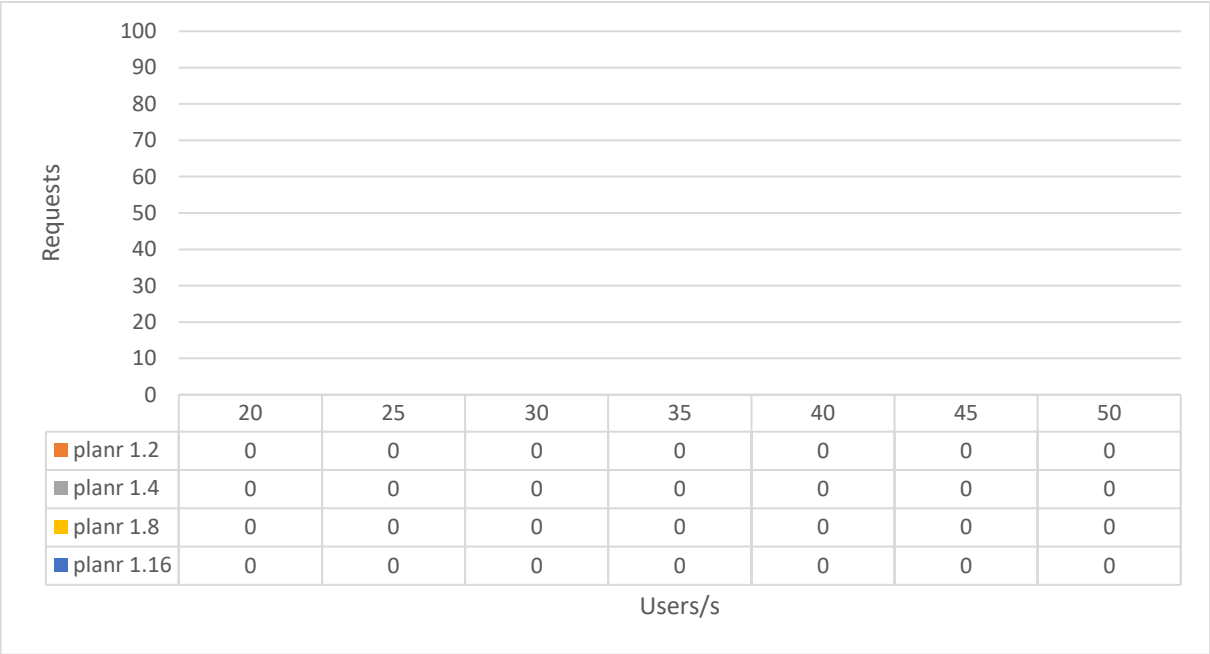


Figure 35: Scenario 5 - Failed requests