University of Padua

Dipartimento di Matematica "Tullio Levi-Civita"

Master's Degree Computer Science

# Go Microservices Runtime Optimization in Kubernetes Environment: the Importance of Garbage Collection Tuning

*Master Thesis*

*Supervisor*

Prof.ssa Silvia Crafa

*Co-Supervisor*

Stefano Cereda

*Author*

Stefano Panozzo

Academic Year 2021-2022

# Abstract

In performance optimization, automatic parameter tuning is a thriving topic. The standard approach consists of having an expert able to improve performances of an IT system using his expertise in a manual way. In a full IT stack system there are hundreds or thousands of knobs to be tuned belonging to the different components of the stack, such as databases, webservers or load balancers. This large number of parameters makes it difficult for human beings to completely understand all the possible changes involved in changing a parameter.
In this thesis, we try to create this knowledge, so a human expert or a machine can understand what a parameter change can involve. We investigate how the Go runtime works, especially its garbage collector and the goroutine scheduler.
Thus, we run a benchmark application and investigate how the tuning of different parameters, both of Go runtime and the Kubernetes environment where the application is deployed, influence the resource utilization and the performance of the system.
From this work we are able to understand how the parameters tuning of the Go language and the Kubernetes orchestrator influence the working application and set boundaries to the parameters domain, so as to narrow the optimization space for future experiments.

***Keywords*** Performance Optimization, Load Testing, Automatic Tuning, Golang, Kubernetes

# Contents

**1 Introduction** **1**

**2 Background** **7**

2.1 Performance Optimization . . . 7
2.1.1 Parameters . . . 8
2.1.2 Performance Metrics . . . 8
2.1.3 Workload . . . 9
2.1.4 Offline and Online Performance Testing . . . 9
2.2 Go language . . . 10
2.2.1 Go scheduler and goroutine . . . 10
2.2.2 Main differences between goroutines and OS threads . . . 11
2.2.3 Types of garbage collectors . . . 12
2.2.4 Go memory management and garbage collector . . . 14
2.3 System Virtualization . . . 17
2.3.1 Hypervisor Virtualization . . . 18
2.3.2 Container Virtualization . . . 19
2.4 Docker Container Engine . . . 20
2.4.1 Docker for Development and Operations . . . 20
2.4.2 Docker Architecture and Components . . . 21
2.5 Container Orchestration Frameworks . . . 23
2.6 Kubernetes . . . 23
2.6.1 Kubernetes Architecture . . . 23
2.6.2 Master Node Components . . . 25
2.6.3 Worker Node Components . . . 26
2.7 Service Mesh . . . 27
2.7.1 Istio . . . 28

**3 Benchmark application and dataset** **33**

3.1 Benchmark application and workload generator . . . 33
3.1.1 The benchmark . . . 34
3.2 Amazon Web Services . . . 36
3.3 The parameters . . . 37
3.3.1 Go . . . 38
3.3.2 Kubernetes . . . 39
3.4 Akamas . . . 43
3.4.1 SOBOL . . . 43
3.4.2 Experiment pipeline . . . 44

3.4.3 Workload generator parameters . . . . . . . . . . . . . . . . . 46

**4 Results 51**

4.1 Preliminary analysis . . . . . . . . . . . . . . . . . . . . . . . . . . 52

4.2 First experiments results . . . . . . . . . . . . . . . . . . . . . . . . 52

4.3 Goroutine and GOMAXPROCS . . . . . . . . . . . . . . . . . . . . 54

4.4 GOGC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

4.5 Throttle time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

4.6 Resource usage and costs . . . . . . . . . . . . . . . . . . . . . . . . 64

**5 Conclusion 69**

**Bibliography 75**

# List of Figures

1.1 Different conditions needs different settings. . . . . . . . . . . . . . . 1

2.1 Performance test process. . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2 Go runtime and OS kernel interaction [33]. . . . . . . . . . . . . . . . 10
2.3 Go workflow graph [34]. . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.4 GC phases: black objects are still in use, white objects are ready to be cleaned up and gray objects still need to be categorized as either black or white. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.5 Types of hypervisor-based virtualization [79]. . . . . . . . . . . . . . . 18
2.6 Container-based virtualization architecture [12]. . . . . . . . . . . . . 19
2.7 Docker Engine architecture [20]. . . . . . . . . . . . . . . . . . . . . . 21
2.8 High-level Kubernetes architecture [37]. . . . . . . . . . . . . . . . . . 24
2.9 Kubernetes master node [37]. . . . . . . . . . . . . . . . . . . . . . . . 26
2.10 Kubernetes worker nodes [37]. . . . . . . . . . . . . . . . . . . . . . . 27
2.11 Service mesh architecture overview [72]. . . . . . . . . . . . . . . . . . 28
2.12 Istio architecture overview [40]. . . . . . . . . . . . . . . . . . . . . . 29

3.1 Prometheus architecture. . . . . . . . . . . . . . . . . . . . . . . . . . 36
3.2 Distribution of GOGC parameters. . . . . . . . . . . . . . . . . . . . . 39
3.3 Distribution of GOMAXPROCS parameters. . . . . . . . . . . . . . . 40
3.4 Distribution of Kubernetes memory requests and limits parameters. 42
3.5 Distribution of Kubernetes CPU requests and limits parameters. . . 43
3.6 Lower bound of the time needed to obtain a full dataset based on the total number of parameters. . . . . . . . . . . . . . . . . . . . . . 44
3.7 Exploring a bidimensional 1x1 space using a Pseudorandom or Sobol. 45
3.8 Median latency for default parameters without Istio. . . . . . . . . 48
3.9 99th percentile latency for default parameters without Istio. . . . . 48
3.10 Median latency for default parameters with Istio. . . . . . . . . . . 49
3.11 99th percentile latency for default parameters with Istio. . . . . . . 49

4.1 Correlation between services' response times . . . . . . . . . . . . . 53
4.2 Containers' vCPU utilization . . . . . . . . . . . . . . . . . . . . . . . 53
4.3 Parameters configuration over time. . . . . . . . . . . . . . . . . . . . 54
4.4 I/O operation metrics over time. . . . . . . . . . . . . . . . . . . . . . 55
4.5 Response time percentiles over time. . . . . . . . . . . . . . . . . . . 56
4.6 Gain % response time percentiles over time. . . . . . . . . . . . . . . 56
4.7 Correlation between GOMAXPROCS, OS threads and goroutines. . 57
4.8 Number of goroutine and service throughput. . . . . . . . . . . . . . 57

4.9 Number of goroutine and response time. . . . . . . . . . . . . . . . . 58
4.10 GOMAXPROCS and memory structures. . . . . . . . . . . . . . . . . . 58
4.11 GOMAXPROCS and CPU utilization. . . . . . . . . . . . . . . . . . . 59
4.12 GOGC and GC pause time minimum value. . . . . . . . . . . . . . . 59
4.13 GOGC and GC pause time at different percentiles. . . . . . . . . . 60
4.14 GOGC and GC pause time correlation. . . . . . . . . . . . . . . . . . 61
4.15 GOGC and number of GC cycles. . . . . . . . . . . . . . . . . . . . . 61
4.16 GOGC and memory utilization. . . . . . . . . . . . . . . . . . . . . . 61
4.17 GOGC and CPU utilization. . . . . . . . . . . . . . . . . . . . . . . . 62
4.18 GOGC and response time. . . . . . . . . . . . . . . . . . . . . . . . . 62
4.19 Throttling and response time. . . . . . . . . . . . . . . . . . . . . . . 63
4.20 CPU utilization and throttling. . . . . . . . . . . . . . . . . . . . . . 64
4.21 Cluster CPU utilization during experiments. . . . . . . . . . . . . . 65
4.22 Cluster memory utilization during experiments. . . . . . . . . . . . 65
4.23 Estimate of AWS Fargate costs during experiments. . . . . . . . . . 66
4.24 Summary of best experiments. . . . . . . . . . . . . . . . . . . . . . . 67

5.1 Summary of best experiments limiting frontend GOGC <10000. . . 70

# List of Tables

2.1 Objext colour assigned by GC. . . . . . . . . . . . . . . . . . . . . . . 16

3.1 AWS EC2 instances chosen for the experiments. . . . . . . . . . . . 38

# Chapter 1

# Introduction

Nowadays, the massive spread of computer systems in the industry yielded the efficiency of those systems to be one of the essential requirements for having success in the business world: if you offer an online service, your customers expect to see results quickly and at any time they please. This has motivated the need for Performance Engineering, a field of study focused on ensuring that non-functional requirements of computing systems are met. Examples of non-functional requirements are the average response time of a system, its memory usage, or its average throughput, i.e., the number of operations per second that the system can carry out.
IT system components, both hardware and software, have some configurable properties called parameters. The set of values assigned to the parameters is called configuration.

**Figure 1.1:** Different conditions needs different settings.

The perfect example to imagine these things applied to a common-known concept is the motorsports world in Figure 1.1. A racing car has a comparable number of knobs, belonging to all its components: there are engine related knobs, brakes, aerodynamical related coefficients and so on. These parameters are tuned during the days before the race in order to obtain a better lap time and a consistent performance in order to win the race. Another important aspect is the fact that all these parameters are strictly dependant on temperatures and conditions of the track. An optimal configuration that works on a 30 degree Celsius track under the direct sun is not the one that gives you the victory on a full wet track. For sure you will have to change the tyres towards more specific ones for rain conditions. Then there are thousands of adjustments to be made in order to make the pilot comfortable during the lap.

It can be shown that the system performance is strongly affected by the configuration applied to the system. Thus, the performance engineer's work is to understand the correlation between parameters and the system performance to find the configuration which best meets a set of predefined requirements.
Modern web services architectures are made of several smaller services one on top of the other: the entire stack on its own provides a complete service, for example, the exposure of a public website of the company, while every singular part of it contributes with smaller services. It is, therefore, necessary to optimize every single service in order to have the greatest benefit for the system as a whole. And due to the diversity of workloads that modern systems have to handle there is not an overall optimal configuration that can always be used.
From intuition, you can imagine that the configuration applied to Amazon services on a normal day and the one applied during the Black Friday, when there is a huge peak in the number of connections, is not the same one. Instead, all the mentioned parameters have to be adjusted based on the type of workload and the intensity of the load that the application is receiving. Due to these difficulties, the manual tuning of all the parameters of a system is become unfeasible.
Indeed, the manual process that performance engineer experts follow can be summarized in the following steps :

∗ Identification of the conditions that the system is experiencing in a real scenario;

∗ Replicate the system and conditions it's subject to in a controlled environment;

∗ With a trial and error methods, guided by the experience and past knowledge, configurations are tested until the optimal configuration is found;

∗ The optimal configuration found on the replicated system is applied to the real one.

These steps are time-consuming and require weeks to complete, even for professionals, and the interactions among software applications in enterprise systems have become highly complex. They are usually organized in a stack architecture where different software modules are one on top of the other and interact with each other. Among these modules there are, for instance, virtual machines, containers like the one provided by Docker[1], or more high-level modules like a Java Virtual Machine (JVM). Their interaction makes the correlation among parameters be usually nonlinear and unpredictable, and it is often tough, even for a performance expert, to understand which is the best configuration.
New versions of modern software modules are now released more quickly, and sometimes they differ substantially from one version to another. Often, when we manage to optimize a system, a new version of the software has already come out,

[1] *Docker*. URL: https://www.docker.com.

and the system should be optimized again. This causes the necessity to automate the system performance optimization to carry it out fast and be up to date with the new software releases.
Many researchers have proposed the use of Artificial Intelligence (AI) to overcome these challenges.
AI studies ways to build intelligent programs and machines that can creatively solve problems, in a way similar to how human beings do. In particular, the set of most applied techniques comes from the Machine Learning (ML) framework, a subset of AI, which provides systems the ability to learn and improve from experience without being explicitly programmed automatically.
ML algorithms are mathematical models that are typically built using past data about the system and then can be used to make predictions on the modeled system. The great advantage of ML models over other techniques is that they are good in nonlinear problems with many variables, which is the case of performance optimization.

Before instructing an AI, however, it is important to have an understanding of the domain of the problem you want to solve. Therefore, a study of the technology stack you want to optimize, an understanding of the various parameters that can be configured and how they actually affect the system under analysis is necessary. The purpose of this work is to provide a starting point for studying the Go[2] language, how its runtime works, and how the available configuration parameters affect system execution. Moreover, the Go language has built-in facilities, as well as library support, for writing concurrent programs that helps the development of microservices systems, so we chose to deploy the system under analysis in an environment orchestrated by Kubernetes[3], the *de facto* standard for the management of containerized microservices systems.

For this purpose, we chose to use as system in analysis the DeathStarBench[4] [11] suite, developed at Cornell Univeristy, and specifically the Hotel Reservation application. It's composed of several services written in Go, a caching system based on Memcached[5] and multiple MongoDB[6] services.

The rest of the work is organized as follows:

* The **Background** Chapter covers all the related work needed to have some knowledge about the literature and the documentation of the tools used. Here we better explain what performance optimization is, the phases of a performance tuning test, and the concept of Go, Kubernetes, system virtualization, and service meshes. In this chapter, we also explain what technology stack parameters are configurable, namely:
  - Kubernetes: CPU request and limit, memory request and limit;
  - Go: GOGC and GOMAXPROCS.

  In particular, we expect Go's most influential parameter to be GOGC, so

[2] *Go*. URL: https://go.dev/.
[3] *Kubernetes*. URL: https://kubernetes.io/it/.
[4] *DeathStarBench*. URL: https://github.com/delimitrou/DeathStarBench.
[5] *Memcached*. URL: https://memcached.org/.
[6] *MongoDB*. URL: https://www.mongodb.com/.

analysis of the results focuses mainly on how it affects system performance and resource usage. We also investigated GOMAXPROCS, but with less interest, as Go's documentation recommends not changing this parameter, so we do not expect much change in the positive direction. However, seeing if we encounter what is already expected is interesting.
As for Kubernetes, it is interesting to see how the parameters affect container scheduling and how limiting resources affect the final results. Therefore, it is also interesting to see how changing the GOGC parameter and the memory.limits parameter change the system's operation, both having solid repercussions on the operation of the system by changing utilization and memory assigned to a container, as an Out Of Memory error results in the termination of a container.

∗ The **Benchmark application and dataset** Chapter describes the system in use: all the services that compose the DeathStarBench Hotel Reservation benchmark and the changes put in place, the choice of the load generator, the AWS infrastructure used for hosting the system, and all the configurable parameters in an environment like this and the coverage of all the possible values. Finally, we explain why Akamas has been so helpful, and the pipeline we configure to run the experiments and retrieve the metrics.
In this chapter, we briefly explain the choice of this benchmark over others. This choice was, however, very limited by the constraint that the application services have to be written in Go. The Hotel Reservation benchmark application's simplicity was rewarded. The choice of the load generator was dictated by the time needed to implement a solution that could be experimented with in the shortest possible time, as the execution of all experiments has been very time-consuming (this is due to the vastness of the parameters' domain and all possible configurations). Instead, the choice of infrastructure was dictated by the search for a good compromise between performance and cost.

∗ The **Results** Chapter explains the analysis carried out on the dataset and the most significant results obtained. We start from the preliminary analysis that gives us some insight into configuring the load generator, the difficulties we have encountered, the correlation between some parameters and memory consumption and CPU utilization, and how the change in some parameters affects the system behaviour. For example, we already know that there is a parameter in Go that affect memory consumption (i.e. GOGC affects the garbage collector cycles, so it is a direct implication), but also the parameter that influences the number of system threads in use affects memory.
For example, we figured out that some of our initial hypotheses about how the GOGC parameter affects memory consumption were correct. However, we also expected a marked correlation with response times, which we did not get. For the GOMAXPROCS parameter, on the other hand, we verified how the indications given in the documentation were well-founded. In addition, we noted how the Kubernetes parameters set an upper limit on performance. Finally, we try to figure out how the change in the value of some parameters would affect the cost on AWS, looking at how, during all the experiments,

resource consumption change and calculating the correspondent cost.

∗ The **Conclusion** Chapter summarizes the most important results, describes what could have been done differently, and, from these summaries, give some hints on how this study could continue.

# Chapter 2

# Background

In this chapter, we present the background necessary to understand our work.
We can define two main sections in this chapter. The first gives an overview of the state of the art in the field of automatic performance optimization with an introduction to some concepts of performance optimization, and the second one gives the theoretical fundamentals to understand some concepts of Go runtime and Kubernetes, the two main components of the system that we want to configure.

## 2.1 Performance Optimization

Modern IT systems' hardware and software components have different features, called parameters, which can be tuned to make the system work more efficiently or use fewer resources. A set of parameters with their values is called a configuration.

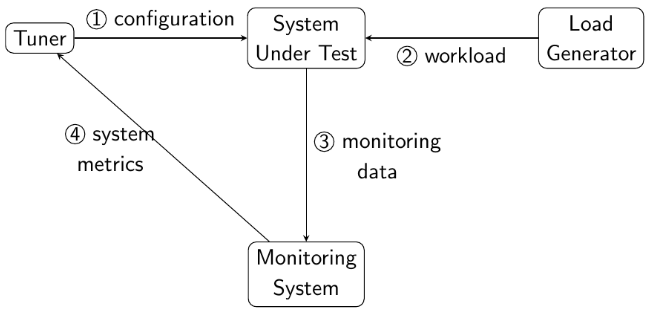

**Figure 2.1:** Performance test process.

Given some configurations, we need a method to evaluate the system performance under a specific configuration. This way, we can compare configurations and choose the one that better approximates our goal. In performance engineering, a configuration is evaluated by collecting measurable properties of the system, called performance metrics, and analyzing them.
To test a configuration, we run what is called a performance test. A performance test is a process like the one depicted in Figure 2.1. We start ① by applying a specific configuration to the System Under Test (SUT). Then, the test begins. During the test, a load generator creates ② requests for the SUT. The type and the number of requests define the workload applied to the system. At the same

time, a monitoring system monitors ③ the SUT to collect performance metrics (also referred to as KPI - Key Performance Indicator). The test lasts minutes or even hours, depending on the application under test and on what we are testing.
When the test finishes, the tuner gathers ④ the collected performance metrics and analyzes them to understand how the configuration has performed.
In performance system optimization, the process described above is repeated until the best configuration (or at least a good one) has been found. The tuner can either be a man or a machine. In the second case, the optimization process is carried out automatically. Moreover, the workload can be simulated by a load generator in the case of an offline performance test or can be a real workload if the performance test is performed on a real production system.
In the rest of the section, we describe a performance test's core elements and highlight the advantages and disadvantages of working in offline or online settings.

### 2.1.1 Parameters

As already pointed out, parameters are things we can tune to improve the system performance and can be related to hardware or software components. For example, the disk size or the number of processors an application is allowed to use are hardware parameters; the maximum heap an application can use or the number of threads instantiated for the application are examples of software parameters.
Parameters can be classified in the following two classes depending on the values we can assign to them:

∗ **Continuous parameters:** a parameter is continuous if it can assume real values in a specific interval.

∗ **Discrete parameters:** a parameter is discrete if it can assume only discrete values. They can be of any kind (e.g., reals, integers, or strings).

Modern computing systems are composed of many interconnected modules at different levels of the IT stack, and system parameters are usually correlated. This correlation is challenging from the performance point of view since we must select a set of parameters and optimize them all together to capture the correlation among them.

### 2.1.2 Performance Metrics

There is no such thing as a general formula to compute the performance of a system; instead, we need to rely on performance metrics — also called system metrics — to compare different configurations and find the best one for our purposes. The system metrics considered depends on the goal we want to achieve. For example, the goal could be to minimize the system response time or maximize the system throughput.
Usually, the optimization goal is expressed as the maximization or the minimization of one or more system metrics combined with any mathematical operation. The goal can also be subject to constraints on other resources. Consider, for example, the following goal applied to a database system: "Maximize the system throughput

over 15 minutes while maintaining the system memory used under 2 gigabytes".
In the proposed goal, we expect that the throughput of the database can increase without consuming too much cache by varying some parameters. Throughput and memory used are two examples of performance metrics, but we can have many others, such as availability, response time, or bandwidth.
Thus, depending on our goal, we focus on different metrics. It is essential to choose metrics carefully because a poor choice of system metrics can lead to erroneous or misleading conclusions [55].

### 2.1.3 Workload

Until now, we have considered only the input, parameters we can control, and the outputs, system metrics we can observe, of the performance optimization problem. We still need to introduce a third element, which can substantially impact the system performance. This element is the workload, which evaluates the type and number of requests processed by the system in a specific period.
Two properties characterize the workload of a system: the workload mix, which is the type of operations that the system has to do (e.g., read or write operations), and the workload intensity, which is the amount of work the system has to carry out.
For instance, the number of users that are using an app is an example of workload intensity; the percentage of reads and writes operations that a specific software performs is, instead, an example of workload mix.
We cannot control the workload because it depends on external factors, such as the number of users using the system or the type of applications running on it. We can only try to characterize it based on some performance metrics we collect. Workload characterization has been widely studied: a good survey can be found in [57]. In this thesis, we do not consider this topic; instead, we assume that we know the workload in advance.

### 2.1.4 Offline and Online Performance Testing

We can run a performance test in two settings: offline and online. Each of them has some advantages and disadvantages that we will describe.
In offline settings, we run performance tests on a copy of the real system, the test environment. We do not have a real workload in this environment, so we have to simulate it. Workload simulation is usually done using a well-known technique called load testing. A load test is a process that simulates a specific workload on a target machine. To be more precise, an offline performance test is the execution of a workload, a replica of a real-world system load, with a specific configuration on a copy of the target machine.
Another advantage of working offline is that we have total control over the workload injected into the system, so we have a stable environment with very little noise, perfect for analysis. Still, there are at least two disadvantages. First, it is hard to simulate precisely the real-world workload, even because, in real environments, it usually changes over time; secondly, it can be costly to replicate with enough precision the system in a test environment.

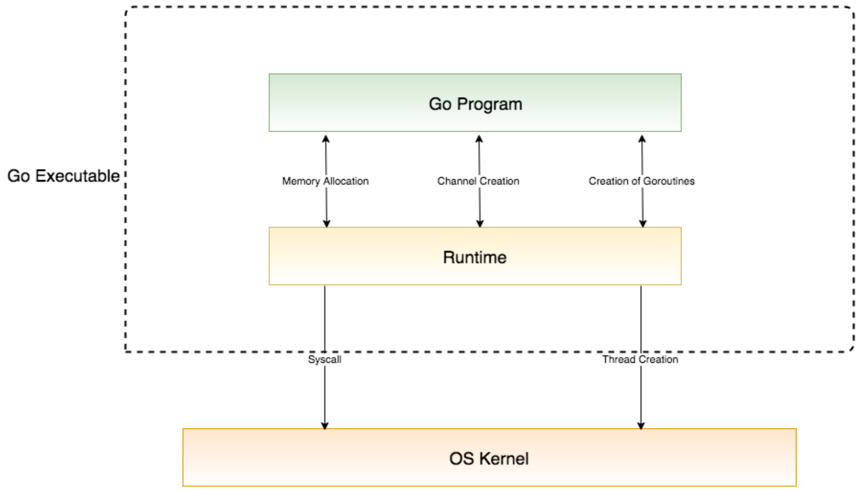

**Figure 2.2:** Go runtime and OS kernel interaction [33].

On the other hand, working online solves these issues because the test is directly run on a real system with a real workload. Unfortunately, though, it brings new challenges as usual in engineering problems. Indeed, there is much more noise in real environments, the workload varies more, and the optimizer should be smart enough to understand the changes. Moreover, we have to be sure that the configuration we are testing does not make the system crash; thus, the optimizer must learn safely in an uncertain and dynamic environment.

## 2.2 Go language

In this section, we present some concepts about Go language that help better understand how its runtime works: it manages scheduling, garbage collection, and the runtime environment for goroutines. In the following sections, we emphasize concepts inherent goroutine and how memory management works under the hood.

### 2.2.1 Go scheduler and goroutine

Go scheduler is responsible for distributing jobs in a multiprocessing environment, so when the available resources are limited, it is the scheduler's task to manage the work that needs to be done in the most efficient way. In Go, the scheduler is responsible for scheduling goroutines by mapping them onto operating system threads. Goroutines are a lightweight version of threads with a meagre start-up cost. Each goroutine is described by a struct called $G$, which contains fields necessary to keep track of its stack and current status (so $G = goroutine$).
Runtime keeps track of each $G$ and maps them onto *Logical Processors*, named $P$. $P$ can be seen as an abstract resource or a context, which needs to be acquired so that OS thread (called $M$, or Machine) can execute $G$.
Go compiler inserts calls into Go runtime in various places (e.g., when it sends a value through a channel - a mechanism that allows goroutines to synchronize without explicit locks or condition variables - or making a call to runtime package),

so that Go can notify its runtime scheduler and execute.
In Figure 2.2 is illustrated breafly the interaction between a Go program, the Go runtime and the OS kernel.

After the definition of what $G$, $M$ and $P$ are, we can describe how Go manage all of them. There are two types of queues for $G$: a global queue in the `schedt` struct (rarely used) and each $P$ maintains a queue of runnable $G$'s.
In order to execute a goroutine, $M$ needs to hold the context of $P$. The machine then pops its $P$ goroutines queue and executes the associated relative code.
When a new goroutine is scheduled (`go` function is called) it is placed into $P$'s queue. When $M$ finishes executing some $G$, then it tries to take another $G$ out of a $P$ queue, and if it is empty, it randomly chooses another $P$ and tries to take another $G$'s from its queue.
When a goroutine makes a blocking syscall, it will be intercepted, and if there are $G$s to run, runtime will detach the thread from the $P$ and create a new OS thread (if an idle thread does not exist) to service that processor (this concept is essential for understanding some characteristics of the GOMAXPROCS parameter that we will see later). When a system calls resumes, the goroutine is placed back into a local run queue, and the thread will park itself (meaning the thread will not be running) and insert itself in the list of idle threads.
If a goroutine makes a network call, Go runtime will do a similar action: the call will be intercepted, but because Go has an integrated network poller, which has its thread, it will be assigned to it.
Essentially Go runtime will run a different goroutine if the current goroutine is blocked on:

- blocking syscall (for example opening a file);
- network input;
- primitives in the sync package;
- channel operations.

In Figure 2.3 is illustrated breafly how a `go` function is scheduled and orchestrated in Go.

### 2.2.2 Main differences between goroutines and OS threads

The differences between OS threads and goroutines can be summarized as these three factors: memory consumption, setup and teardown costs, and switching time.

#### Memory consumption

Creating goroutines requires much less memory than threads, only 2kB of stack space, and they grow by allocating and freeing heap storage as required[1][2]. Threads

[1] *Effective Go: goroutines.* URL: https://go.dev/doc/effective_go#goroutines.
[2] *Goroutine stack memory consumption reduction.* URL: https://go.dev/doc/go1.4#runtime.

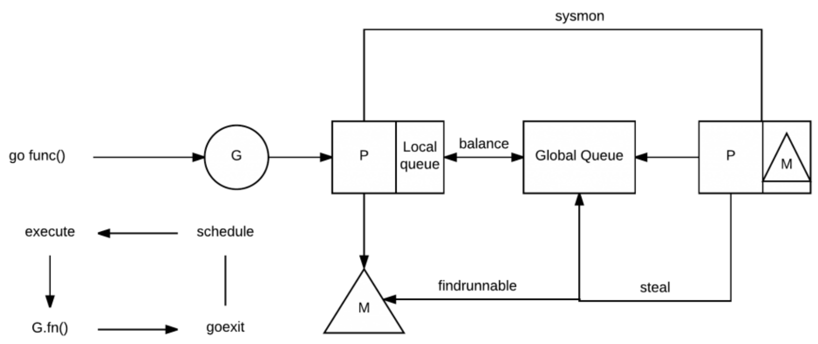

**Figure 2.3:** Go workflow graph [34].

instead usually require 1MB (500 times more as compared to goroutines), along with a region of memory called a *guard page* that acts as a guard between one thread's memory and another[3]. Goroutines are designed so that their stack size can grow and shrink according to the need of an application.
There might be only one thread in the program with thousands of goroutines. For example, a server handling incoming requests can create one goroutine per request without a problem, but one thread per request will eventually lead to *OutOfMemory* error.

#### Setup and teardown costs

Threads have significant setup and teardown costs because they have to request resources from the OS and return them once done. The workaround to this problem is to maintain a pool of threads. In contrast, the runtime creates and destroys goroutines, and those operations are cheap.

#### Switching costs

This difference is mainly because of the difference in the scheduling of goroutines and threads.
When a thread blocks, another has to be scheduled in its place. Threads are scheduled preemptively, and during a thread switch, the scheduler needs to save/restore all registers. This is quite significant when there is rapid switching between threads. Goroutines are scheduled cooperatively, and when a switch occurs, only three registers need to be saved/restored - Program Counter, Stack Pointer, and DX, so the switching cost is much lower.

### 2.2.3 Types of garbage collectors

In this Section, we summarize the different types of garbage collectors and then focus on how Go's garbage collector works.

[3] *Thread guard page*. URL: `https://dave.cheney.net/2014/06/07/five-things-that-make-go-fast`.

### Cleanup At The End: no GC

One possible way to clean up garbage is to wait until a task is finished and dispose of it all at once. This is a valuable technique, especially if you can break the task down into chunks.
The Apache Web server, for example, creates a small memory pool for each request and throws away the entire pool when the request is finished.

### Reference Counting Collector

Another simple solution is to count how many times a resource (an object in memory, in this case) is used and to delete it when the count drops to zero. A valuable property of reference counting is detecting garbage as soon as possible.
Unfortunately, reference counting has many problems. The worst is that it cannot handle cyclic structures. These are very common: anything with a parent or inverse reference creates a loop that leaks memory. The frequent updates it involves are a source of inefficiency. Also, less importantly, reference counting requires every memory-managed object to reserve space for a reference count. In addition, if the memory is allocated from a free list, reference counting suffers from poor locality and can not move objects to improve cache performance. Naive implementations of reference counting do not generally provide real-time behaviour or some guarantee on response time because any pointer assignment can potentially cause a number of objects bounded only by total allocated memory size to be recursively freed, and in the meantime, a thread cannot perform other work. It is possible to avoid this issue by delegating the freeing of unreferenced objects to other threads at the cost of extra overhead.

### Mark-Sweep Collector

Mark-sweep eliminates some of the problems of reference counting. It can easily handle cyclic structures and has a lower overhead since it does not have to maintain counts. It gives up the ability to detect garbage immediately.
The first stage is the mark stage which does a tree traversal of the entire *root set* and marks each object that is pointed to by a root as being *in-use*. All objects that those objects point to, and so on, are marked as well, so that every object that is reachable via the root set is marked. In the second stage, the sweep stage, all memory is scanned from start to finish, examining all free or used blocks; those not marked as being *in-use* are not reachable by any roots, and their memory is freed. For objects which were marked in-use, the in-use flag is cleared, preparing for the next cycle. This method has several disadvantages, the most notable being that the entire system must be suspended during collection; no mutation of the working set can be allowed. This can cause programs to freeze periodically (and generally unpredictably), making some real-time and time-critical applications impossible. In addition, the entire working memory must be examined, much of it twice, potentially causing problems in paged memory systems.

#### Mark-Compact Collector

In the algorithms described above, objects never move. Once an object is allocated in memory, it stays in the same place.
Mark-compact disposes of memory by not just marking it as free but by moving objects to the free space. Objects always remain in the same memory order, but the empty spaces caused by object disposal are filled by objects moving down.
The idea of moving objects means creating new objects at the end of the memory used is always possible. This is called a *bump* allocator and is as cheap as stack allocation but without the limitations of stack size.
Another advantage is that when objects are compacted in this way, programs have better memory access patterns, favorable to modern hardware memory caches.

#### Copying Collector

It is a moving collector like mark-compact, but simpler. It uses two memory spaces and copies live objects back and forth between them. In practice, there are more than two spaces, and the spaces are used for different generations of objects. New objects are created in one space, are copied to another space if they survive, and finally are copied to a holding space if they are long-lived.
Generational and ephemeral garbage collectors are usually multi-space copy collectors.

### 2.2.4 Go memory management and garbage collector

Go allocator is similar to TCMalloc (Thread-Caching Malloc)[4], it works in runs of pages (spans objects), uses thread-local cache, and divides allocations based on size.
The usage of TCMalloc instead of standard malloc is due essentially to the speed and memory usage difference between the two:

∗ **Speed**: glibc's malloc takes 300 nanoseconds to execute a malloc/free on a 2.8GHz processor, whereas the TCMalloc took a meagre 50 nanoseconds for the same set of operations [74];

∗ **Memory usage**: TCMalloc also reduces lock contention for multi-threaded programs. For small objects, there is virtually zero contention. For large objects, TCMalloc tries to use fine grained and efficient spinlocks. ptmalloc2 also reduces lock contention by using per-thread arenas, but there is a big problem with ptmalloc2's use of per-thread arenas. In ptmalloc2, memory can never move from one arena to another. This can lead to huge amounts of wasted space.

In Go, memory allocation has diverged quite a bit from TCMalloc, but the notion is kept essentially the same. Memory allocation work is organized in spans, which are contiguous regions of memory of 8kB in size, and allocations can also be done in multiple span sizes. There are three types of spans:

[4] *TCMalloc : Thread-Caching Malloc.* URL: `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

* **idle**: span that has no objects and can be released back to the OS, or reused for heap allocation, or reused for stack memory;
* **in use**: span that has at least one heap object and may have space for more;
* **stack**: span which is used for goroutine stack; this span can live either in stack or in heap, but not in both.

When allocation happens, Go maps objects into 3 size classes:

* **Tiny** class for objects smaller than 16 bytes;
* **Small** class for objects up to 32 kB;
* **Large** class for other objects.

Any free page of memory can be split up into a set of objects of one size class, which is then managed using a free bitmap.

### Garbage collector

Go uses a concurrent mark and sweep garbage collector.
It is called concurrent because it can safely run in parallel with the main program, so it doesn't need to halt the execution to run a garbage collection cycle (or, at least, it halts it only for a tiny part of the whole GC cycle).
The garbage collection process can be summarized in these two major phases:

* **Mark phase**: identify and mark the objects that are no longer needed by the program;
* **Sweep phase**: for every object marked as *unreachable* by the mark phase, free up the memory to be used elsewhere.

As we said previously, there is a period during the GC cycle (in the collection phase) when the program is halted, called *Stop the World*. During this time, the garbage collector temporarily halts everything but itself in order to modify the state safely. We generally prefer to minimize *Stop the World* phases because they slow our programs down. Some garbage collectors stop the world the entire time garbage collection is running, and these are "non-concurrent" garbage collectors. Instead, as we already said, Go's garbage collector is largely concurrent, and these *Stop the World* events happen in two places:

* **Before mark phase**: set up state and turn on the write barrier. The write barrier ensures that new writes are correctly tracked when GC is running (so that they are not accidentally freed or kept around);
* **After mark phase**: clean up mark state and turn off the write barrier.

Go's GC is based on the tricolor algorithm by Dijsktra [24]. The GC views heap as a connected graph of objects. The objects are either white, grey or black - the three colors of the tricolor algorithm as described in the Table 2.1.

| Color | Definition | Garbage Collected? |
|---|---|---|
| White | Set of objects that are going to be garbage collected | Yes |
| Black | Objects which are i) reachable from the root, and ii) having no outgoing references to any white object | No |
| Grey | Objects which are i) reachable from the root, and ii) yet to be scanned to check for references to white objects | No |

**Table 2.1:** Objext colour assigned by GC.

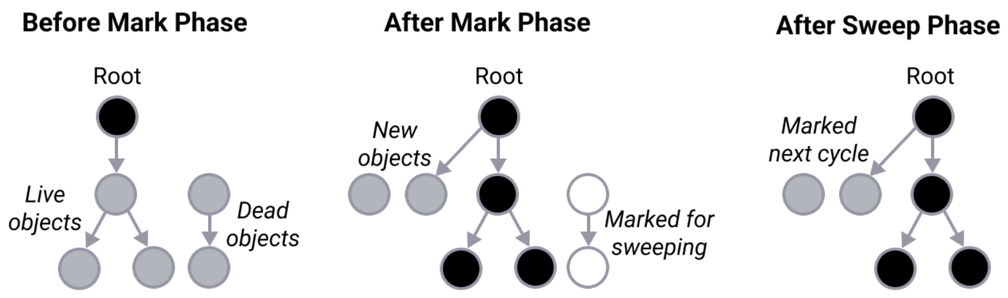

**Figure 2.4:** GC phases: black objects are still in use, white objects are ready to be cleaned up and gray objects still need to be categorized as either black or white.

At the start of a garbage collection cycle, all of the objects are marked white. Then, the GC collector scans for accessible items, starting by inspecting the stacks for all existing goroutines to find root pointers to heap memory. Then the collector must traverse the heap memory graph from those root pointers, and marks them grey. If the GC finds a new reachable white object during the second scan, it turns the white object into grey. Finally, the GC chooses a grey object, blackens it, and then scans it for pointers to other objects[5]. This is done till the there are no more grey objects. At this point, unreachable white objects and their corresponding memory may be reused for other purposes.
The GC and the running program maintain the invariant that black objects in the heap never hold a reference to white objects. Furthermore, the state of the color of an object changes from white to grey and grey to black.
Figure 2.4 illustrates the different colours of objects during a GC cycle.
During the marking phase, Go has to be sure it will mark the memory faster than it will make new allocations. Indeed, if the collector is marking some memory while, for the same period of time, the program is allocating the same amount of memory, the garbage collector would have to trigger as soon as it is finished.

[5] *Tracing Garbage Collection*. URL: https://en.wikipedia.org/wiki/Tracing_garbage_collection.

In order to address this issue, Go tracks the new allocations while marking the memory and watch when the garbage collector is in debt. The first step starts when the garbage collector is triggered. It will first prepare one goroutine per processor that will sleep, waiting for the marking phase.
Once those goroutines spawned, the garbage collector will start the marking phase that will check which variable should be collected and swept. The goroutines marked *GC dedicated* (usually 25% of the CPU capacity) will run marks without preemption while the ones marked as *GC idle* are running the program concurrently. The ones marked as *GC idle* can be preempted. In that case, they have to stop program execution and participate in the GC phase, so there is the possibility that the program execution is slowed down further.
The garbage collector is now ready to mark the variable not in-use anymore. For each variable scanned, it will increase a counter in order to keep track of the current work and be able to get the picture of the remaining work as well. When a goroutine is scheduled for work during the garbage collection, Go will compare the required allocation to scanning done already in order to compare the pace of the scanning and the requirement in allocation. If the comparison is favorable for the scanning, the GC goroutine does not need help. On the other hand, if the scanning is in debt compared to allocation, Go will use the other goroutines that are running the main program for assistance.
Go's garbage collector uses a *pacer*[6] algorithm to estimate the optimal times for triggering a garbage collection cycle. The algorithm depends on a feedback loop that the collector uses to gather information about the running application and the stress the application is putting on the heap. Stress can be defined as how fast the application is allocating heap memory within a given amount of time. So stress determines the pace at which the collector needs to run. To make the decision and trigger the next GC cycle, at every run the pacer updates its internal goal for when it should run GC next. Since one goal of the collector is to eliminate the need for mark assist from other goroutines, if any given collection ends up requiring a lot of mark assist, the pacer can decide to trigger the start of the next garbage collection earlier. This is done in an attempt to reduce the amount of mark assist that will be necessary on the next collection. There is also a parameter, GOGC, that can be tuned and has a major importance on the triggering ratio of the pacer. We will come back later on this since it is one of the goal of this work.

## 2.3 System Virtualization

Virtualization, one of the key concepts of cloud computing, which has been well established for decades [69], refers to the creation of a virtual version of a resource, such as the operating system, hardware, storage or network, in an abstract layer from the real one. These emulated virtualized systems can be configured, maintained and replicated more efficiently and on demand [71]. Furthermore, thanks to virtualization, the IT infrastructure is assigned to users and applications according

[6] *GC Pacer Redesign.* URL: `https://go.googlesource.com/proposal/+/a216b56e743c5b6b300b3ef1673ee62684b5b63b/design/44167-gc-pacer-redesign.md#gc-pacer-redesign`.

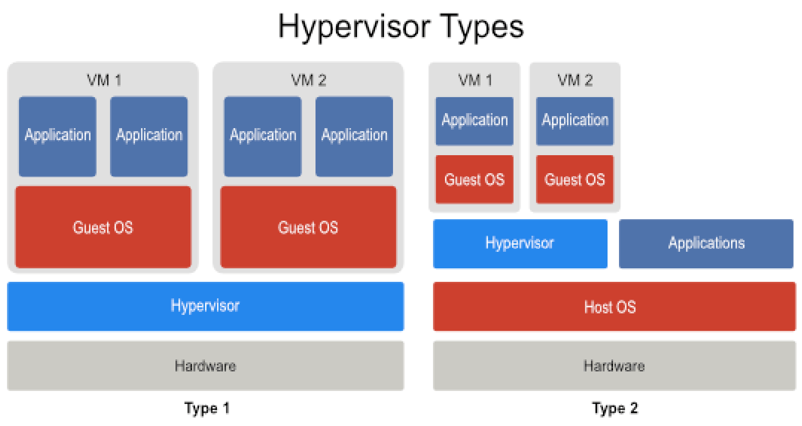

**Figure 2.5:** Types of hypervisor-based virtualization [79].

to their actual needs and therefore resources are used much better. This leads to a decrease in initial operating costs and a reduction in carbon emissions. In Linux terms, virtualization refers to running one or more virtual machines managed by the Linux operating system on a single physical computer. The two most popular Linux virtualization technologies include hypervisor-based and container-based virtualization, which are described in the following sections.

### 2.3.1 Hypervisor Virtualization

The first well-known virtualization technology that has been around for decades is the hypervisor, also called Virtual Machine Monitor (VMM). In a hypervisor, the computer hardware or software is the host and provides a complete abstraction for one or more virtual machines acting as a guest [70]. The host machine divides and allocates the resources available locally to the guest machines. In this way, each guest virtual machine will have its own operating system and will operate in isolation from the others. Therefore, multiple machines with different operating systems can run concurrently on a single physical host.

There are two different type of hypervisor, as shown in Figure 2.5:

- **Type 1 hypervisors**: called native or bare-metal hypervisors, run directly on hardware to control it and manage operating systems for guests [58]. The most well-known examples of this architecture include Oracle VM, VMWare ESX [2], Microsoft Hyper-V [3], and Xen [65];

- **Type 2 hypervisors**: called embedded or hosted hypervisors, on the other hand, require a host operating system to run their operations on top of it [58]. Hence, these type of hypervisors are dependent on the host OS for their resource allocation. Some popular hypervisor tools coming from this architecture are Oracle VM VirtualBox [66], VMWare Workstation [41] and QEMU [27].

Despite the differences in type, hypervisors offer several advantages. The main one is the isolation of the guest machines. Isolation ensures that the consequences of any operation within one virtual machine do not affect other virtual machines

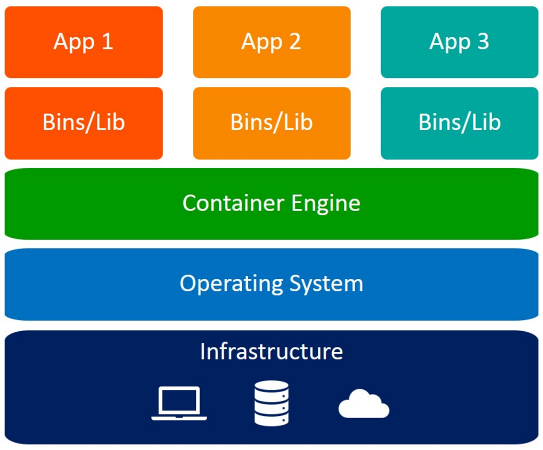

**Figure 2.6:** Container-based virtualization architecture [12].

or the host. This prevents any crashes, failures or security threats caused by one environment from disrupting the functionality of other machines. Additionally, running a hypervisor allows users to have multiple machines with different operating systems on a single host machine. In addition, this introduces various business opportunities for service providers, who will be able to meet a wide range of customer needs with limited resources.
Despite their advantages, hypervisors have several disadvantages. First, guest virtual machines' dependence on host machine startup causes slow startup time. Also, each virtual machine must boot like a regular operating system, which makes it even slower. Other disadvantages are upfront costs and possible security vulnerabilities[16].

### 2.3.2 Container Virtualization

Container-based virtualization, also known as OS-level virtualization, is a lightweight alternative to hypervisors. This type of virtualization uses features of the host kernel to create multiple isolated instances in the user space, called containers. A container, from the point of view of the processes running inside it, looks like a real separate machine, while in reality they are running in an isolated space within the host operating system and sharing their resources. So, despite the hypervisors, containers don't have their own virtualized hardware and operating system, but they use their host's resources. In this way, each container acts as an independent operating system, without any intermediate layer and without a virtualized guest operating system, as happens in hypervisors. Figure 2.6 illustrates this architecture.

Since there is no emulated hardware in containers, they don't need time to boot an entire operating system. This leads to a fast startup, in milliseconds, which is more efficient than conventional hypervisors [58]. A container encapsulates all packages it might need, such as libraries, binaries, runtimes, and other system-specific configurations[7]. However, it is still lighter than a virtual machine,

[7] *What is a container?* URL: https://www.docker.com/resources/what-container.

which contains a set of toolchains to run an entire operating system, including kernels and drivers. This small footprint of resources in containers introduces better performance, greater security and good scalability [58].
Container-based virtualization can be implemented on any operating system, but the most popular techniques today, such as Docker, are based on the characteristics of the Linux kernel.
In Linux, resource management for containers is accomplished through control groups (*cgroups*). Cgroups limit and prioritize the use of host hardware resources, such as memory, CPU, and I/O, for containers [23]. In addition, Linux namespaces provide isolation for containers, so that each process has its own specific view of the system. An example of this controlled view is the ability for a container to see parts of the host's file system and not all, and also check the list of visible processes in the container process tree.

## 2.4 Docker Container Engine

Docker, an open-source project in the Linux container category, is one of the most popular container virtualization technologies [13]. It is a lightweight platform for developing, deploying and running applications inside containers. As one of the most powerful technologies at the moment, it is even considered synonymous with containerization in some terminologies. The benefits Docker offers for deployment and operations are discussed in the following subsection. Then, in the second subsection, its architecture and the most critical components are illustrated.

### 2.4.1 Docker for Development and Operations

Docker offers a fast, automated approach to deploying an application within portable containers. In this way, the application is suitable for any environment, can be scaled and will be configured to interact with the outside world. Furthermore, the applications built with Docker are independent of the infrastructure and can run on any platform. This feature solves the *dependency hell* for developers[8] and makes deployment, shipping and testing of the application easier and faster and shortens the lifecycle for application release. This is particularly beneficial for CI/CD workflows[9].

Additionally, Docker offers the user the ability to configure infrastructure components, such as CPU, memory or network, through its configuration files. In this way, developers are able to manage infrastructure in the same way they treat applications. To organize these features, docker introduces an application-level kernel and API into the Linux container, which run processes such as CPU, I/O, and memory in isolation [13]. Also, to deploy and run containerized applications, Docker uses the two most important features of the Linux kernel, cgroups and namespaces, as described in the Section 2.3.2.

[8] *The industry-leading container runtime.* URL: https://www.docker.com/products/container-runtime.

[9] *Docker overview.* URL: https://docs.docker.com/engine/docker-overview/.

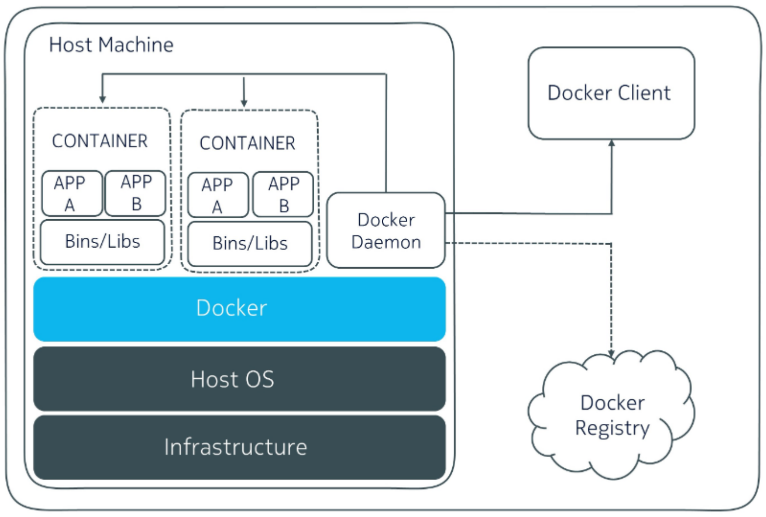

**Figure 2.7:** Docker Engine architecture [20].

Applications can run in a secure and isolated space within Docker containers. Hence, using Docker, multiple applications or microservices can run simultaneously within different containers on the same physical host. With this capability, developers can split a huge monolithic system into a number of smaller services, each of which can be deployed as a separate Docker container. This would makes debugging, managing and updating each component easier. Also, the application can be scaled horizontally with only the necessary services. Also, being lighter than traditional hypervisors, Docker can uses hardware resources more efficiently and can handle the workload dynamically.

### 2.4.2 Docker Architecture and Components

Docker uses a universal packaging approach to wrap all application dependencies within a container, which run on the Docker engine[10]. The Docker engine is based on a client-server architecture:

* **Docker Daemon**: which runs on the host system, manages images and containers. It is also responsible for monitoring system health, enforcing policies, executing client commands, and determining namespace environments for running containers;

* **Docker Client**: is where users interact with Docker.

A command line tool, which connects to the Docker daemon, is responsible for managing operations related to images, containers, networks, swarm mode and configurations of the Docker engine. Using sockets or RESTful APIs, the Docker daemon and the Docker client communicate with each other. Docker's architecture is shown in Figure 2.7.
The other components of Docker are described below:

[10] *The industry-leading container runtime.* URL: `https://www.docker.com/products/container-runtime`.

∗ **Docker Image**: A Docker image is the main building block of Docker. It provides source code to build and run containers. An image contains all the libraries and binary files needed to build and run an application. There are pre-built Docker images, which developers can download and use, but they can also create their own images. To do this, you need to write the necessary instructions in a Docker file. The image template (Docker file) starts with a base image. An image is a series of data layers built on top of the base image. To combine the different layers of an image and treat them as a single layer, Docker uses a particular filesystem called the Union File System (UnionFS) [73]. Union File System allows you to combine files and directories from different file systems into one consistent file system [73]. When you apply changes to the image, a new layer is added to the existing ones. Therefore, despite the traditional hypervisor-based approach, there is no need to rebuild the entire image. This process makes image reconstruction quite fast [9].

∗ **Docker File**: The Docker file is a template that contains all the instructions needed to create a Docker image. The Docker engine gets from this file the information it needs to configure a container or run containerized applications. These instructions are executed in order by a `Docker build` command issued by a user. In the event that one of these instructions results in a change to the current image content, the changes are be applied to a new layer based on the layering approach described above. The base image is specified with the `FROM` command written in the first line of the Docker file. This is where the whole image is built from.

∗ **Docker Container**: Docker containers are running instances created and deployed from Docker images and allocated system resources, which can be managed through the Docker client tool. Although different containers can be launched from the same image or from various images on the same host, they are all isolated from each other and act separately. When starting a container, a new writable layer, the *container layer*, is created on top of the used image[11]. Any changes that occur within the running container apply only to the container level. When different containers run on top of the same image, they write the changes into their own writable layers, and when a container stops, all changes made to its base image are lost. This way the images remain immutable and you can create different containers from a single Docker image.

∗ **Docker Registry**: The Docker registry is where Docker images can be stored. The primary purpose of a registry is to simplify the distribution of images. Therefore, developers can create Docker images and keep them in registries. Then, after accessing the registry via the Docker client, other users can download and use the available images. We can consider these registers similar to repositories of source code [14]. The Docker registry can be public or private.

[11] *About storage drivers.* URL: `https://docs.docker.com/storage/storagedriver/#images-and-layers`.

## 2.5 Container Orchestration Frameworks

With the growth of virtualization technologies, particularly with the launch of Docker, and with the growing interest of PaaS vendors, application virtualization has become popular. This popularity is due to the fact that containers, by their nature, potentially solve many known problems of application development and deployment. By encapsulating an application with all its dependencies in standalone software, which can run on any platform, they solve the *dependency hell* problem. This makes portability of the application much more manageable. Being lighter than virtual machines, they improve performance overhead and reduce boot time [44].

Operating containers at scale increases the demand for a management and orchestration tool. Multiple containers, which make up a distributed architecture, need to interact seamlessly with each other. Consequently, as the application footprint grows, the demand for an automation process increases. Dynamic deployment, automation, management, scaling and monitoring of packaged applications with multiple containers can be achieved with a container orchestration framework [22]. This framework distributes processes to different physical hosts, monitors them and tracks their health. To do this, a container management framework employs a software layer that abstracts the complexity of hosts and views the entire cluster as a single resource pool. Furthermore, containers within a cluster are able to communicate with each other regardless of the physical host on which they are deployed. For this purpose, the management framework creates virtual networks between containers. Some of the current frameworks offer even more and provide load balancing and service discovery mechanisms. Several orchestration frameworks have been developed, and Kubernetes is one of the most popular, so we have conducted our study on it, which is explored in the next section.

## 2.6 Kubernetes

Kubernetes is an open-source cluster manager, initially developed and presented by Google at the Google Developer Forum in June 2014 [22]. The origin of Kubernetes adopted many ideas from Google's first internal container management technology, called Borg [7]. Google applications were run and managed internally at scale using Borg. Later, many external developers became interested in Linux containers, and Google developed its public cloud infrastructure. These advances motivated Google to develop its open-source container management framework, known as Kubernetes. Kubernetes can be used for effective deployment, updating, management, resource sharing, monitoring and scaling of multi-container applications in a highly distributed environment.

### 2.6.1 Kubernetes Architecture

The high-level architecture of Kubernetes is shown in Figure 2.8. As illustrated, each Kubernetes cluster consists of a *master node* and one or more *worker nodes*. This structure allows Kubernetes to optimize the power of cluster computing by

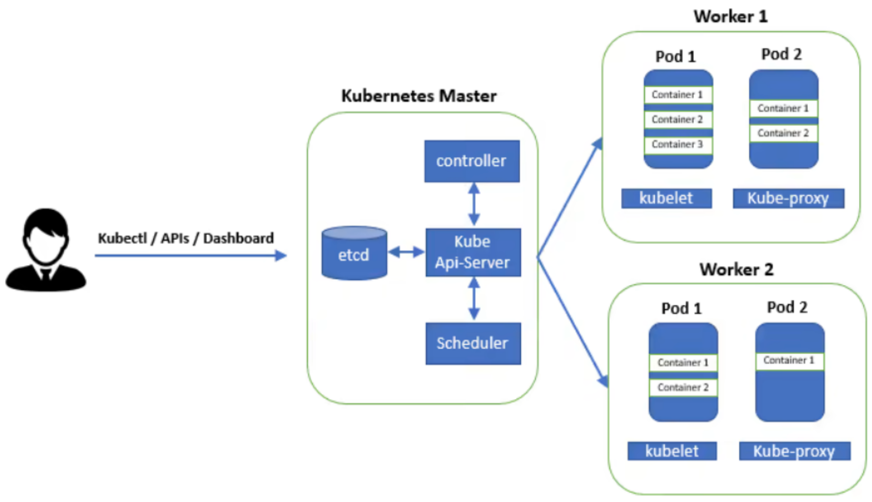

**Figure 2.8:** High-level Kubernetes architecture [37].

distributing the containers on different worker nodes, while their management is entrusted to the master node. The master node includes components such as the *API server*, *controller manager*, *scheduler*, and *etcd*. The worker node has only two components: *kubelet* and *kubeproxy*. Each of these modules is discussed in detail. The key components of Kubernetes are described below:

∗ **Master and Worker Nodes**: following a master-slave architecture, Kubernetes consists of master and worker nodes[12]. A node, in general, refers to the host device, which is a virtual or physical machine. Worker nodes run pods and are managed by the master node. The master node within a cluster is responsible for container management and consists of three processes: *kube APIserver*, *kube controller manager* and *kube scheduler*[13].

∗ **Pods**: the smallest distributable compute unit in Kubernetes is called the pod[14]. Each pod consists of one or more application containers, which are distributed on the same physical host and share the same set of resources, such as storage and networking. In other words, a pod models an application-specific *logical host* and includes several containers that are tightly coupled. Therefore, the pod ranks and operates at a higher level than individual containers[15]. Kubernetes applies its scheduling and orchestration mechanisms on top of pods instead of containers. Each pod is assigned a unique IP address. While containers from the same pod can see each other on localhost, containers from different pods communicate using the pod's IP addresses. In addition, the containers of a single pod share the same directories and volume resources. Hence, a pod resembles a virtual host machine which includes all the necessary resources for its containers.

[12] *Understanding Kubernetes Architecture.* URL: https://geekflare.com/kubernetes-architecture/.

[13] *Kubernetes concepts.* URL: https://kubernetes.io/docs/concepts/.

[14] *Kubernetes pods.* URL: https://kubernetes.io/docs/concepts/workloads/pods/pod/.

[15] *Kubernetes, viewing pods and nodes.* URL: https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/.

* **Replication Controller**: replication controller is responsible for ensuring that the specified pod number is always up and running and, if not, replicates the required pod number[16]. Also, terminate some of the pods if too many are running. Replicas are managed according to the rules defined in the pod model. A manually created pod can be deleted in the event of a failure, while using a replica controller you can define pods that are replaced in the event of a failure or deletion for any reason. Therefore, it would be a good practice to define a replication controller instead of manually creating the pods. This ensures the health of the application even if it contains only one pod.

* **ReplicaSet**: ReplicaSet is a Kubernetes API object that manages pod scaling. It checks the status of the pods and keep the number desired at any time. According to the Kubernetes documentation[17], ReplicaSets are the next generation of replication controllers and offer more functionality.

* **Deployment**: Deployment is a higher level concept than ReplicationSets and pods. By defining a deployment, we can declare updates for pod and ReplicaSet. In other words, the desired state is described and the deployment controller verifies the actual state against the desired state, so instead of using pods or ReplicaSets directly, it is recommended that you define them through deployments.

* **Services**: a service is an abstract way to expose applications running on pods to users. As mentioned, each pod within a cluster is assigned a unique IP address. However, because pods can be created or removed from the replica controller, their IP addresses are not stable. Each newly created pod receive a new IP; as a result, it would not be a good approach for users to connect to applications via pod IPs. This is where Kubernetes services solve the problem by offering an endpoint API, which allows services to be accessed externally. Furthermore, by assigning a single DNS name to each service, Kubernetes provides an internal service discovery mechanism and developers do not need to use an external approach for this[18].

### 2.6.2 Master Node Components

The master node, the controller of the Kubernetes cluster, is responsible for maintaining and managing the state record of all system objects. To do this, it runs continuous control loops to respond to changes. Therefore, the control plane is the place where users and system administrators interact with Kubernetes. It accepts client requests and makes sure to bring the current state of the cluster to the desired state, as described by users via the Pod Lifecycle Event Generator (PLEG) [48].

[16] *Kubernetes replication controller.* URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/.

[17] *Kubernetes replicaset.* URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/.

[18] *Kubernetes services.* URL: https://kubernetes.io/docs/concepts/services-networking/service/.

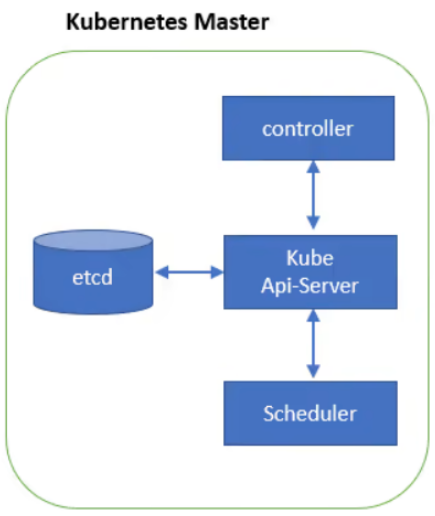

**Figure 2.9:** Kubernetes master node [37].

The Kubernetes master node adopts a set of processes that manage the state of the cluster. These components are described below:

- **API Server**: the API server exposes a REST API, which makes it possible for all cluster components to communicate. It also allows users to configure and validate all objects in the cluster, such as pods, replication controllers, services etc. In addition, the API server manages communications between the master and worker nodes.

- **Controller Manager**: the controller manager is a control loop that runs on the master and, using the API server, checks the status of the cluster. If a change occurs, the controller manager shifts the current state of the cluster to the desired state. There are several controllers in Kubernetes, including the Replica Controller, Namespace Controller, Endpoint Controller, and Service Account Controller.

- **Scheduler**: the scheduler acts as a resource controller and manages the workload of a cluster. More specifically, it is responsible for assigning pods to different worker nodes based on various metrics, such as node compute resources, pod political constraints, and quality of service requirements. For this purpose, the scheduler also keeps track of the general overview of the resources to see which are free or busy.

- **etcd**: etcd is a lightweight, consistent, highly available, and distributed key-value data store used to store all cluster data, including cluster configurations and state information. To function as a data store, etcd relies on the Raft consensus algorithm [15].

### 2.6.3 Worker Node Components

Worker nodes are the machines on which the application runs. The user usually does not interact with these nodes and the master node controls each of them. Each

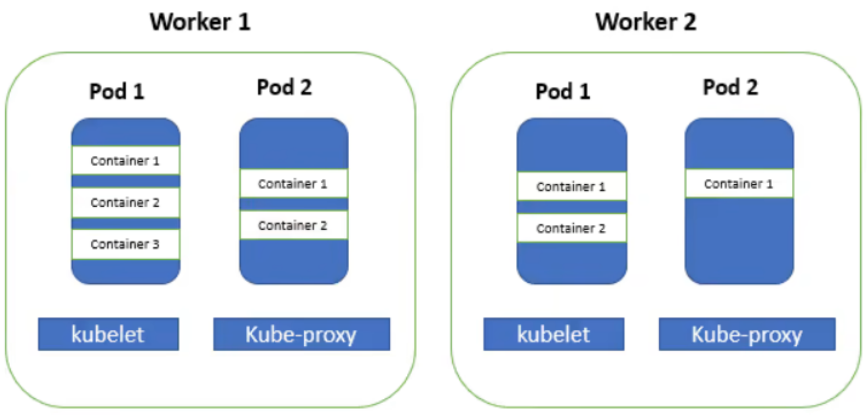

**Figure 2.10:** Kubernetes worker nodes [37].

worker node consists of a few components described below:

* **Container Runtime**: a runtime container must be installed on each worker node. The container runtime is the software responsible for running containers. Kubernetes supports several container runtimes, but Docker is the most popular.

* **Kubelet**: being responsible for managing the pods and their containers on each node, kubelet is the most critical component of worker nodes. Kubelet receives the instructions from the master node and, interacting with etcd, updates the configurations. Based on the information gained, he makes sure the pods are healthy and functioning correctly. Then, it reports the status of the nodes to the cluster.

* **Kube-Proxy**: Kube-Proxy is a network proxy that takes care of the network rules on each worker node. These rules allow network communication with pods from inside or outside the Kubernetes cluster. In other words, by mapping containers to services and using load balancing mechanisms, it provides access to the distributed application from the outside world. These network proxies work based on TCP and UDP streams.

## 2.7 Service Mesh

A service mesh can be described as a specific infrastructure layer that manages communication between the services of a containerized system via load balancing, routing, authentication, encryption, and monitoring. Service mesh takes the concept of decoupling features from the application, but, based on proxies that work alongside the main container, they are applied in a more distributed way: different controllers are associated within each service, and a global controller provides configurations updating and coordinating all the points of application. It leads to a separation of concerns between Kubernetes, which focuses on deployment and management of container workloads at scale, and the service mesh, which

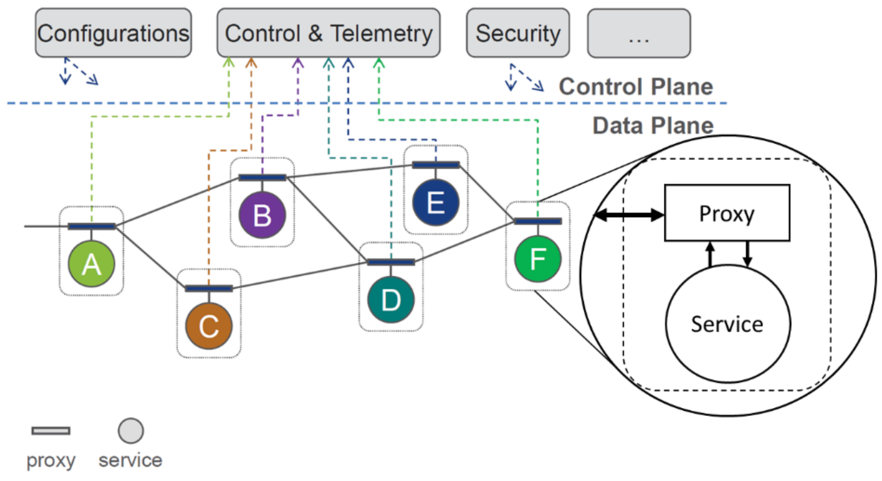

**Figure 2.11:** Service mesh architecture overview [72].

cooperates to manage the network traffic in a secure, reliable, and observable way. A service mesh conceptually has two modules: the *data plane* and the *control plane*. The data plane carries the application request traffic between service instances through service-specific proxies, called *sidecar proxy*. The control plane configures the data plane, provides a point of aggregation for telemetry, and provides APIs for modifying the behavior of the network through various features, such as load balancing, circuit breaking, or rate-limiting. Service meshes create a small proxy server instance for each service within a microservices application. The sidecar proxy forms the data plane, while the runtime operations needed for enforcing security (access control or communication-related features) are enabled by injecting policies (e.g., access control policies) into the sidecar proxy from the control plane. Traffic, therefore, does not pass through the control plane, but only between the proxies, appropriately configured. Another concept to note regarding the implementation of a service mesh is the transparency of this additional layer from the point of view of the applications. In most situations, they are not be aware of its presence and no changes need to be made in the implementation.

## 2.7.1 Istio

The open platform Istio matches precisely the service mesh model just described. Figure 2.12 shows a complete Istio architecture representation from the official documentation and the analogy with Figure 2.11 is clear. Each service interacts just with its own proxy. The latter, on the other hand, takes part of the data plane and connects to the other proxies to actually send the traffic to any destination. As the figure shows, unique proxies named *Ingress* and *Egress* are also part of the data plane providing respectively connectivity from external networks and the possibility to establish connections to APIs located outside the cluster. The Istio core element is the control plane. Through it, all the proxies are injected, managed and updated, leading to a wide list of provided functionalities that include:

∗ **Traffic management**: traffic routed between proxies can also be controlled

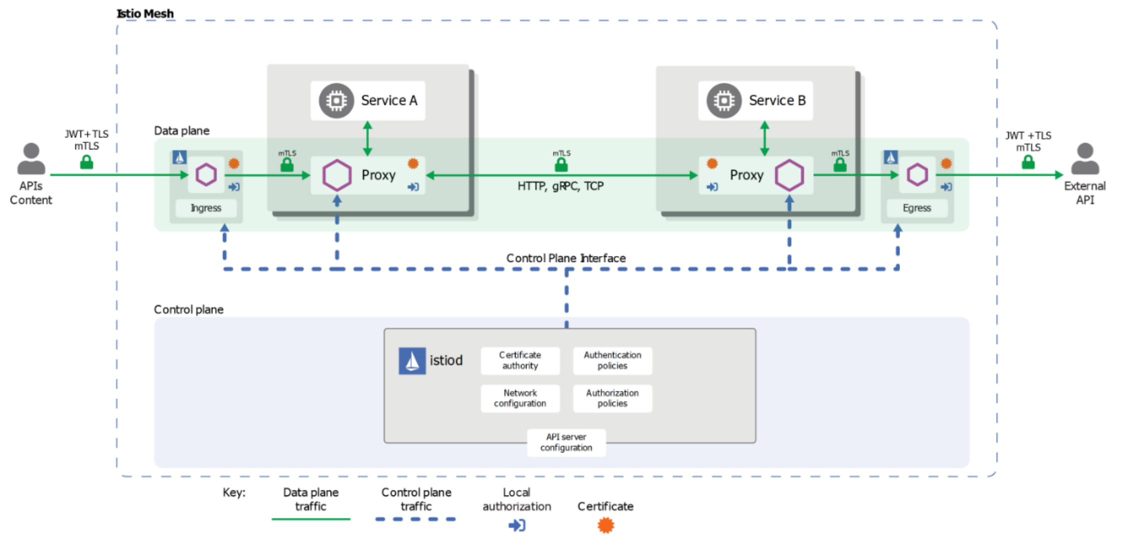

**Figure 2.12:** Istio architecture overview [40].

with:

– Traffic shaping with matching rules and weight-based traffic splits;

– Circuit breakers;

– Request timeouts;

– Retries;

– Mirrored traffic;

– Fault Injection.

∗ **Observability**: traffic across the proxies is monitored, permitting to understand the behaviour and performance of the cluster. The customizable telemetry includes detailed metrics, distributed traces and access logs;

∗ **Security capabilities**: Protection of microservices deployments are based on security-by-default functionalities and features that permit to go towards defense in depth approach like peer authentication based on mutual TLS, end-user authentication JWT-based and custom authentication and authorization policies;

∗ **Extensibility**: proxies can be extended with custom logic thanks to developed modules executed inside a sandbox. The goals of this extension technology are:

– Efficiency: low latency, CPU, and memory overhead;

– Isolation: each extension is executed isolated inside a sandbox;

– Configuration: Istio control plane handle the dynamic configuration of the extensions.

### Data plane

The solution adopted by Istio is to implement the proxies as sidecar containers by deploying *envoy* proxies inside the pod of every service. Therefore, communication never take place directly between the pods, but the traffic is routed to the sidecars and they ensure networking and any other features. The set of these proxies forms the Istio data plane. Envoy is an open-source C++ proxy designed to be implemented in a distributed way at the edge, close to single services and applications[19]. By doing so, network functionalities are outside the logic of the application and independent of the application language used. Envoy provides a wide variety of functionalities extending the core feature of being an L3/L4 network proxy:

- A pluggable filter chain mechanism to write filters or apply some of the already implemented ones for common tasks and applications;
- HTTP L7 support both in terms of filter layer to perform tasks such as rate limiting and routing, and flexibility about protocols used at L7. It is indeed capable of bridge HTTP/1.1 and HTTP/2 in any communication direction;
- gRPC support, widely used in distributed client-server architectures;
- A configuration API allowing different approaches for configuration management, from static ones to dynamic configurations;
- A health checking system to keep tracking the status of the pods. It can be active, sending requests and analyzing the response, or passive, based on outliers detection such as consecutive response errors or success rate;
- Advanced load balancing capabilities based on the availability of the hosts, specific application level parameters, the possibility of choosing between different algorithms and via global decisions defined by a control plane;
- TLS termination and TLS origination to remove security duties from the main logic and, at the same time, being able to take advantage of unencrypted traffic;
- Observability via logging and statistics collection to monitor and debug network-level problems;
- Support for tracing via external tracers like Jaeger.

### Control Plane

Envoy implements advanced features, but its scope is the single pod and, as described, pods are made to be ephemeral. The key characteristic of Istio consists in its ability to instantiate, manage, monitor and distribute configurations to all the dynamic envoy proxies distributed across the cluster. This is accomplished by *istiod*, the core element of Istio's control plane. Istiod is composed of a set of pods and services inside a specific namespace called *istio-system*. Istiod is capable of:

[19] *Evoy proxy project.* URL: `https://www.envoyproxy.io/`.

∗ Convert high-level written routing rules, typically given as YAML configuration files, into envoy configurations and then propagate them throughout the mesh to each envoy sidecar;

∗ Manage authentication and authorization policies to enable service-to-service authentication, end-user authentication and access control for each workload;

∗ Act as Certificate Authority (CA): generates, rotates and revokes certificates to provide built-in identity and TLS encryption across the mesh;

∗ Collects telemetry.

Another element that is still part of the Control Plane is the *istio-agent* (also called *pilot-agent*). One istio-agent instance is executed inside each pod. It has the function of bootstrapping envoy and permit it to connect to the service mesh providing configuration and secrets. Istio, at a high level, can be seen as an envoy proxies manager: it inherits their functionalities and centralises the management by adding configuration automation, flexibility and a set of services to take full advantage of envoy. First and foremost, the data plane must be in place to exploit the service mesh, so sidecars must be deployed. Istio itself is capable of handling it, providing a manual injection into the pod template via the Istio configuration command line (*istioctl*) but, above all, it permits to automate the injection inside each deployed pod. By adding a label (*istio-injection=enabled*) to a namespace, any pod deployment in that namespace is effortlessly customized implementing the sidecar proxy. From the point of view of the functionalities provided by adding the Istio layer to the Kubernetes cluster, it is possible to divide them into three categories: traffic management, security, and observability features. In our work, Istio has been used for the latest one since it has been exploited for monitoring response times of the entire system and the single services. This is explained in more detail in the next chapter.

# Chapter 3

# Benchmark application and dataset

In this chapter, we present the benchmark application chosen, what parameters we have tested, how we have conducted our experiment, and how we have gathered telemetry data.

We can define four main sections in this chapter. The first gives an overview of the application chosen for our test, how it is structured, what changes we made to the original code and how we generated the workload. The second specifies the characteristics of the AWS environment that hosted the cluster. The third one gives an overview of the parameters we could tune and make some assumptions on how they should change the system's behavior. Finally, the forth one specifies how we executed our experiment using Akamas.

## 3.1 Benchmark application and workload generator

For our experiment, we searched for a distributed application based on microservices that can run on Kubernetes and was written in Go. Unfortunately, the choice was limited since most benchmark applications that have already been tested extensively were written in Java. Our choice therefore focused on two possibilities:

* Online Boutique: a cloud-native microservices demo application developed by Google that consists of a 11-tier microservices application. The application is a web-based e-commerce app where users can browse items, add them to the cart, and purchase them[1];

* DeathStarBench: a suite of microservices based applications developed at Cornell University composed of three solutions: Social Network, Media Service, and Hotel Reservation. The latter is the one that gets our attention since it is entirely written in Go.[2]

[1] *Online Boutique project.* URL: https://github.com/GoogleCloudPlatform/microservices-demodeathas.

[2] *DeathStarBench project.* URL: https://github.com/delimitrou/DeathStarBench.

Our choice was the latter suite, particularly the Hotel Reservation[3] application. This is because having more services written in Go has allowed us to test the interaction of the services with each other and have a broader coverage of the parameter configurations for each experiment since Online Boutique has a multiplicity of services written in other languages.

### 3.1.1 The benchmark

#### Hotel Reservation application

The application we have chosen simulates a website that displays some hotels based on the geographical area and their relative information, such as the price per night, the availability of rooms, and the services offered, allowing you to book rooms. The backend part consists of eight services written in Go that communicate each other using gRPC[4]:

* frontend service: manages all user requests and redirects various calls to other services. It is the entry point of all requests, so we can rely on this service to check the total response time;
* geo service: manage hotel ids and their geographical coordinate location that is used for the plot on the geographic map;
* profile service: manage hotel ids and various hotels info, such as their name, their phone number, their description, and their address;
* rate service: manage the hotels' rate for a specific date range and the different types of rooms;
* recommendation service: manage the recommendation system based on given requirements such as distance from a point or the rate;
* reservation service: check if a hotel is available for the given dates and manage new reservations. This is the only service that makes POST calls and writes to the database;
* search service: manage hotel search returning hotels near the desired coordinates and get their rates;
* user service: manage users' data that are pre-registered in a database, so it checks if the user and password for some user are correct.

All the information these services use and store (remember that only the reservation service makes write-calls to the database, all the other info are pre-saved) are stored in instances of MongoDB with the help of a caching system based on Memcached. These instances are:

* mongodb-profile;

[3] *Hotel Reservation application.* URL: https://github.com/delimitrou/DeathStarBench/tree/master/hotelReservation.

[4] *gRPC.* URL: https://grpc.io/.

* mongodb-geo;
* mongodb-rate;
* mongodb-recommendation;
* mongodb-user;
* mongodb-reservation;
* memcached-reserve;
* memcached-rate;
* memcached-profile.

#### What we changed

After some experiments, we noticed that the caching system represents a problem: if in a situation of actual usage, adopting a caching system turns out to be an excellent choice to optimize the response time of some requests, thus improving the performance of the system and reducing the consumption of resources, in our case having something that over time, with more and more experiments, the performance of the benchmark improved, it was very misleading, as some metrics did not depend on the configuration adopted but on the cache-hit/cache-miss ratio of the caching services. For this reason, we decided to keep Memcached's caching services, but to perform a restart of the relevant containers at each test start, so that they would not hold data in memory that would affect program performance.
From the perspective of the services Go code, some changes were made, although the underlying logic remained unchanged. These changes only affected the version of Go used, which was initially 1.9[5] to a more recent one, 1.17[6]. These two versions were released four years apart, which is an enormity in a programming language as recent as Go. Significant strides have been made in the meantime regarding garbage collector operation and package dependency management. We made another change to the underlying code to extrapolate the Go runtime metrics. We used Prometheus[7] to manage all the telemetry parts and, therefore, the retrieval of metrics from the various components. Before Prometheus can monitor application services, we need to add instrumentation to our code via one of the Prometheus client libraries. Client libraries lets define and expose internal metrics via an HTTP endpoint on the application's instance. After implementing the Go library, Prometheus then, at regular and pre-established time intervals, makes calls to the services of interest, receives the related metrics, and then saves them in the form of time series. Later it is possible to query the time series through its language: PromQL. The high-level architecture of Prometheus is reported in Figure 3.1.

[5] *Go 1.9*. URL: `https://go.dev/doc/go1.9`.
[6] *Go 1.17*. URL: `https://go.dev/doc/go1.17`.
[7] *Prometheus*. URL: `https://prometheus.io/`.

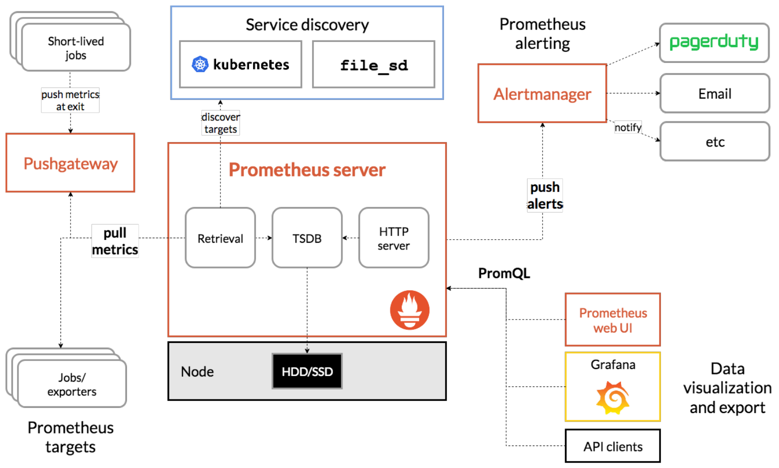

**Figure 3.1:** Prometheus architecture.

### The load generator

For generating our workload, we had several options: for example, the Online Boutique benchmark that we mentioned earlier uses Locust[8] that is open source, but there are also other ones that are more directed toward an enterprise environment, like NeoLoad[9], LoadRunner[10] and JMeter[11]. Defining the logic behind the request generation for the application takes time and is often bound to the preferred language of the load generator; for this reason, we chose to use the one already supplied with the tested application: wrk2[12]. This load generator allows you to define the test execution logic through a script in Lua, in which you can define which calls to make to the application to be tested, how often, and which results to report in the final report generated. Unfortunately, the final test metrics provided by wrk2 do not include a time series of the test but only the summary data metrics, which is fine for high-level aggregations but does not allow us to understand if, during the test, the metrics have remained constant or if there were spikes in response times which, in a production environment, could lead to problems. For this reason, as already mentioned, it was decided to add Istio to the system to continuously control the data flow between the various services and the related response time metrics.

## 3.2 Amazon Web Services

Amazon Web Services (AWS) is a provider of on-demand cloud computing platform to individual, companies and governments. In particular we used two services provided by AWS:

[8] *Locust*. URL: https://locust.io/.
[9] *Tricentis Neoload*. URL: https://www.tricentis.com/products/performance-testing-neoload/.
[10] *Microfocus LoadRunner*. URL: https://www.microfocus.com/en-us/products/loadrunner-professional/overview.
[11] *Apache JMeter*. URL: https://jmeter.apache.org/.
[12] *wrk2*. URL: https://github.com/giltene/wrk2.

* Amazon Elastic Compute Cloud (EC2), that allows to rent virtual computers on which to run applications;

* Amazon Elastic Kubernetes Service (Amazon EKS), a managed container service to run and scale Kubernetes clusters.

All the benchmarks considered have been run on EC2 instances, how Amazon call virtual machine. Amazon EC2 provides a wide selection of instance types optimized to fit different use cases, that comprehend varying combinations of CPU, memory, storage, and networking capacity. The reason to use AWS as cloud provider for computing resources is mainly due to these factors: AWS is considered the most enterprise-ready[13], mission critical hyper-scale provider; it's one of the most known and used cloud provider that allows us to guarantee that when running our experiments we are in environment similar to the production ones. These factors lead to the adoption of AWS as cloud provider for our experiments.
For our EKS cluster we have instantiated three EC2 machines:

* a `c5.2xlarge` instance for the benchmark application;

* a `c5.large` instance for the load generator, wrk2;

* a `t3.large` instance for Prometheus fetching metrics.

The choice of these types of instances was mainly dictated not to be too castrated in terms of performance of the benchmark application but to contain the cost of the cluster as much as possible. For this reason, we initially opted for Spot instances, a type of instance much cheaper than the standard ones but can be terminated and restarted at any time based on the provider's resource needs. However, restarting the instances resulted in problems retrieving some metrics related to the cluster nodes and a slight variation in the performance of the benchmark, so in the end, we used standard instances. In Table 3.1 is reported some characteristics of the EC2 instances that we chose.

## 3.3 The parameters

During the preliminary phases of the project, there was a research phase to understand which runtime parameters we could act on and how they could influence the behavior of the Go runtime and the environment in which the application lives, Kubernetes. Unlike other programming languages, the Go runtime has far fewer configurable parameters. For example, in Java the JVM has hundreds of configurable parameters, some of which considerably affect the execution of an application; Go, from this point of view is minimal, having only two of them. This makes it easier to use its runtime "as is" but does not allow you to optimize its performance and efficiency to the maximum.

[13] *Gartner: AWS.* URL: https://www.gartner.com/doc/reprints?id=1-2710E4VR&ct=210802&st=sb.

| Instance type | CPU | vCPU | Memory (GiB) | Network (Gbps) | EBS (Mbps) |
|---|---|---|---|---|---|
| c5.2xlarge | Intel Xeon Cascade Lake 8275CL | 8 | 16 | Up to 10 | Up to 4,750 |
| c5.large | Intel Xeon Cascade Lake 8275CL | 2 | 4 | Up to 10 | Up to 4,750 |
| t3.large | Intel Xeon Cascade Lake 8259CL | 2 | 8 | Up to 5 | Up to 2,780 |

**Table 3.1:** AWS EC2 instances chosen for the experiments.

### 3.3.1 Go

As anticipated in the previous chapters, Go has only two configurable runtime parameters: `GOGC` and `GOMAXPROCS`.

#### GOGC

`GOGC` determines the trade-off between GC CPU and memory at a high level. It works by determining the target heap size after each GC cycle, a target value for the total heap size in the next cycle. The GC's goal is to finish a collection cycle before the total heap size exceeds the target heap size. Total heap size is defined as the live heap size at the end of the previous cycle, plus any new heap memory allocated by the application from the last cycle. Meanwhile, target heap memory is defined as:

$$Targetheapmemory = Liveheap + Liveheap * GOGC/100$$

For example, consider a Go program with a live heap size of 8 MiB, 1 MiB of goroutine stacks, and 1 MiB of pointers in global variables. Then, with a `GOGC` value of 100, the amount of new memory allocated before the next GC runs will be 10 MiB, or 100% of the 10 MiB of work, for a total heap footprint of 18 MiB. With a `GOGC` value of 50, then it'll be 50%, or 5 MiB. Finally, with a `GOGC` value of 200, it'll be 200%, or 20 MiB. The heap target controls GC frequency: the bigger the target, the longer the GC can wait to start another mark phase and vice versa. So while the precise formula is helpful for making estimates, it's best to think of `GOGC` in terms of its fundamental purpose: a parameter that picks a point in the GC CPU and memory trade-off. The key takeaway is that doubling `GOGC` will double heap memory overheads and roughly halve GC CPU cost, and vice versa. Note that `GOGC` may also be used to turn off the GC entirely by setting `GOGC=off`. Conceptually, this setting is equivalent to setting `GOGC` to a value of infinity, as the amount of new memory before a GC is triggered is unbounded. GC frequency is still the primary way the GC trades off between CPU time and memory for throughput, and in fact, it also takes on this role for latency. This is because most of the costs for the GC are incurred while the mark phase is active. The key

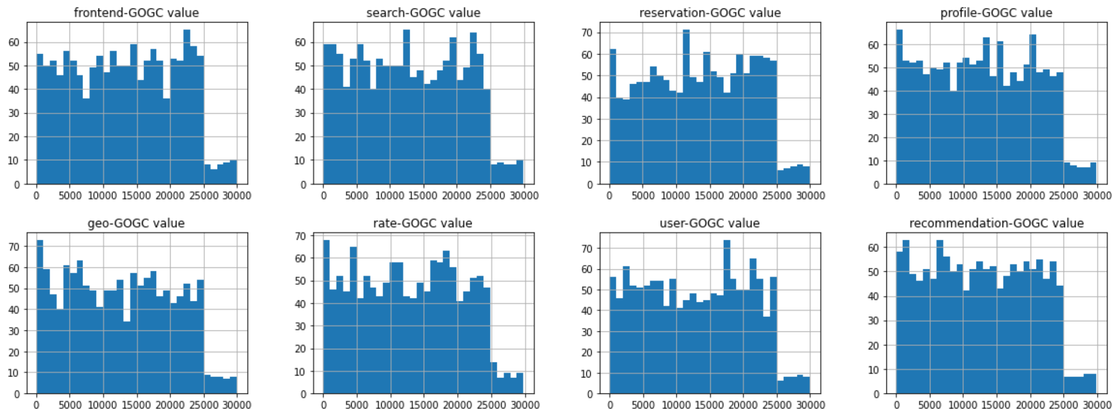

**Figure 3.2:** Distribution of GOGC parameters.

takeaway is that reducing GC frequency may also improve latency. However, latency is often more complex to understand than throughput because it is a product of the moment-to-moment execution of the program and not just an aggregation of costs. As a result, the connection between latency and GC frequency is less direct. The domain of the parameter tested in the experiments is [1-30000]. Figure 3.2 shows the distribution of `GOGC` parameters in all the experiments performed.

### GOMAXPROCS

As already mentioned, the operating system schedules threads to run against available processors, and the Go runtime schedules goroutines to run within a logical processor that is bound to a single operating system thread. The `GOMAXPROCS` environment variable allows to set the number of OS threads that can execute user-level Go code simultaneously. Starting with Go version 1.5, the default value of `GOMAXPROCS` is the number of logical cores available in the machine. Set `GOMAXPROCS` to a value smaller than the default one involves the allocation of a greater number of goroutines on individual threads, while setting it to a higher value means that, in addition to the overhead due to the allocation of multiple OS threads, the Go runtime no longer has complete control over the scheduling of the goroutines, since thread scheduling is an operation of interest to the OS. It's important to note that there is no limit to the number of threads that can be blocked in system calls on behalf of Go code, and those do not count against the `GOMAXPROCS` limit. So, the number of OS threads used by a running Go code can exceed the number set by `GOMAXPROCS` when there are blocking syscalls or any operation that puts a goroutine in a waiting state for some external resource. The domain of the parameter tested in the experiments is [1-100]. Figure 3.3 shows the distribution of `GOMAXPROCS` parameters in all the experiments performed.

## 3.3.2 Kubernetes

Kubernetes provides a shared pool of resources that it allocates based on how we configure our containerized applications. The allocation process occurs when a scheduler places pods on nodes. After checking the container's resource configuration,

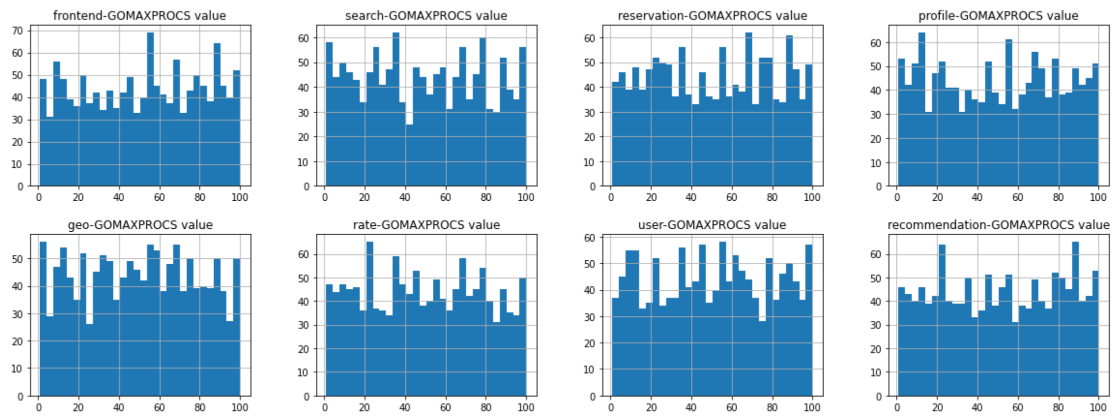

**Figure 3.3:** Distribution of GOMAXPROCS parameters.

the scheduler selects a node that can guarantee the availability of the resources specified by the container configuration. It then places the container's pod on the suitable node. Typically, this ensures that the deployed microservice will not encounter a resource shortage. However, some workloads may require more resources than the containerized application has been configured to use. Without a means to restrict the resources an application can request to use, the service can incur high costs, wasted resources, poor application performance, and even application failures. Therefore, it is essential to implement a system of boundaries to prevent resource waste and costly failures. This is why resource requests and limits are vital components of the Kubernetes environment. Resource requests and limits are optional parameters specified at the container level and can be applied to CPU and memory. Kubernetes computes a Pod's request and limits as the sum of requests and limits across all of its containers. Kubernetes then uses these parameters for scheduling and resource allocation decisions. For implementing these resource parameters, Kubernetes delegates to the container runtime (docker/containerd for example), and the container runtime delegates to the Linux kernel. When you set a limit but not a request, Kubernetes defaults the request to the limit. From the scheduler's perspective, it makes sense, since it has to ensure that a node has sufficient resources to start the container.

### Memory resources

To control the amount of memory that a container process can access, docker configures a property of a control group or cgroup. Cgroup controls how the kernel runs a process, and there are specific cgroups to control memory, CPU, devices, etc. Cgroups are hierarchical, meaning that each cgroup has a parent from which it inherits properties, all the way up to the root cgroup, which is created at system start. The container `memory limit` value is mapped to the cgroup property `memory.limit_in_bytes` and, when a host comes under memory pressure, the kernel may elect to kill processes. Processes that are not in memory cgroups are handled by the global *oomkiller*. When the kernel cannot allocate pages it will essentially kill the process using the most physical ram, scaled by a factor called the `oom_score_adj` that is used to protect essential processes. Processes that are in

memory cgroups are affected by the cgroup oomkiller, which will always kill them if they set a limit and then exceed it. Since Kubernetes' job is to pack as much stuff onto a node as possible, memory pressure on those hosts is not uncommon. So if a container is using too much memory, it is likely to be oom-killed. If it is docker will be notified by the kernel, Kubernetes will find out from docker and, depending on the settings, may try to restart the pod.
`memory request` has a different role than `memory limit`. Limits tell the Linux kernel when to consider your process a candidate for freeing up memory. Requests help the Kubernetes scheduler figure out where it can run a pod. Not setting them, or placing them artificially low, can have harmful effects. Furthermore, Kubernetes use `memory request` for scheduling on the nodes, but this parameter isn't forwarded to the Linux kernel; in fact the cgroup does not map this variable For example, suppose you run a pod with no `memory request` and a high `memory limit`. As we saw, Kubernetes will default the request to the limit, and if no node has that much ram available, the pod will fail to schedule even though its actual requirements might be much less. On the other hand, if you run a pod with an artificially low request, you encourage the kernel to oom-kill it. Let's assume the pod usually uses 100 Mi of RAM, but it's run with a 50 Mi request. If the node has 75 Mi free, the scheduler may choose to run the pod there. When pod memory consumption later expands to 100 Mi, this puts the node under pressure. When this happens, the kubelet may decide to evict pods that are using more than their requested amount of memory, whether or not the pod has hit the hard limit. The domain of the parameters tested in the experiments varies based on the containers because they are strongly influenced by what the service does. Therefore we have discarded some values that a priori would not have generated interesting results. Figure 3.4 shows the distribution of `memory request` and `memory limit` parameters in all the experiments performed.

### CPU resources

CPU resources are controlled by the exact cgroups mechanism that also controls memory resources, but it's just a little bit more complicated. The unit of measurement of CPU resources is *millicore*, and stands for a thousand of a core, so **1000m** corresponds to a core in Kubernetes, but both cgroup and docker divide a core into 1024 shares. Unlike `memory request`, `CPU request` is propagated to cgroup and stored in the `HostConfig.CpuShares` property. The `CPU limit`, though, is a little less obvious. It is represented by two values: `HostConfig.CpuPeriod` and `HostConfig.CpuQuota`. These docker container configuration properties map to two additional properties of the process's CPU cgroup: `cpu.cfs_period_us` and `cpu.cfs_quota_us`, that are mapped to the same value specified in the `CPU limit`.
`CPU request` and `CPU limit` are implemented using two separate control systems.
`CPU request` uses the CPU shares system, the earlier of the two. Cpu shares divide each core into 1024 slices and guarantee that each process will receive its proportional share of those slices. If there are 1024 slices and each of the two processes

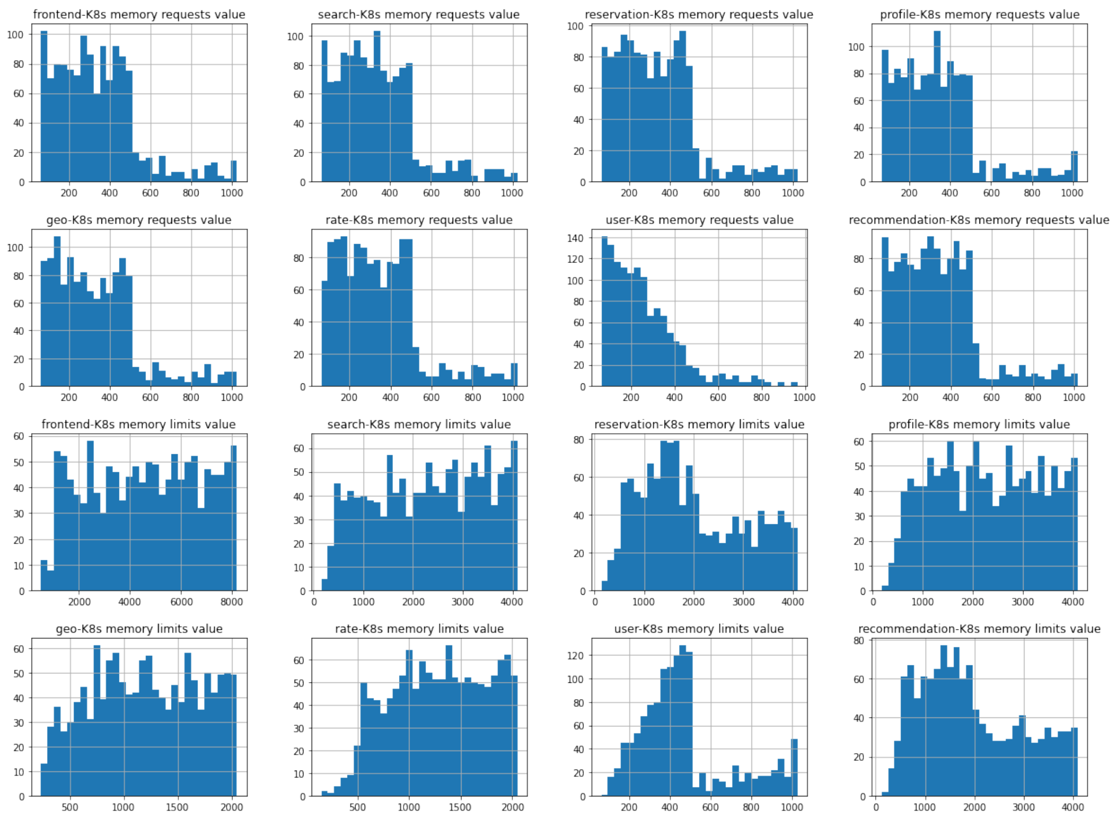

**Figure 3.4:** Distribution of Kubernetes memory requests and limits parameters.

sets `cpu.shares` to 512, then they will each get about half of the available time if there is contention for the resource. The CPU shares system, however, cannot enforce upper bounds. If one process doesn't use its share, the other is free to.
`CPU limit`, on the other hand, uses the CPU bandwidth control [67]. The bandwidth control system defines a period, usually 1/10 of a second, and a quota representing the maximum number of slices in that period that a process is allowed to run on the CPU. So, for example, if we set a `CPU limit` of 100m (that is 100/1000 of a core, or 10000 out of 100000 microseconds of CPU time), the limit translates to setting `cpu.cfs_period_us=100000` and `cpu.cfs_quota_us=10000` on the process's cgroup. The *cfs* in those names stands for *Completely Fair Scheduler*, which is the default Linux CPU scheduler.
So, setting a `CPU request` in Kubernetes ultimately sets the `cpu.shares` cgroup property, and setting `CPU limit` engages a different system through setting `cpu.cfs_period_us` and `cpu.cfs_quota_us`. As with `memory limit`, the request is primarily useful to the scheduler, which uses it to find a node with at least that many CPU shares available. Unlike `memory request` setting, a `CPU request` also sets a property on the cgroup that helps the kernel allocate that number of shares to the process. Limit is also treated differently from memory. Exceeding a `memory limit` makes a container process a candidate for oom-killing. In contrast, a process basically can't exceed the set CPU quota and will never get evicted for trying to use more CPU time than allocated. The system enforces the quota at the scheduler, so the process just gets throttled at the limit. The domain of the

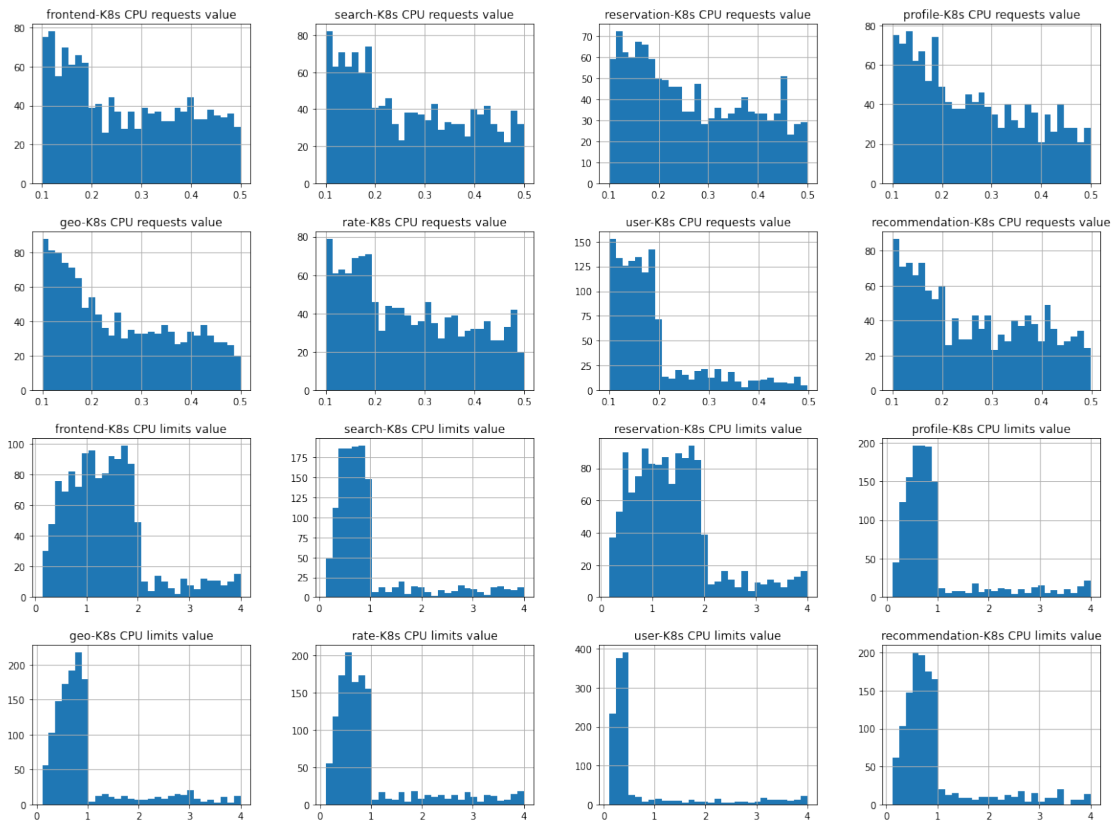

**Figure 3.5:** Distribution of Kubernetes CPU requests and limits parameters.

parameters tested in the experiments varies based on the containers because they are strongly influenced by what the service does. Therefore we have discarded some values that a priori would not have generated interesting results. Figure 3.5 shows the distribution of CPU requests and limits parameters in all the experiments performed.

## 3.4 Akamas

Akamas is an AI-powered autonomous optimization solution that enables to optimize a technology stack respecting the constraints set and the goals to pursue. The technology stack's domain was new, so we did not base Akamas' processes on artificial intelligence algorithms to optimize the goal. As our interest was to evaluate the entire domain of the parameters to evaluate how these affect some objectives, primarily the response time of services and their use of resources.

### 3.4.1 SOBOL

The time needed to visit all the possible configurations depends on the number of parameters and their cardinality. Figure 3.6 shows on y-axis the amount of time, in years, needed to entirely explore a space composed by the number of parameters in the x-axis. This represents a lower bound for our problem because the graph is computed considering only binary parameters, i.e. True or False values, which is

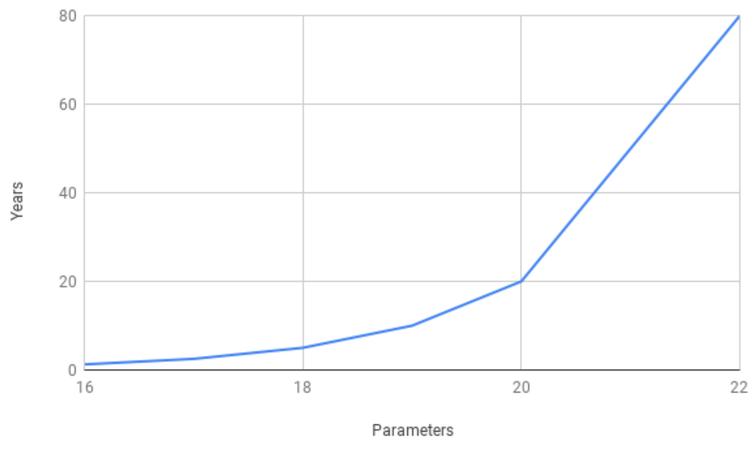

**Figure 3.6:** Lower bound of the time needed to obtain a full dataset based on the total number of parameters.

not our case. The actual equation used is:

$$y = \frac{2^x * 10}{60 * 24 * 365}$$

where $2^x$ represents the number of combinations of x elements with two values, 10 is the average number of minutes to execute an experiment with a single configuration, and the denominator is the number of minutes in one year. It's clear that we can not explore the entire space since we have a limited amount of time to complete this work, but instead we should rely on a smart way to explore the space. For this task, we relied on Sobol. Sobol is an algorithm that generates quasi-random low-discrepancy sequences, first introduced by Ilya M. Sobol in 1967. Low-discrepancy sequences are designed to look like random sequences. However, those sequences' primary objective is to fill the unit hypercube of parameters as uniformly as possible. Sobol aims to generate points in the entire n-dimensional plane such that the points are uniformly distributed in that space. When we add more points to the sequence, they will be generated to fill the spaces between the already generated points. This way, we can generate as many points as we like and still have a uniform distribution. Figure 3.7 shows the comparison between filling a 1x1 space grid with 2000 points using a pseudorandom algorithm and Sobol. From this visual example, it is clear that Sobol is suitable for exploring large parameter spaces by uniformly sampling each dimension. From the histograms above the figures, we can notice how the Sobol sequence is more uniformly distributed on the x-axis than the pseudorandom one. Another property of Sobol is the fact that the generated sequence is deterministic. This allows someone else to extend the dataset, going on by sampling new data points to increase the level of detail. Furthermore, it permits to parallelize the data retrieval easily.

### 3.4.2 Experiment pipeline

Choosing the next parameters to configure is just a little step in the process of running the entire experiment. All this pipeline is managed by Akamas and can be

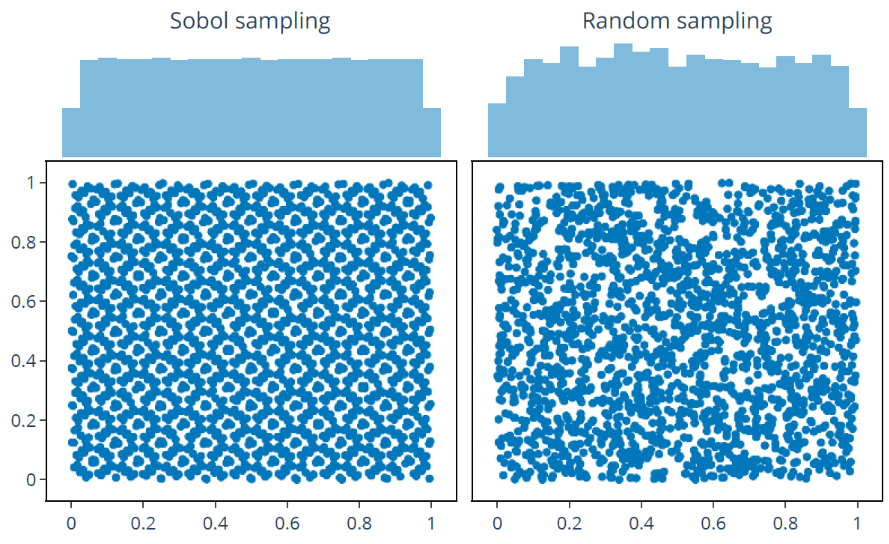

**Figure 3.7:** Exploring a bidimensional 1x1 space using a Pseudorandom or Sobol.

summarized in this steps:

1. **Configure Services**: during this phase the new configuration (based on Sobol algorithm) is applied to the Kubernetes deployment, copying the new parameters into the template that we previously define;

2. **Apply new configuration and restart pods**: the new configuration is deployed into the Kubernets cluster using a pipeline of these three command concatenated:

   ∗ restart the Consul service, used for the networking between the services. This was necessary because we noticed that sometimes Consul doesn't update it's routing table if it has already in memory other entries;
   `kubectl rollout restart -n hotel-res deployment consul`

   ∗ apply the new configuration to the cluster and wait ten seconds;
   `kubectl apply -f hotelRes/hotel-res.yaml && sleep 10s`

   ∗ restart all the services. We made this because it happens that some services wasn't affected by the new configuration, so we can start the new experiment with a clean start for all the container;

   ```
   kubectl rollout restart -n hotel-res deployment frontend
   mongodb-user mongodb-reservation mongodb-recommendation
   mongodb-rate mongodb-geo mongodb-profile memcached-profile
   memcached-rate memcached-reserve user geo search reservation
   recommendation profile rate
   ```

3. **Check system up**: check that Kubernetes has scheduled all the services and that they are up and running;

```
declare -a services=(frontend user search
   reservation recommendation rate profile geo)
for service in "${services[@]}"; do
    echo "Describe service $service"
    kubectl describe pod "$service" -n hotel-res
        | grep -Pzo '.*Events(.*\n)*'
    echo "Checking rollout status for service
       $service"
    kubectl rollout status --timeout=3m -n hotel
       -res deployment "$service"
done
```

4. **Drop mongodb reservation**: we noticed that, after many experiments, the script used by the load generator no longer had possible combinations between the pre-configured users and the available rooms. That is, it created problems because, in some cases, no writes were made to the database, which made the experiments inconsistent. There were two alternatives: either rewrite part of the load generator script or drop the database to start from scratch every time. We opted for this option as it seemed to us the best to make the experiments as fair as possible with each other;
   ```
   kubectl exec -it svc/mongodb-reservation -- bash -c
   "mongosh reservation-db --eval 'db.dropDatabase()'"
   ```

5. **Start load generator**: this is the beginning of the experiment. The load generator script starts and we start collecting metrics about the system, the Go runtime, and the Kubernetes cluster;
   ```
   kubectl exec wrk2 -- /home/wrk2/wrk -t1 -d10m -c800 -R1100
   -L -s /home/wrk2/mixed-workload_type_1.lua
   http://frontend.hotel-res.svc.cluster.local:5000/index.html
   ```

6. **Copy results**: after the load generator finishes, we retrieve the summary metrics it generates. We have chosen not to use them because we inserted a "start-up phase" (the first five minutes of the load generator execution) so the system stabilizes. Another reason we do not use these metrics is that they are summary metrics, already aggregated to obtain the average of the results. At the same time, we were also interested in having the time series of the results. For this reason, as already mentioned, we have included Istio in the system and relied on that and Prometheus for collecting the metrics of interest.
   ```
   kubectl cp hotel-res/wrk2:result.csv /home/hotelRes/result.csv
   ```

### 3.4.3 Workload generator parameters

As previously mentioned, we used wrk2 to generate the load on our system. This application requires six configuration parameters (as can also be seen from the command in the point 5 used to launch its execution), which we will explain in this section, together with the reasons that led us to these choices.

1. The first parameter is the number of threads assigned to the load generator: in our case, we did not notice any differences as this parameter changed, so we left it at the default value.

2. The second parameter is the time for which you want to run the test: in our case, we opted for 10 minutes, then excluded the first 5 minutes that we considered as system warm-up from the data set, noting that in the first minutes the statistics relating to the Go runtime had unreliable values, as there were not yet enough values obtained, and then stabilized in the second period.

3. The third parameter is the number of parallel connections to open to the target service. This parameter can be seen as the number of users using the application simultaneously.

4. The fourth parameter is the target of requests per second to the application. As for the number of connections to the service, we chose this parameter after testing the system with the default parameters and obtaining a baseline for each configuration. Moro on this later.

5. The fifth parameter is the path of the Lua script in which it is defined how to generate the load, for example, which HTTP GET and POST calls make to the service, and which aggregate data to report at the end of the test.

6. The sixth parameter is the target link to direct requests to.

To decide the number of connections and the RPS target (requests per second), we first tested the system with the Go runtime and Kubernetes configuration parameters at default values. Our choice was then based on two conditions:

∗ that the system was under load but that it managed to maintain constant throughput and constant response times for the entire duration of the test;

∗ that the cluster CPUs remained around 50 % of the load to avoid the intervention of phenomena related to Hyper-Threading[14], which are unpredictable and complicate the analysis of the dataset.

As can be seen from the results of the first tests, visible in Figure 3.8 and 3.9, the system performance begins to degrade with RPS >3000. For the second condition, however, the limit came first: the overall load on the CPU was, in fact, 70/75 % with 3000 RPS and 55/60 % at 2500 RPS. Furthermore, since the response time had unusual behaviors with 100 and 200 parallel connections, we decided to target 500 connections and 2200 RPS as the target of the load generator. After running the first tests, however, we realized that we could not rely on the statistics provided by the load generator regarding response times. This is mainly because, in the aggregate calculation of the metrics that this returned, it also took into account the first 5 minutes of testing that we had considered as "warm-up" and did not provide

[14] *Hyper-Threading.* URL: `https://www.intel.co.uk/content/www/uk/en/gaming/resources/hyper-threading.html`.

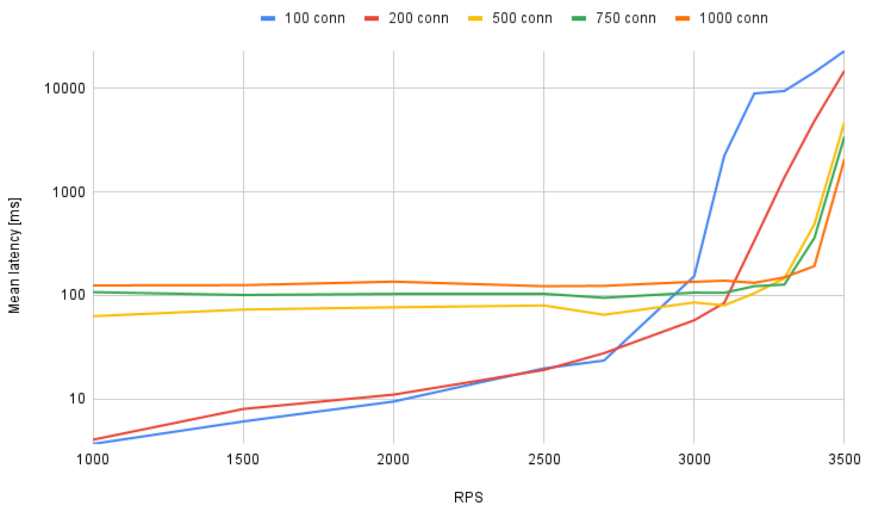

**Figure 3.8:** Median latency for default parameters without Istio.

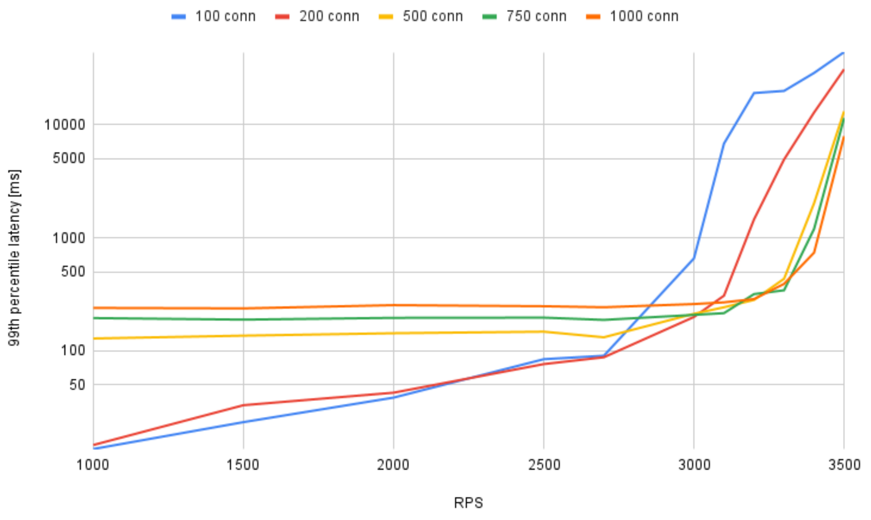

**Figure 3.9:** 99th percentile latency for default parameters without Istio.

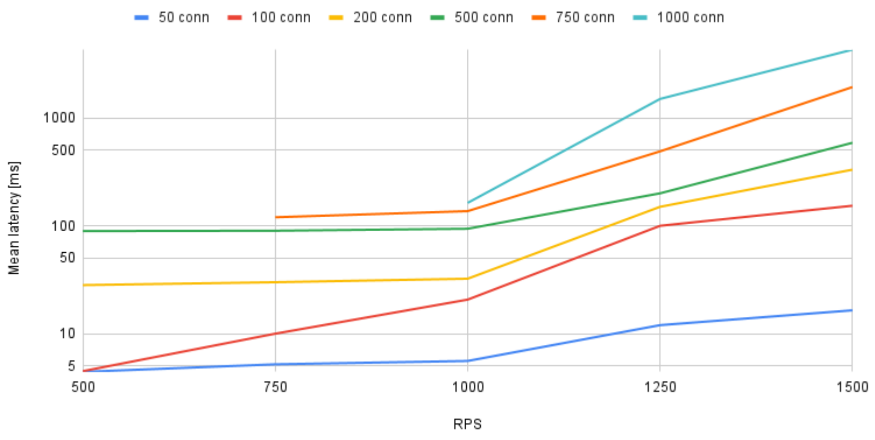

**Figure 3.10:** Median latency for default parameters with Istio.

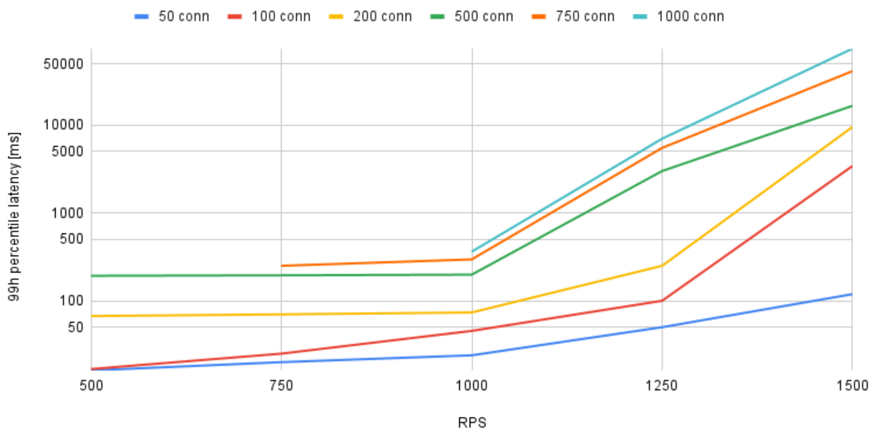

**Figure 3.11:** 99th percentile latency for default parameters with Istio.

results as time series but only aggregate counts. For this reason, as previously explained, we have decided to add Istio to the system. However, this entailed many problems in the system's behavior under load. It was necessary to re-run the tests to define the load generator parameters and recalibrate them, considering the resources used by Istio. Figure 3.10 and 3.11 shows the new response times with Istio present in the system, and you can immediately see how the performance is degraded. As you can see, the system performance is significantly degraded, as is the CPU usage. Service mesh services, as also explained in the 2.7 section, have various benefits but are rather influential on system resources. For this reason, we re-evaluated our target for the 800 connection and 1100 RPS load generator, as we saw that with these parameters, the load on the CPU was ∼50%, and the response time was stable.

# Chapter 4

# Results

In this chapter, we will present the analysis of the dataset obtained from the experiment. The methodology with which the experiments were carried out was explained in the previous chapter, but in reality, we have not performed a single, long experiment. We have collected three different types of datasets:

* a set of experiments in which we have optimized only Go parameters;
* a set of experiments in which we have optimized only Kubernetes parameters;
* a set of experiments in which we have optimized all the parameters at our disposal.

The latter dataset has more experiments, allowing us to have a complete view of the system and see how others influence some parameters. For this reason, however, we have chosen to perform the first two sets of experiments to have a clearer idea and limit the experiments' domain so we can compare them in case of doubt. We have chosen to do this because, from the beginning, there was the awareness that trying to optimize a parameter, `GOGC`, which can involve an increase in memory used by a service and, at the same time, optimize another, `memory limit`, which limits the memory available to the same service, could have been limiting. Most of the following analysis and the following graphs, however, come from the analysis of the complete dataset, as we have not noticed particular problems with the optimization of all the parameters together, having been performed:

* 466 experiments in which we have optimized only Go parameters;
* 328 experiments in which we have optimized only Kubernetes parameters;
* 2214 experiments in which we have optimized all the parameters at our disposal.

Each experiment was repeated twice, i.e., the pipeline described in the chapter 3.4.2 was run with the same parameter values for two successive experiments so that we could then calculate the average value for each configuration and have values on which to base the analysis that were not too influenced by random events that might have affected system performance. The points in the dataset on which the analysis was performed, therefore, are half of those described in the previous list. Another important thing for understanding some of the graphs is that in cases where experiments are placed on the x-axis in numerical order, the first value always corresponds to the average of the baselines and thus is to be understood as the starting value with the default parameters.

## 4.1 Preliminary analysis

As a first insight into the benchmark application, we want to know how the different services' response times depend on each other. We already know how these communicate and what calls are redirected to which service. However, looking at the correlation between response times seems a good start to understanding later graphs. We can notice from Figure 4.1 how *frontend* and *reservation* services' response times are closely related: this is expected, as the reservation service is the only one that writes to the database, so we can expect a longer response time that influences the frontend when there is a write in it. We can also notice an interesting correlation between *rate* and *search* services: search service, for each call, decomposes the request towards geo and rate; the latter manages the availability of rooms in a hotel in a specific period, uses a query that takes some time and hence the correlation between the two services.
Another aspect that we want to dig deeper as preliminary analysis is the CPU utilization of each services. As we already mentioned, we want to stay Another aspect we wanted to explore in these preliminary analyses was the CPU utilization by each container. As mentioned earlier, we wanted to stay on a ∼50% CPU utilization to avoid Hyper-threading-related phenomena, and Istio's utilization took up a good part of this capacity. As can also be seen from Figure 4.2, the sum of the containers uses ∼1.4vCPU, considering the 8vCPU available in the chosen instance, means that Istio alone consumed more computational resources than the sum of the services. From this, we can assume that optimising the Go runtime in a production environment with more effective use by services could lead to better results than those tested during our experiments.

## 4.2 First experiments results

As the first general checks of the results, we wanted to get a general idea of how the experiments had gone and check that there were no events that could significantly affect the results obtained. Figure 4.3 shows how the parameters was tuned during the experiment that configure both Go and Kubernetes parameters. It can be seen that there are "steps" at a certain point: during those experiments, we decided to limit the domain to act on to run more experiments in that domain. We did this because we noticed how the domain initially defined was unnecessarily large, and therefore some experiments provided results that were not very meaningful.
Another aspect they wanted to check was disk utilization by the benchmark. I/O operations are usually slow and are particularly influential on the response time of services. Therefore, we wanted to be sure that disk utilization was constant throughout the experiments. As Figure 4.4 shows, there are some peaks where there are more I/O operations in various experiments. However, these do not particularly

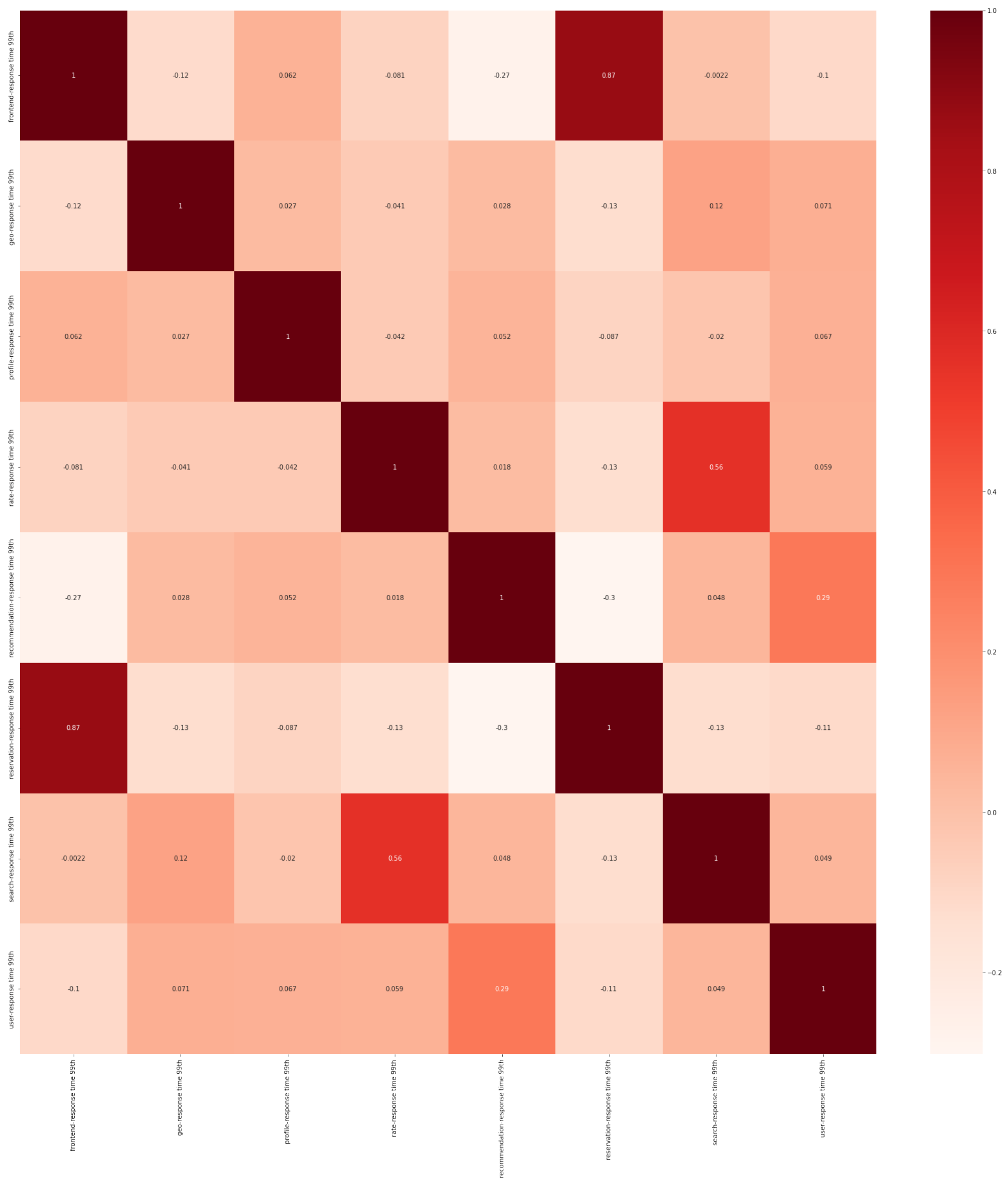

**Figure 4.1:** Correlation between services' response times

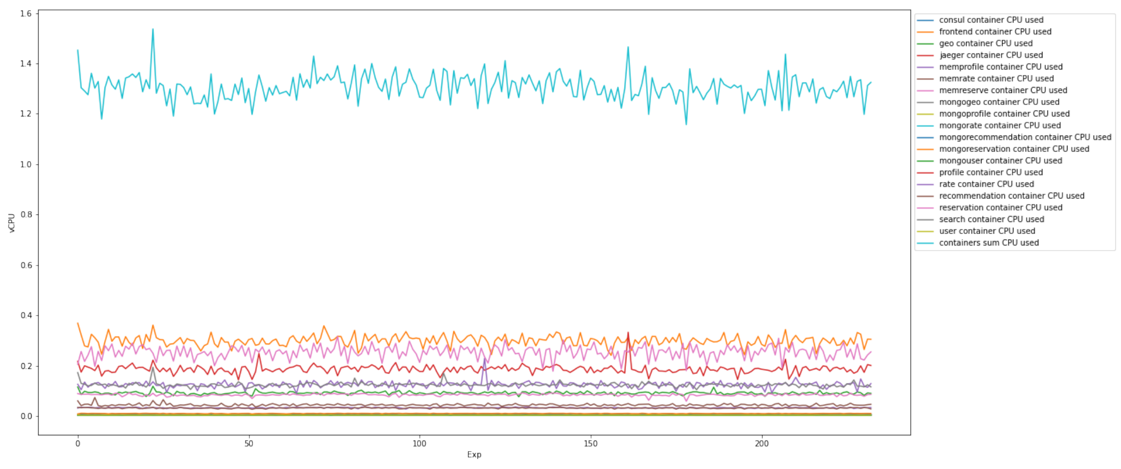

**Figure 4.2:** Containers' vCPU utilization

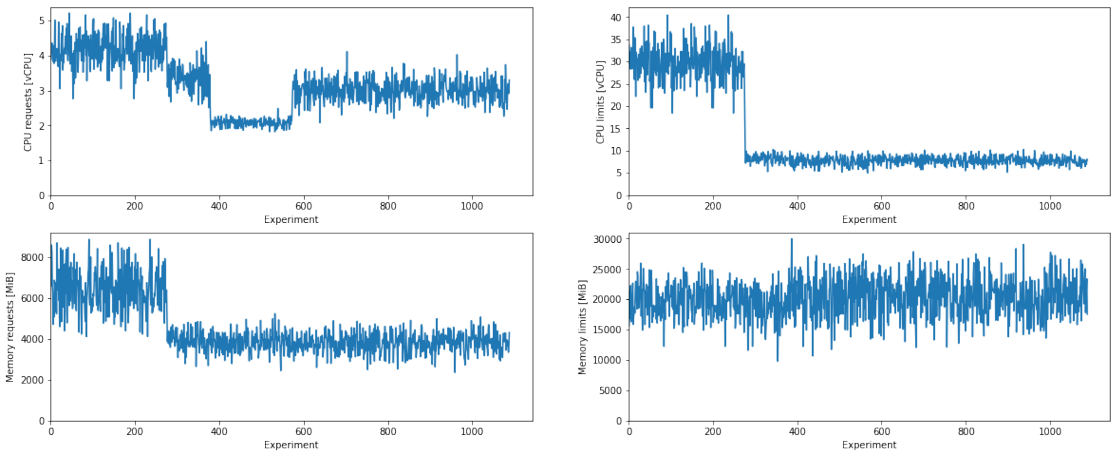

**Figure 4.3:** Parameters configuration over time.

affect read and write times (visible in the two central graphs), except in a few particular cases that are, however, minimal and of minimal value in absolute terms. This graph includes all experiments performed, so even those in which Go and Kubernetes are configured individually. The last thing we wanted to check was the overall response time at the various percentiles to see what was achieved regarding response time optimization. In Figures 4.5 and Figure 4.6, the entire dataset was considered but filtered to keep only those experiments in which response times (at all percentiles) are less than the baseline values. Another motivation for this is to take into account only those experiments in which the actual system RPSs are >1050, considering that the theoretical target of the load generator was 1000 RPS. This filter was applied in almost all tests to discard the experiments in which the target transactions could not be obtained for some reason. In most cases, this was because one or more containers were going out of memory and thus were unreachable. In this case, the response time to the affected service was practically zero, as it responded immediately with an error.

## 4.3 Goroutine and GOMAXPROCS

One of the most interesting concept of Go is goroutines, as explained in Section 2.2.1. One of the first things we wanted to verify is whether these are independent from Go's `GOMAXPROCS` parameter, which in theory should influence only the number of OS threads. To do this, we checked that the number of OS threads, the number of goroutines and the parameter `GOMAXPROCS` were correlated in practice. As can be seen in Figures 4.7, the number of goroutines and `GOMAXPROCS`, as well as the number of goroutines and the number of OS threads, are not correlated but depend on external parameters. On the other hand, the number of OS threads is strongly correlated by the parameter `GOMAXPROCS`, which confirms what is stated in theory and has implications, as we will see below.
We can notice how the number of goroutines seems to slightly depend on the number of requests successfully handled by the service, as seen in Figure 4.8. It should be noted that this is closely related to the benchmark implementation since

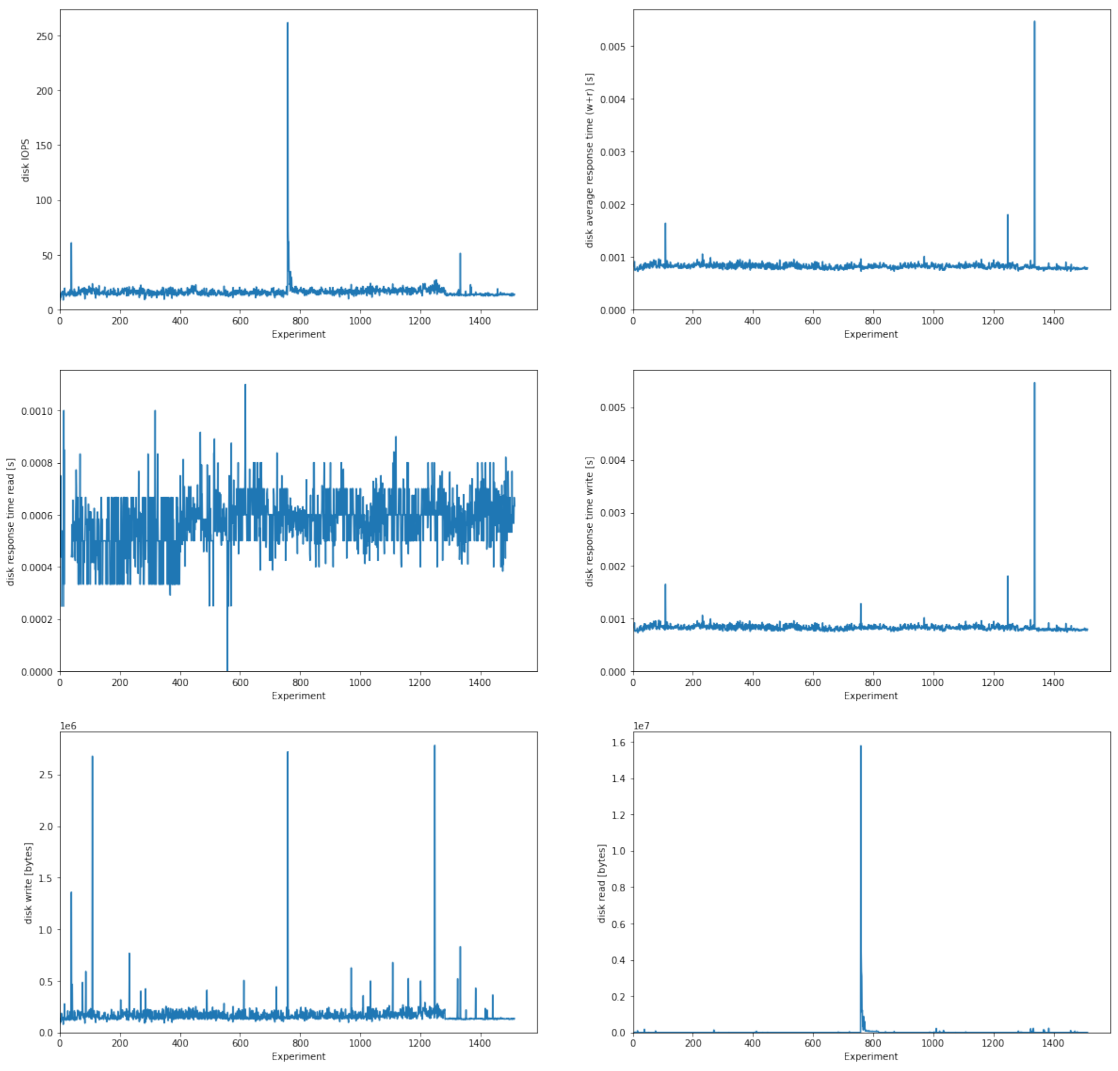

**Figure 4.4:** I/O operation metrics over time.

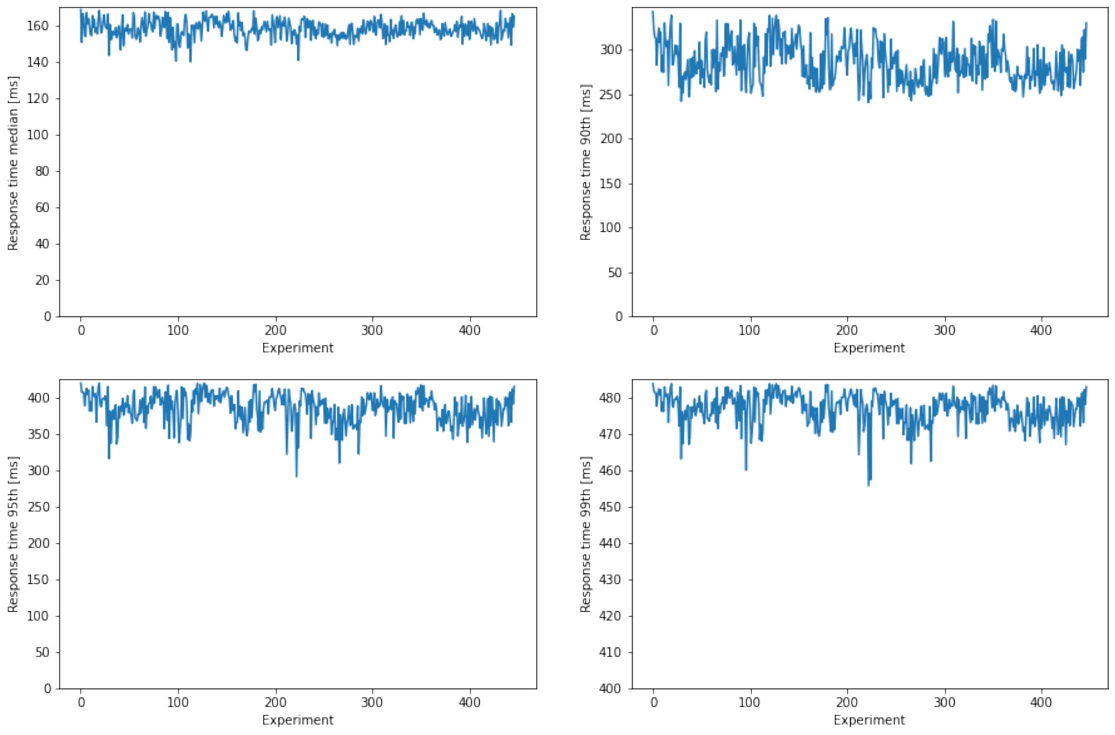

**Figure 4.5:** Response time percentiles over time.

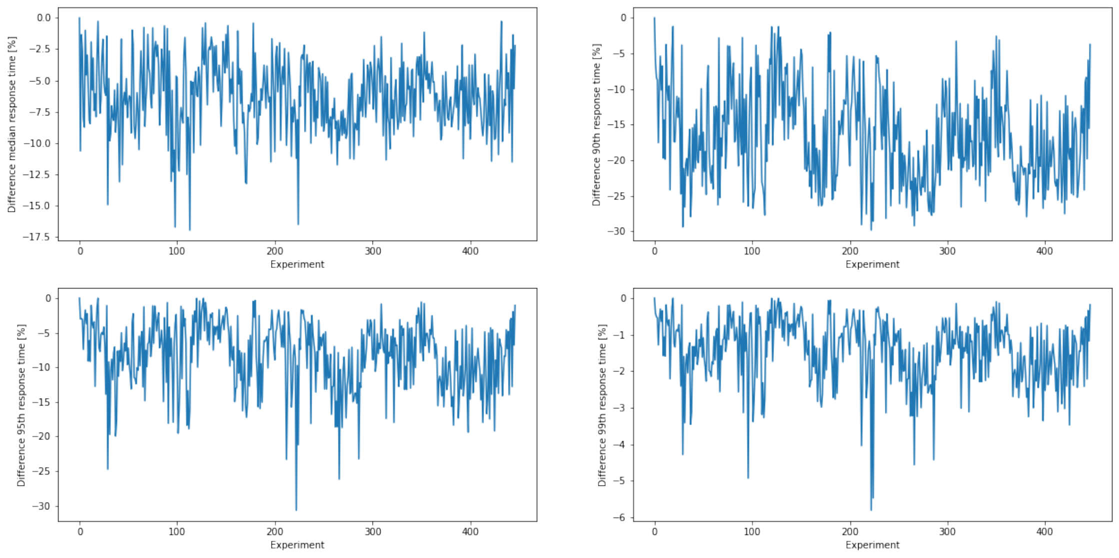

**Figure 4.6:** Gain % response time percentiles over time.

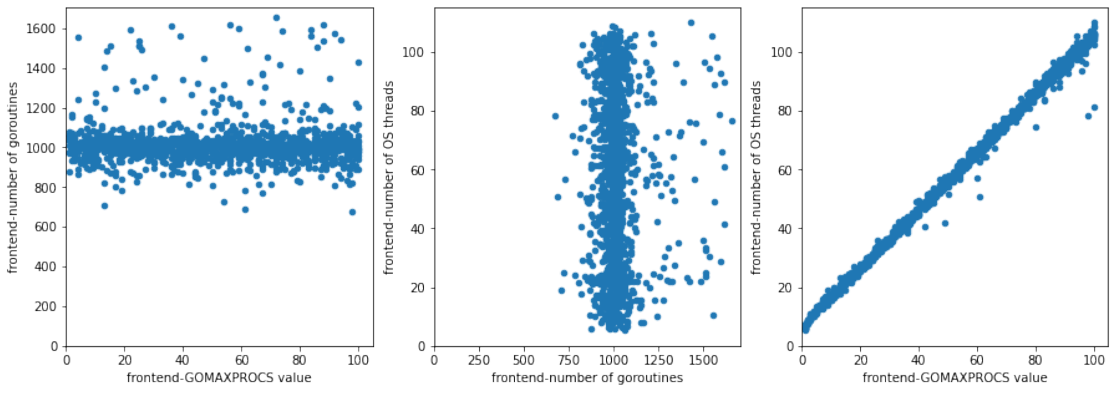

**Figure 4.7:** Correlation between GOMAXPROCS, OS threads and goroutines.

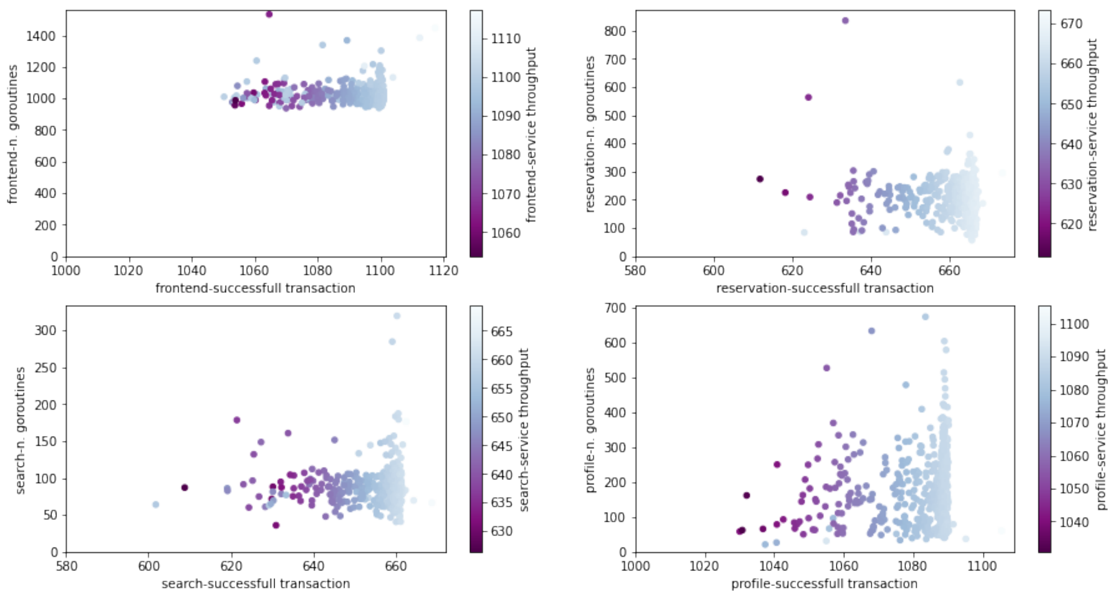

**Figure 4.8:** Number of goroutine and service throughput.

for each new request, a goroutine is spawned to handle it, which is very common in the various web services implementations in Go. Note how in the graphs just mentioned, the x-axis does not start from 0; otherwise, the graph would be too squashed. Also, for this graph, only experiments with an RPS number >1050 were considered as before to discard most of the problematic experiments. This filter will also be applied very often in future graphs. The next step was to see if response time (in this case at the 90th percentile) is related to the number of goroutines in a service, as shown in Figure 4.9. Again, no particular correlation is apparent, and the number of goroutines seems to be affected by other events. We also ran a few runs of the *Random Forest* algorithm for calculating *feature importance* to see if the number of goroutines was affected by some other metric, but without success. The most plausible thing is that these are mainly influenced by program logic or external events, such as the accumulation of goroutines in the queue in the case of decreasing processed requests of a service.

Two metrics that the `GOMAXPROCS` parameter certainly influences are the bytes

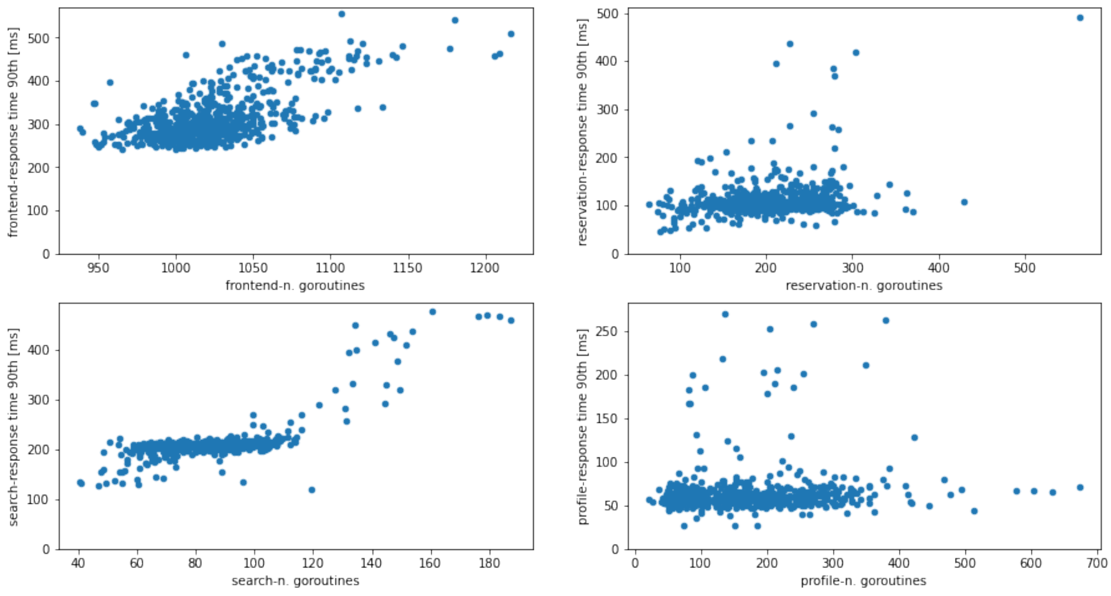

**Figure 4.9:** Number of goroutine and response time.

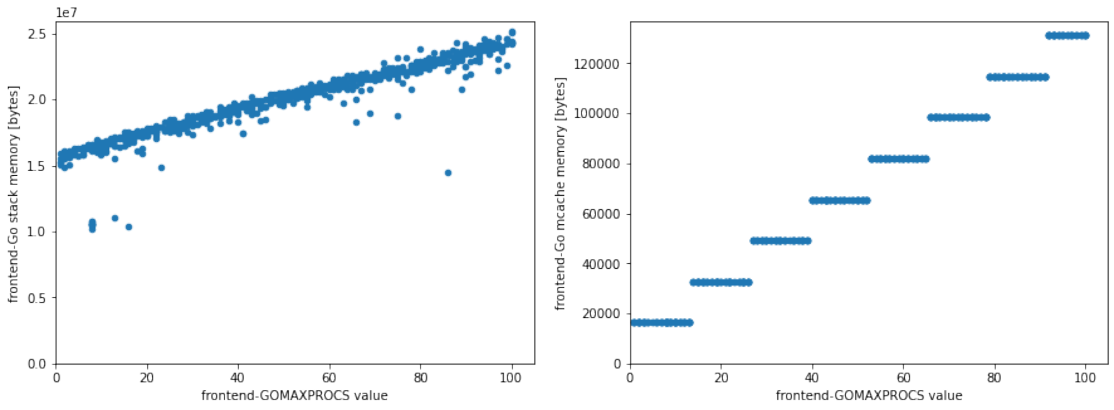

**Figure 4.10:** GOMAXPROCS and memory structures.

of stack memory obtained from OS and the bytes obtained from OS for *mcache* structures, as shown in Figure 4.10. The first one is quite intuitive since the number of OS threads is strongly correlated to `GOMAXPROCS`, so an increase in this parameter corresponds to an increase in memory dedicated to threads' stack. The second is also intuitive, considering that the mcache structure in Go is a per-thread (or per-P in Go's terminology 2.2.1) cache for small objects. Hence, an increase in the number of OS threads corresponds to an increase in memory dedicated to this data structure.

Regarding the influence of the `GOMAXPROCS` parameter on CPU, Figure 4.11 clearly shows how it negatively affects this resource. There remains a fair amount of variability in CPU utilization even at similar parameter values, but the trend is clear.

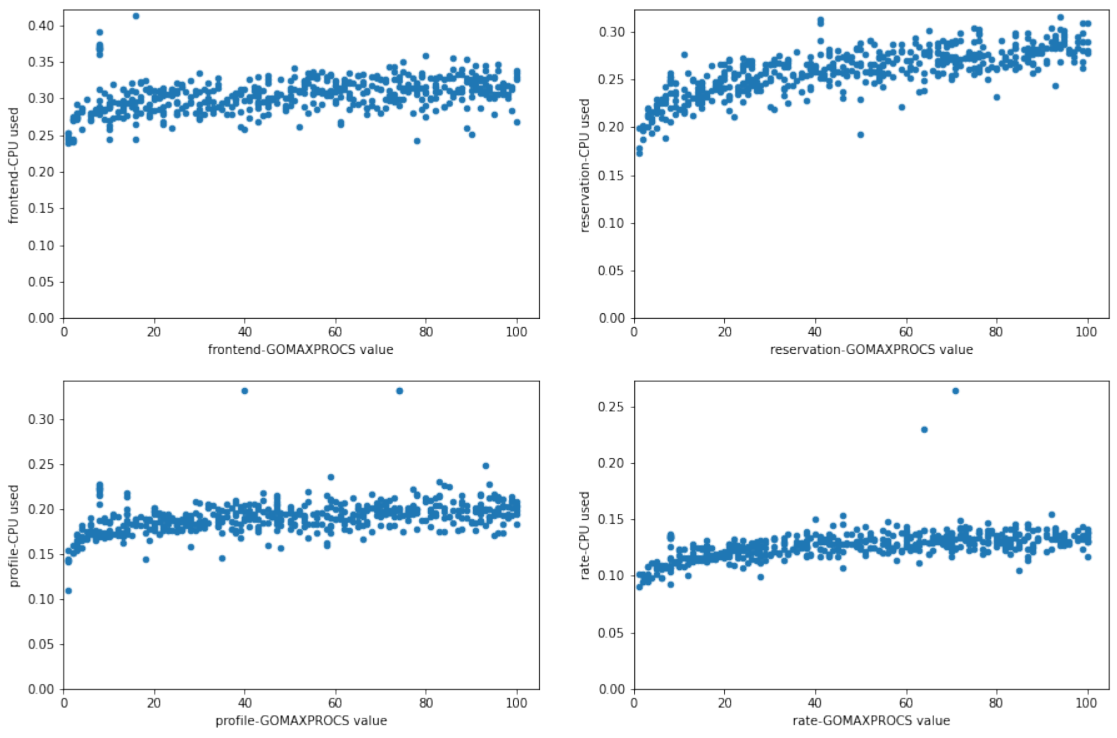

**Figure 4.11:** GOMAXPROCS and CPU utilization.

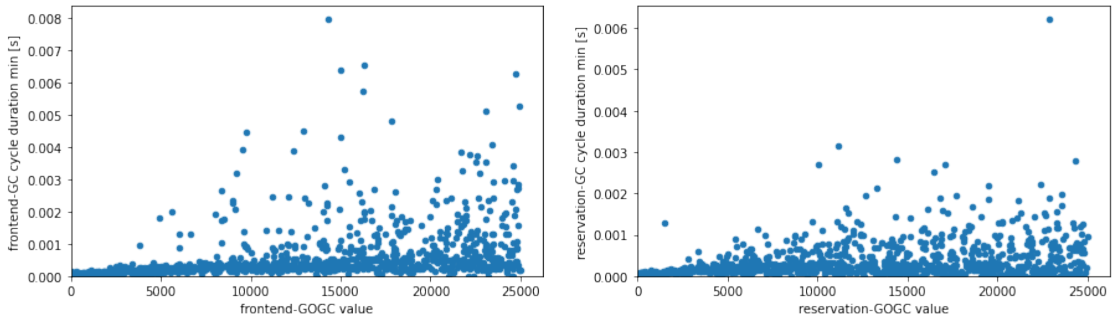

**Figure 4.12:** GOGC and GC pause time minimum value.

## 4.4 GOGC

One of the main aspects affecting the performance and resource usage by the runtime of a programming language is its garbage collector, as explained in Section 2.2.4. In this case, the parameter that acts on this is `GOGC`. As we can see from Figure 4.12, as this parameter varies, it can be seen that the minimum GC pause time (i.e., the minimum value of time in which the GC did not do its work concurrently with other goroutines but had to interrupt program execution), among all GC executions during the evaluated time interval of the experiment, increases as the value of `GOGC` increases.

What is most interesting, however, is how the GC pause times vary with the `GOGC` parameter as the various percentiles change. Figure 4.13 shows that, for the recommendation service, there is a kind of inversion for >50th percentiles or the sum of all GC pause values. Numerically, one can also see this trend from the

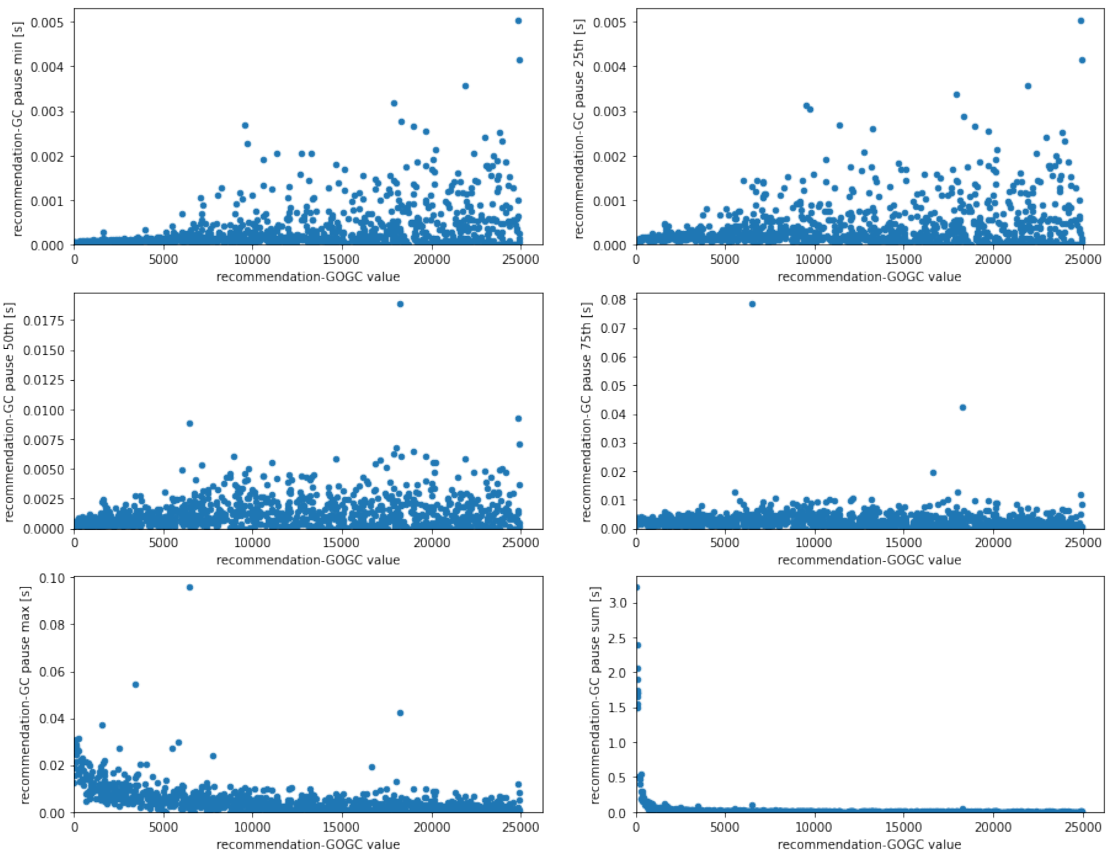

**Figure 4.13:** GOGC and GC pause time at different percentiles.

correlation graph plotted for the frontend service 4.14. The most important thing to note is how, as the value of `GOGC` increases, the pauses to program execution by the GC are more unfavourable when looking at the minimum pause times, but they become performance convenient when looking at the maximum pause times and especially when it comes to the sum of total pause times, where these decrease dramatically.

Finally, Figure 4.15 that the number of GC cycles decreases significantly as the `GOGC` parameter increases, further confirming how it affects the GC. It can be seen, however, that this decrease occurs mainly at `GOGC` values <5000, far below the maximum domain value we set (25000/30000). This could suggest further limiting the domain of this parameter for future experiments.

Another critical aspect to consider when configuring the `GOGC` parameter is how it affects resource consumption . While we expected an increase in the memory required by the individual container, since the GC as `GOGC` increases is invoked more rarely, we also expected CPU consumption to decrease. These assumptions were confirmed by the graphs shown in Figures 4.16 and 4.17. In particular, it can be seen that memory consumption increases almost linearly until it stabilizes at a certain value in some services. The most plausible hypothesis is that this behaviour again depends on the logic of the application under consideration: in fact, it is likely that after a specific value of `GOGC`, between one GC cycle and another, the service manages to keep in memory almost all the necessary data, and therefore the

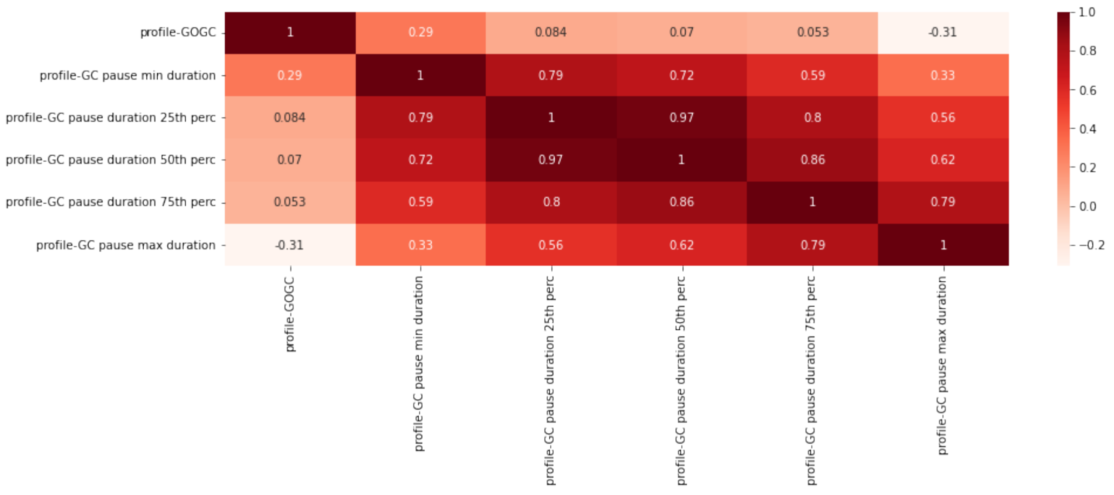

**Figure 4.14:** GOGC and GC pause time correlation.

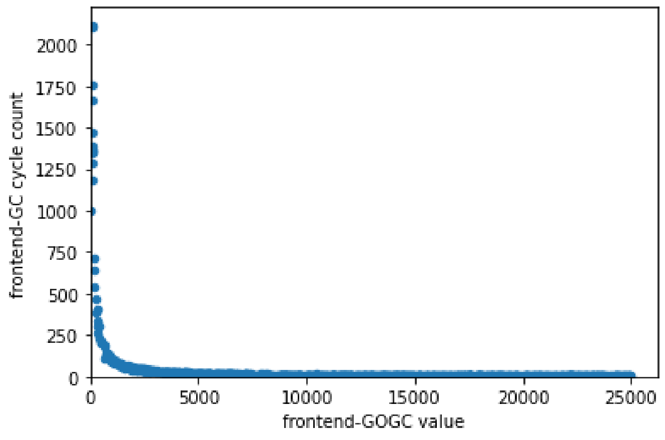

**Figure 4.15:** GOGC and number of GC cycles.

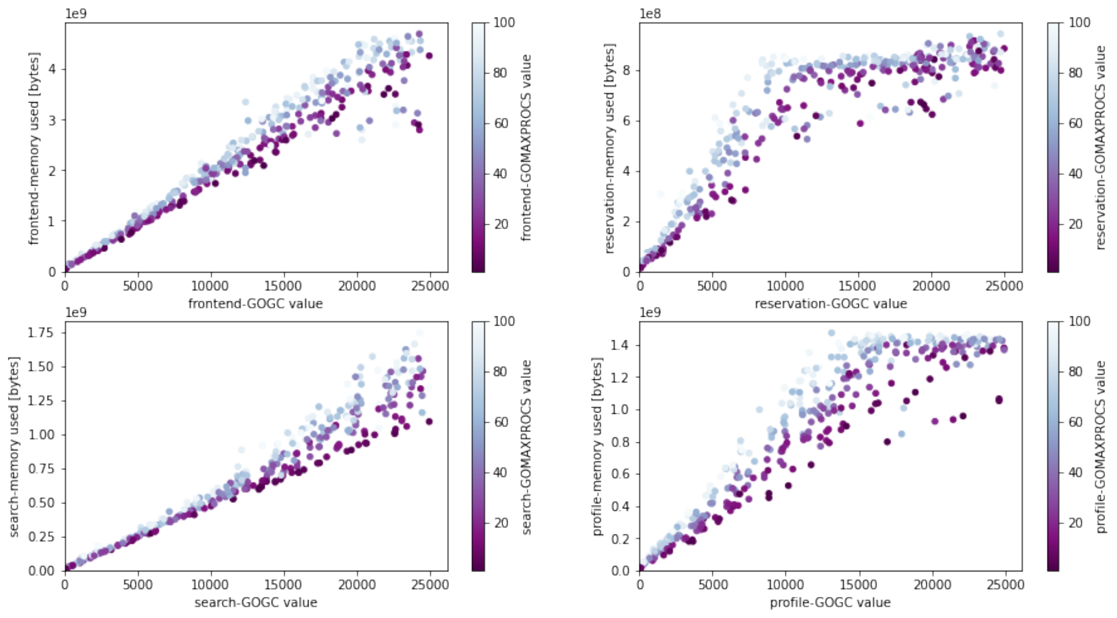

**Figure 4.16:** GOGC and memory utilization.

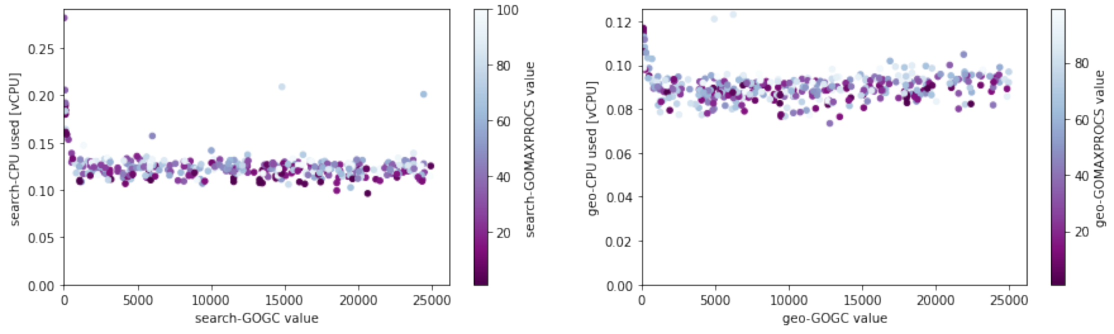

**Figure 4.17:** GOGC and CPU utilization.

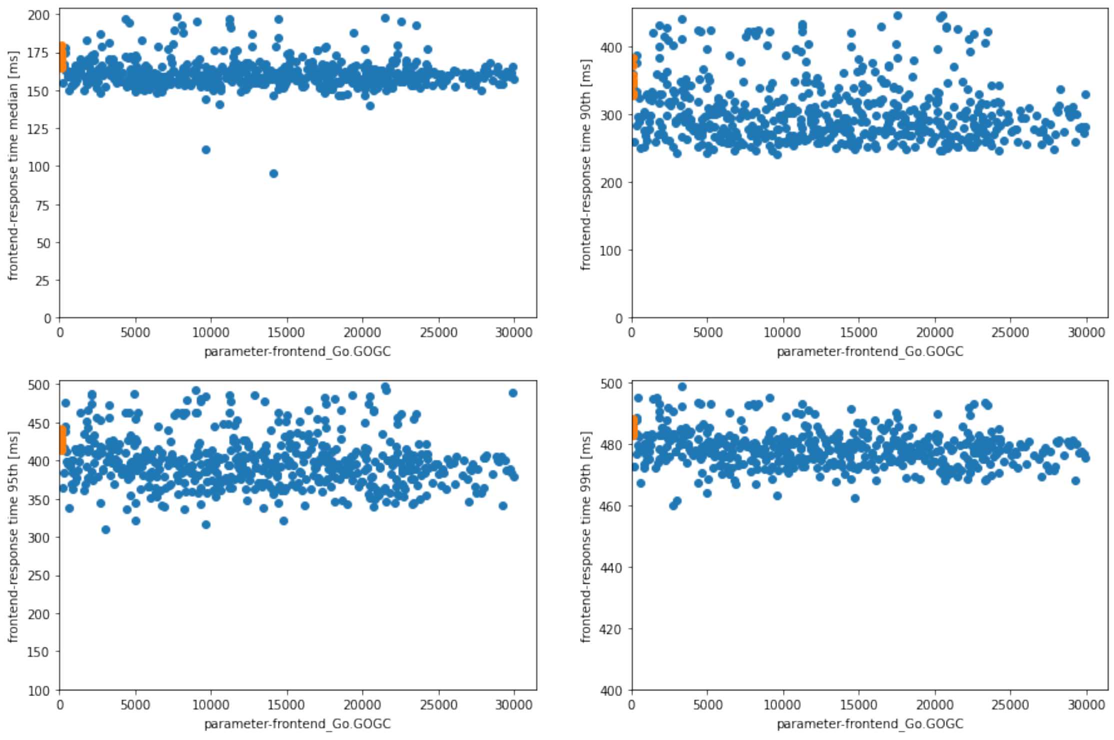

**Figure 4.18:** GOGC and response time.

allocations of new objects slow down until they almost stop. Another interesting thing to note is how the `GOMAXPROCS` parameter also affects memory usage, as also anticipated in 4.10. It can be seen from the colour scale in Figure 4.16 how as this parameter increases, the memory required by the service increases.
As far as CPU is concerned, on the other hand, it can be seen that the consumption of this resource decreases significantly. However, in some services, the decrease is smaller than in others (though noticeable), depending on the service under consideration.

Regarding system performance, we expected a decrease in response times at all percentiles. However, we noticed that this decrease does not seem to correlate linearly with the value of `GOGC` as we might have expected. Figure 4.18 shows that there is indeed a performance improvement compared to the baseline experiments (coloured orange), but that it does not seem to correlate with the value of `GOGC`.

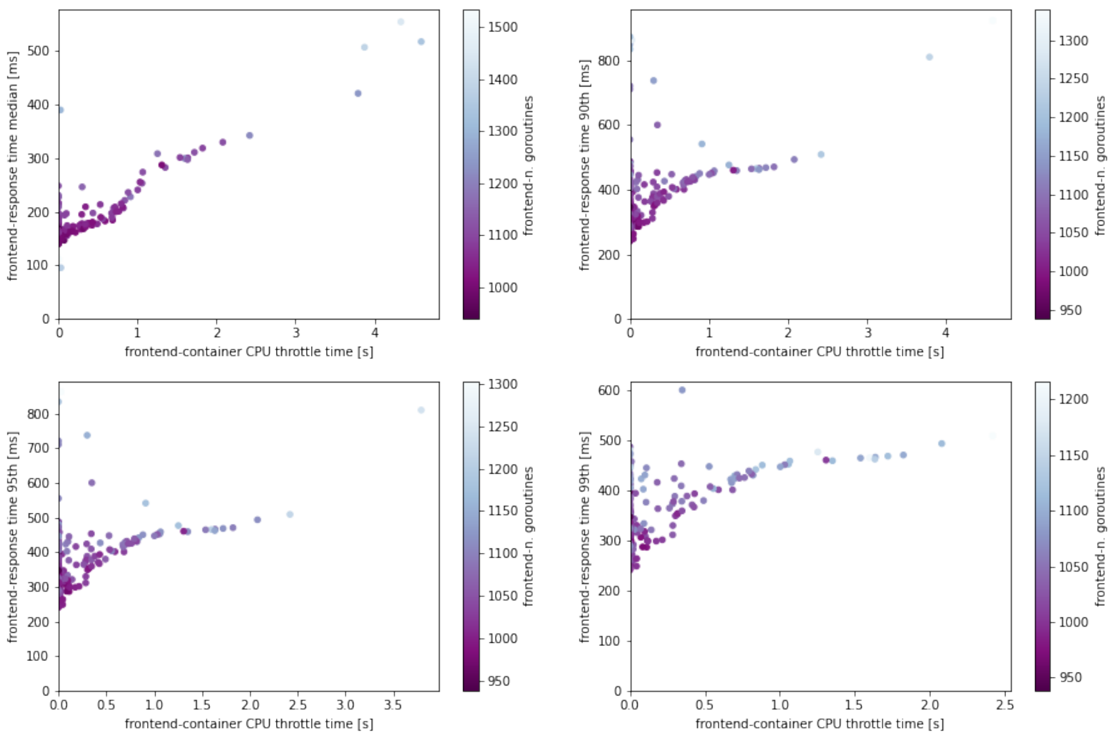

**Figure 4.19:** Throttling and response time.

It should be mentioned that, in this analysis, we considered the `GOGC` value of the frontend service only. In contrast, the response time related to this service is the "general" response time since, as already explained, traffic is redirected to the other services based on the request.

## 4.5 Throttle time

One of the parameters that most affect the response time of a service is undoubtedly the cpu_limit. Going to limit the periods for which a container can require the CPU. If this parameter is set too low, the service will not have enough computational resources to execute and will have to wait for the next period to continue, as explained in 3.3.2. Figure 4.19 shows how throttle time influences every percentile of the frontend service response time.

We can also see that higher values of throttle time correspond to a high value of service goroutines. As noted earlier, the most likely hypothesis for this behaviour is the creation of a queue of requests when the service fails to maintain a constant throughput, which then queues up while waiting for the CPU.
We can also observe the same behaviour by comparing CPU utilization with throttle time, as Figure 4.20 shows. This graph shows two very interesting things: how, already at a CPU utilization >0.5, Kubernetes starts throttling the container and how, at the same CPU utilization values but in different experiments, corresponds different throttling values. In this case, we have not been able to reach a comprehensive explanation, and the answer to these questions deserves additional work.

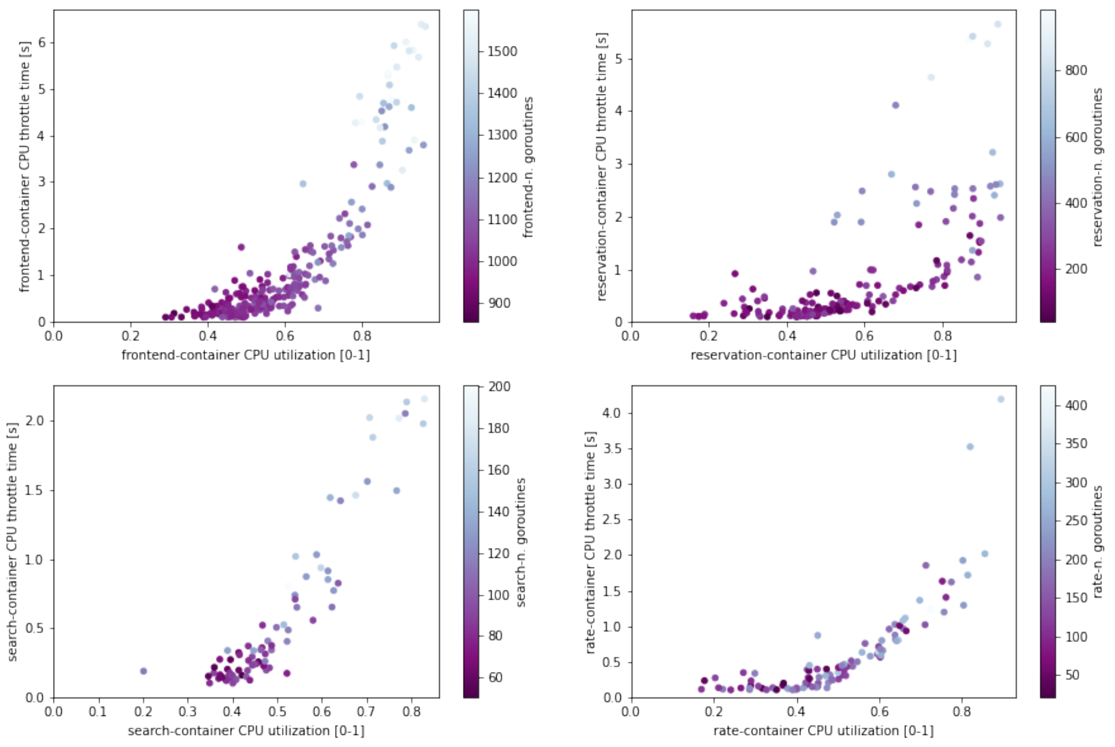

**Figure 4.20:** CPU utilization and throttling.

## 4.6 Resource usage and costs

As a final thing, we wanted to see how the overall resource utilization of the cluster had changed during the various experiments and try to budget the cost to run the application by implementing these parameter changes. From what we have seen and presented above, we might have expected an overall reduction in CPU utilization at the expense of memory utilization, which, as we have seen, is influenced mainly by the `GOGC` parameter. However, it is to be considered how the latter is a much cheaper resource than the former, and thus there is a more significant margin.
As Figure 4.21 shows, CPU utilization decreases heavily, up to percentage gain values (evaluated as resource savings) of ∼25% in two experiments. Only in a few cases resource utilization is greater than the baseline value, which corresponds to the first point in the graphs.

Regarding memory utilization, however, as Figure 4.22 shows, this is heavily negatively affected throughout the experiments by changing the various parameters. Indeed, it can be seen that the application uses approximately 2GB in the baseline conditions, while changing the parameters value, the average goes up to approximately 7Gb with peaks of 12GB. In absolute values, this is by no means an excessive consumption, but in relative terms, it is an increase of ∼250%. However, what is perhaps of more interest at the resource consumption level is how these translate into a cost to run the application. To make this estimate, we relied on the AWS Fargate service, which allows the use of on-demand containers and allocates

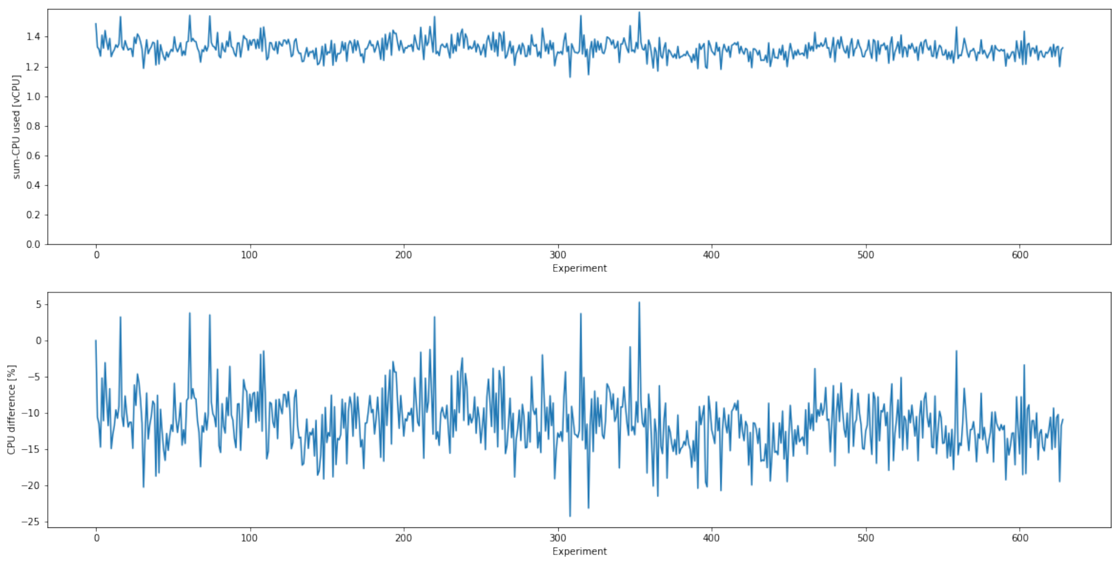

**Figure 4.21:** Cluster CPU utilization during experiments.

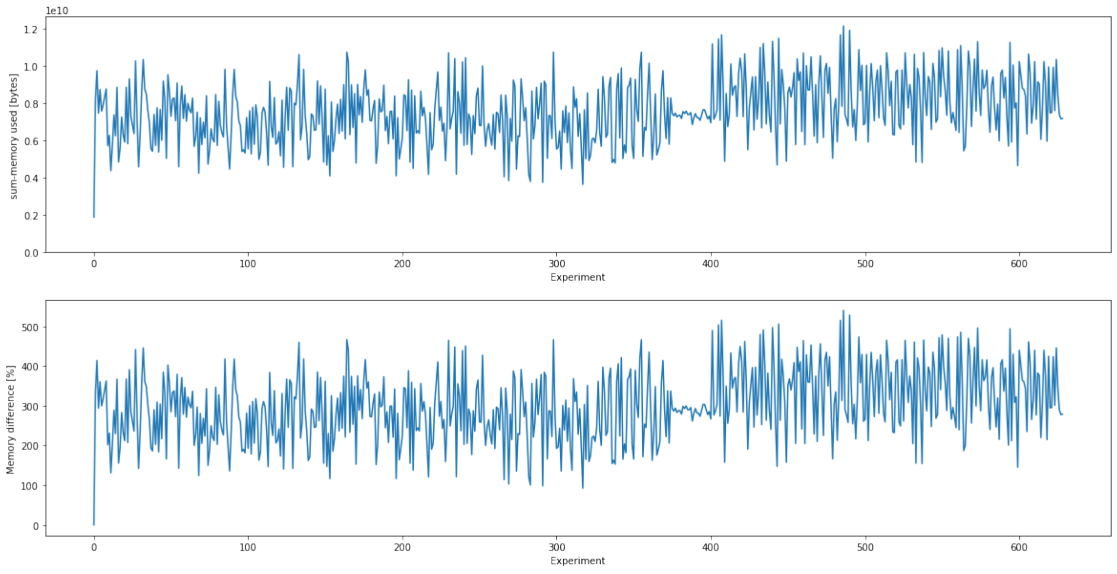

**Figure 4.22:** Cluster memory utilization during experiments.

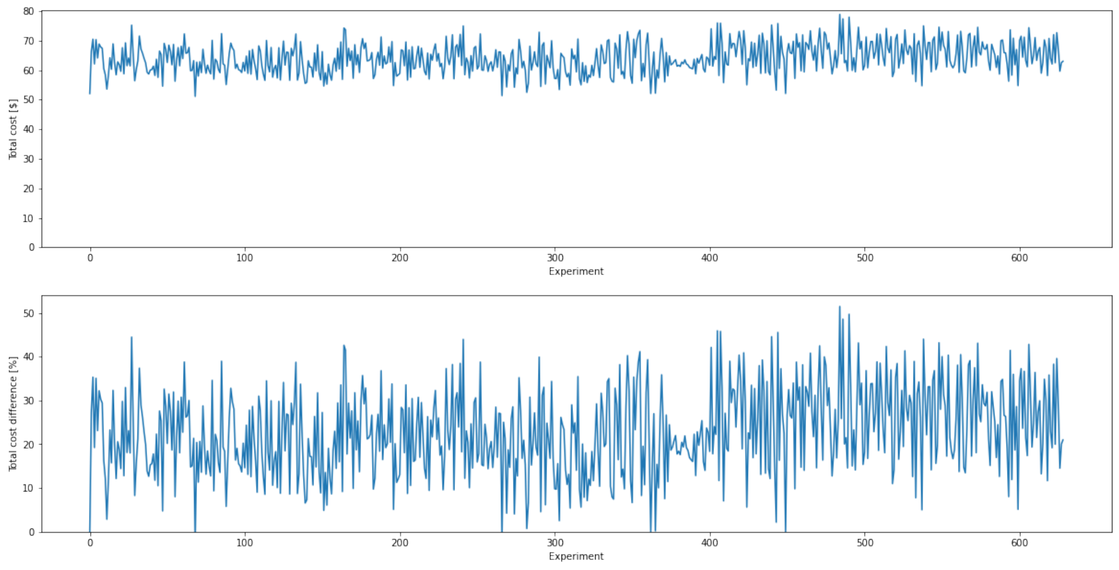

**Figure 4.23:** Estimate of AWS Fargate costs during experiments.

resources elastically. In our case, however, we only wanted a rough estimate and simplified the resource cost calculation even further. The AWS service is priced in increments of required resources, but we performed our calculations with a linear cost, i.e., as if we paid for the resources used without any margin of unused resources. In this way, we obtained the graph in Figure 4.23. As can be seen, despite the lower CPU resource utilization, the significantly higher memory utilization results in a higher overall cost of running the application.

After seeing how system performance and cost estimation change with the new configuration parameters, we wanted to understand which and how many experiments were in a specific range. Figure 4.24 shows the response time at the 90th percentile and the configuration cost of all experiments where the response times are better than the baseline, in all percentiles. Experiments that improve the 90th percentile response time >-20% and with an overall cost <20% are highlighted in orange. It can be seen that for these 71 experiments, there is no correlation between cost and response time. Therefore, it is interesting to note that the average value of `GOGC` is ∼12156, while for frontend service, which affects the total cost the most, it is ∼7811. If we then consider the only experiment whose total cost is less than the baseline, we have that the average value of `GOGC` is ∼8829, and the frontend service is 2525.

In the next chapter, we will also briefly address how this could be improved and, in general, how the study and optimization of Go could continue.

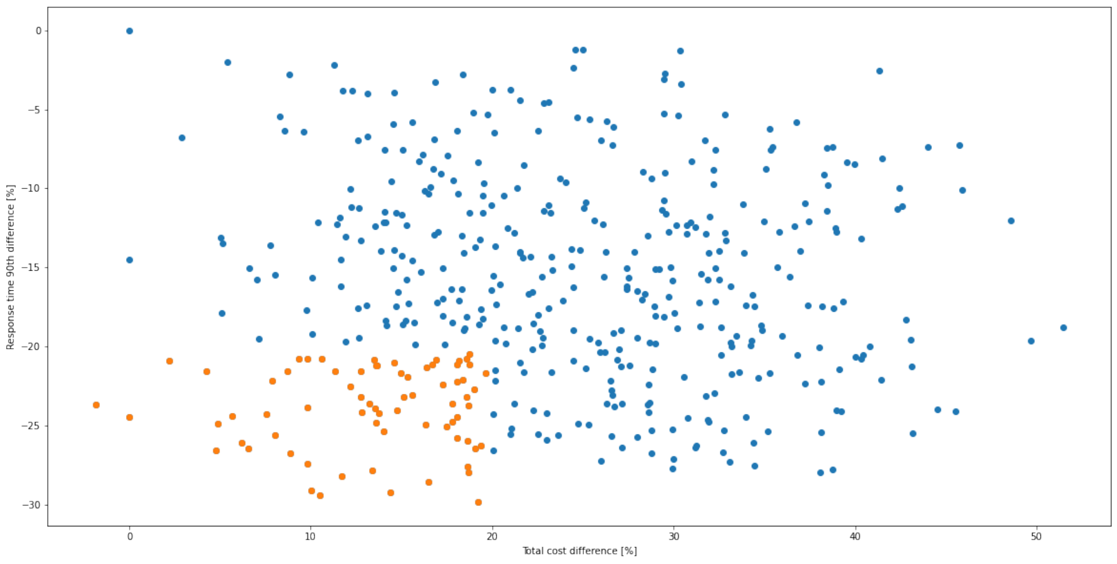

**Figure 4.24:** Summary of best experiments.

# Chapter 5

# Conclusion

In this final chapter, we want to summarize the results of this work, what could have been done differently and propose how this study could continue.

As we have seen, changing Go and Kubernetes parameters significantly impact resource utilization. We have seen how `GOGC` is the parameter that most influences runtime behaviour, and how this affects GC behaviour, despite the fact that there does not seem to be a direct correlation between parameter value and system response time. This, probably, is the aspect that raises the most questions related to the behaviour of this benchmark, and for this reason, it is the most interesting topic for future studies. As we saw in Section 4.4, it seems that the most significant positive influence of the parameter on the runtime behaviour of Go (at least for the benchmark tested in this work) is for values <5000, and as we saw in Section 4.6, the best results were obtained with a value of `GOGC` well below the maximum value of the domain set. For this reason, it would be interesting to focus on a new set of experiments with a maximum value of this parameter <10000. For example, if we plot the graph depicted in Figure 4.24 again but select only the experiments with a `GOGC` value <10000 for the frontend service, we derive the graph in Figure 5.1. We notice how a good portion of the experiments depicted in the left half of the graph are part of this subset, indicating that the response time improves and the cost (which can be view as an aggregation of CPU and memory usage) becomes much lower.

We have already mentioned how the configurable parameters in Go are much less than those in the Java JVM, especially concerning garbage collection and memory management. This may be since Go is intended to be a simple language, to be used as-is, but this does not allow pushing it to the limit. However, one has to consider the outstanding work done regarding garbage collector management, and the value of pause time during a GC cycle is minimal. Probably the less customization of Go's runtime is also due to the youth of this language and to the fact that most of the resources have been devoted to improving GC performance by thinking a priori about a model that could adapt to the major use cases, rather than a module that could be customized according to the specific application. Also, seeing Java as a more general-purpose language than Go, as the former is now used for a wide variety of applications, while the latter remains more limited to distributed environments and Web servers, contributes to the choices at the language development level.

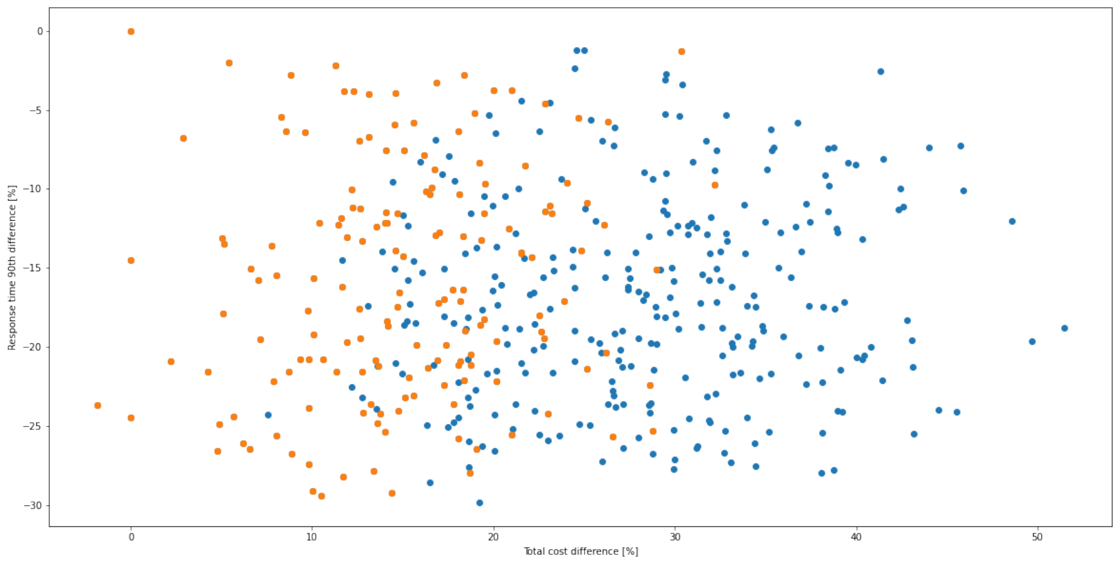

**Figure 5.1:** Summary of best experiments limiting frontend GOGC <10000.

An example of how, even from a performance analysis point of view, an individual must approach it differently depending on the technology in use can be seen from the results obtained: in Java, one of the first results that one goes to analyze is how the pause time of the GC influences the response time of the system, whereas in the case of Go there is an influence (as we have seen, as the value of `GOGC` increases, the minimum duration of pause time also increases, the duration at high percentiles of pause time decreases, and, in general, the sum of pause times and the number of cycles of GC 4.4), but not a direct correlation with response time. If we consider the entire technology stack in use and how the most influential metric on response time is the throttling that Kubernetes applies to services, we can see that an increase in the `GOGC` parameter can significantly influence this behaviour, as we have seen how fewer occurrences of the GC result in less CPU utilization. Thus more resources are available for the application core functionalities to execute before throttling occurs. As for the `GOMAXPROCS` parameter, there seems to be no particular improvement as its value changes. The official Go documentation and various online sources recommend setting it to the number of logical CPU of the machine on which the service is running, which is probably the best choice. In the case where the execution environment limits the CPU resource, whether at a basic level by cgroup or a higher level such as Kubernetes, it is interesting to see if this rule still applies. In any case, considering the first statement and the fact that the limit imposed on the resource is limiting for the value of available logical CPUs, one could run a new set of experiments by setting the upper bound to the domain of `GOMAXPROCS` equal to the value of logical processors in the system and see how it behaves.

One aspect not deeply analyzed in this study that is worthy of further study is the influence of Kubernetes' `CPU request` parameter. As explained in Section 3.3.2, this, in addition to defining the minimum value of computational resource required to execute, also defines the percentage division of CPU resource for each pod. While this under non-high load conditions, as in our case, does not affect

the program's behaviour, under high load conditions where there is contention for the CPU, it can be significant in helping the scheduler decide who can get the resource. We have seen that the parameter `CPU limit` imposes a hard limit on the amount of CPU a pod can use, which in heavy load conditions can be decisive on the scheduler's decision on whom to execute, but in the case of modest CPU load, where the pod would have the resources to be able to execute but is limited by this parameter, it can be counterproductive. There are conflicting views on the use of this parameter between those who say it is better not to set it and instead precisely define the `CPU request` for each pod, so that in the case of CPU contention it defines which one has priority to execute, and those who think it is a good way to define the limit of usable resources. For this reason, one of the continuations of this study could be the test of the benchmark with a higher system load to see how the system behaviour varies in the environment where only `CPU request` and where both `CPU request` and `CPU limit` are set.
Regarding the more general aspect of the work done, it would be interesting to re-run the experiments without using a service mesh. As we also saw in 3.4.3, these services heavily influence CPU utilization, and, as we wanted to maintain an overall CPU load ∼50% to avoid Hyper-Threading phenomena, it strongly influence the maximum load we could run on the benchmark. Recalling that the main reason we chose to use a service mesh was the need to have a time series of response times and not just an already processed aggregate, we could opt for a different load generator that would give more satisfactory data as output. In this case, we could not have the response times of the individual service but only of the frontend service, so less visibility into what is happening "behind the scenes", but we would also have the possibility to test the benchmark and Go runtime with a more sustained load and more like a real production environment. In addition, there are other possibilities to compensate for the observability that a service mesh can provide, for example, specific services for distributed tracing such as Jaeger[1].

[1] *Jaeger*. URL: `https://www.jaegertracing.io/`.

# Bibliography

## Articles

[1] A. Balalaie, A. Heydarnoori, P. Jamshid. “Migrating to cloud-native architectures using microservices: an experience report”. In: (2015).

[2] A. Muller and S. Wilson. “Virtualization with vmware esx server”. In: (2005) (cit. on p. 18).

[5] Amir Hossein Ashouri et al. “COBAYN: Compiler Autotuning Framework Using Bayesian Networks”. In: (2016).

[7] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes. “Borg, omega, and kubernetes”. In: (2016) (cit. on p. 23).

[8] Bohan Zhang et al. “A demonstration of the ottertune automatic database management system tuning service”. In: (2018).

[9] C. Anderson. “Docker [software engineering]”. In: (2015) (cit. on p. 22).

[10] C. Boettiger. “An introduction to docker for reproducible research”. In: (2015).

[11] Christina Delimitrou et al. “An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems”. In: (2019) (cit. on p. 3).

[13] D. Bernstein. “Containers and cloud: From lxc to docker to kubernetes”. In: (2014) (cit. on p. 20).

[14] D. Jaramillo, D.V. Nguyen, R. Smart. “Leveraging microservices architecture by using docker technology”. In: (2016) (cit. on p. 22).

[15] D. Ongaro, J. Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: (2014) (cit. on p. 26).

[16] D. Perez-Botero, J. Szefer, R.B. Lee. “Characterizing hypervisor vulnerabilities in cloud computing servers”. In: (2013) (cit. on p. 19).

[22] E. Casalicchio. “Container Orchestration: A Survey”. In: (2018) (cit. on p. 23).

[23] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan. “Security of os-level virtualization technologies”. In: (2014) (cit. on p. 20).

[24] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, E. F. M. Steffens. "On-the-Fly Garbage Collection: An Exercise in Cooperation". In: (1978) (cit. on p. 15).

[27] F. Bellard. "Qemu, a fast and portable dynamic translator". In: (2005) (cit. on p. 18).

[41] J. Sugerman, G. Venkitachalam, B.H. Lim. "Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor". In: (2001) (cit. on p. 18).

[43] Jason Ansel et al. "OpenTuner: An Extensible Framework for Program Autotuning". In: (2014).

[44] K.T. Seo, H.S. Hwang, I.Y. Moon, O.Y. Kwon, B.J. Kim. "Performance Comparison Analysis of Linux Container and Virtual Machine for Building Cloud". In: (2014) (cit. on p. 23).

[57] M. Calzarossa and G. Serazzi. "Workload characterization: a survey". In: (1993) (cit. on p. 9).

[58] M. Eder. "Hypervisor- vs Container-based Virtualization". In: (2016) (cit. on pp. 18–20).

[62] Omid Alipourfard et al. ""Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics". In: (2017).

[63] Omid Alipourfard et al. "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics". In: (2017).

[65] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. "Xen and the art of virtualization". In: (2003) (cit. on p. 18).

[66] P. Dash. "Oracle VM VirtualBox". In: (2013) (cit. on p. 18).

[67] P. Turner, B.B. Rao, N. Rao. "CPU bandwidth control for CFS". In: (2010) (cit. on p. 42).

[69] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, L. Smith. "Intel virtualization technology". In: (2005) (cit. on p. 17).

[70] R.P. Goldberg. "Architectural Principles for Virtual Computer Systems". In: (1973) (cit. on p. 18).

[71] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, A. Ghalsasi. "Cloud computing — the business perspective". In: (2011) (cit. on p. 17).

[73] T. Bui. "Analysis of docker security". In: (2015) (cit. on p. 22).

[81] Valentin Dalibard et al. "Building auto-tuners with structured bayesian optimization". In: (2017).

## Websites

[4] *About storage drivers*. URL: https://docs.docker.com/storage/storagedriver/#images-and-layers (cit. on p. 22).

[6] *Apache JMeter*. URL: https://jmeter.apache.org/ (cit. on p. 36).

[12] *Container-based virtualization architecture*. URL: https://canonical.com/blog/deploy-container-on-ubuntu-pro-on-google-cloud (cit. on p. 19).

[17] *DeathStarBench*. URL: https://github.com/delimitrou/DeathStarBench (cit. on p. 3).

[18] *DeathStarBench project*. URL: https://github.com/delimitrou/DeathStarBench (cit. on p. 33).

[19] *Docker*. URL: https://www.docker.com (cit. on p. 2).

[20] *Docker Engine architecture*. URL: https://code.visualstudio.com/learn/develop-cloud/containers (cit. on p. 21).

[21] *Docker overview*. URL: https://docs.docker.com/engine/docker-overview/ (cit. on p. 20).

[25] *Effective Go: goroutines*. URL: https://go.dev/doc/effective_go#goroutines (cit. on p. 11).

[26] *Evoy proxy project*. URL: https://www.envoyproxy.io/ (cit. on p. 30).

[28] *Gartner: AWS*. URL: https://www.gartner.com/doc/reprints?id=1-2710E4VR&ct=210802&st=sb (cit. on p. 37).

[29] *GC Pacer Redesign*. URL: https://go.googlesource.com/proposal/+/a216b56e743c5b6b300b3ef1673ee62684b5b63b/design/44167-gc-pacer-redesign.md#gc-pacer-redesign (cit. on p. 17).

[30] *Go*. URL: https://go.dev/ (cit. on p. 3).

[31] *Go 1.17*. URL: https://go.dev/doc/go1.17 (cit. on p. 35).

[32] *Go 1.9*. URL: https://go.dev/doc/go1.9 (cit. on p. 35).

[33] *Go schedule*. URL: https://riteeksrivastava.medium.com/a-complete-journey-with-goroutines-8472630c7f5c (cit. on p. 10).

[34] *Go workflow*. URL: https://speakerdeck.com/retervision/go-runtime-scheduler?slide=14 (cit. on p. 12).

[35] *Goroutine stack memory consumption reduction*. URL: https://go.dev/doc/go1.4#runtime (cit. on p. 11).

[36] *gRPC*. URL: https://grpc.io/ (cit. on p. 34).

[37] *High-level Kubernetes architecture*. URL: https://viveksahu26.medium.com/kubernetes-part-1-80e9539ed936 (cit. on pp. 24, 26, 27).

[38] *Hotel Reservation application*. URL: https://github.com/delimitrou/DeathStarBench/tree/master/hotelReservation (cit. on p. 34).

[39] *Hyper-Threading*. URL: https://www.intel.co.uk/content/www/uk/en/gaming/resources/hyper-threading.html (cit. on p. 47).

[40] *Istio architecture overview*. URL: https://www.businessprocessincubator.com/content/managing-the-kubernetes-and-istio-ecosystem/ (cit. on p. 29).

[42] *Jaeger*. URL: https://www.jaegertracing.io/ (cit. on p. 71).

[45] *Kubernetes*. URL: https://kubernetes.io/it/ (cit. on p. 3).

[46] *Kubernetes API server*. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/.

[47] *Kubernetes concepts*. URL: https://kubernetes.io/docs/concepts/ (cit. on p. 24).

[48] *Kubernetes control plane*. URL: https://kubernetes.io/docs/concepts/#kubernetes-control-plane (cit. on p. 25).

[49] *Kubernetes controller manager*. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/.

[50] *Kubernetes pods*. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod/ (cit. on p. 24).

[51] *Kubernetes replicaset*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/ (cit. on p. 25).

[52] *Kubernetes replication controller*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/ (cit. on p. 25).

[53] *Kubernetes services*. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (cit. on p. 25).

[54] *Kubernetes, viewing pods and nodes*. URL: https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/ (cit. on p. 24).

[56] *Locust*. URL: https://locust.io/ (cit. on p. 36).

[59] *Memcached*. URL: https://memcached.org/ (cit. on p. 3).

[60] *Microfocus LoadRunner*. URL: https://www.microfocus.com/en-us/products/loadrunner-professional/overview (cit. on p. 36).

[61] *MongoDB*. URL: https://www.mongodb.com/ (cit. on p. 3).

[64] *Online Boutique project*. URL: https://github.com/GoogleCloudPlatform/microservices-demodeathas (cit. on p. 33).

[68] *Prometheus*. URL: https://prometheus.io/ (cit. on p. 35).

[72] *Service mesh architecture overview.* URL: https://www.businessprocessincubator.com/content/managing-the-kubernetes-and-istio-ecosystem/ (cit. on p. 28).

[74] *TCMalloc : Thread-Caching Malloc.* URL: http://goog-perftools.sourceforge.net/doc/tcmalloc.html (cit. on p. 14).

[75] *The industry-leading container runtime.* URL: https://www.docker.com/products/container-runtime (cit. on pp. 20, 21).

[76] *Thread guard page.* URL: https://dave.cheney.net/2014/06/07/five-things-that-make-go-fast (cit. on p. 12).

[77] *Tracing Garbage Collection.* URL: https://en.wikipedia.org/wiki/Tracing_garbage_collection (cit. on p. 16).

[78] *Tricentis Neoload.* URL: https://www.tricentis.com/products/performance-testing-neoload/ (cit. on p. 36).

[79] *Types of hypervisor-based virtualization.* URL: https://shellcode.blog/Applied-Purple-Teaming-Series-P1/ (cit. on p. 18).

[80] *Understanding Kubernetes Architecture.* URL: https://geekflare.com/kubernetes-architecture/ (cit. on p. 24).

[82] *What is a container?* URL: https://www.docker.com/resources/what-container (cit. on p. 19).

[83] *wrk2.* URL: https://github.com/giltene/wrk2 (cit. on p. 36).

## References

[3] T. Velte A. Velte. *Microsoft virtualization with Hyper-V.* McGraw-Hill, 2009 (cit. on p. 18).

[55] David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide.* Cambridge University Press, 2005 (cit. on p. 9).