Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Leonardo Araujo Benicio dos Santos

Consul: Rails web application scalability and performance improvements with distributed computation

São Paulo
Dezembro de 2020

Consul: Rails web application scalability and performance improvements with distributed computation

Final monograph of the course
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Alfredo Goldman vel Lejbman
[Cosupervisor: Vanessa Mi Tonini]

São Paulo
Dezembro de 2020

Resumo

Este projeto tem o objetivo de estudar como uma aplicação em Ruby on Rails escala com um número crescente de usuários simultâneos acessando o serviço. Mais especificamente, vamos estudar o Consul e propor melhorias para diminuir o tempo de resposta e aumentar o número de usuários simultâneos atendidos.

Consul é um aplicativo de código aberto sob a GPLv3 desenvolvido pela prefeitura de Madrid. Seu objetivo é facilitar o processo de participação do cidadão no processo legislativo, debates e pesquisas. Mas quem é melhor para definir Consul do que ele mesmo? Segundo o site da Consul, “Consul é a mais completa ferramenta de participação cidadã para um governo aberto, transparente e democrático”. A parte incrível desse software é a comunidade ao seu redor, com mais de 35 países, 135 instituições e 90 milhões de cidadãos participando, sem falar nas mais de 1000 estrelas no GitHub, 700 forks e 15.000 commits.

Além dos desafios técnicos, o que me levou a este projeto é o seu contexto sob o conceito de cidades inteligentes. Seguindo a definição, "uma cidade inteligente é uma cidade que incorpora tecnologias de informação e comunicação (TIC) para melhorar a qualidade e o desempenho dos serviços urbanos, como energia, transporte e serviços públicos, para reduzir o consumo de recursos, o desperdício e os custos gerais". O Consul entra em ação ajudando os cidadãos a participarem mais do processo legislativo para ajudar as autoridades públicas a concentrar seus esforços onde está o problema.

Para entender o comportamento do aplicativo, usaremos um software de simulação de teste de carga chamado Gatling para simular cargas de usuário de um único usuário para um lote de usuários simultâneos durante intervalos de tempo fixados. O Consul será executado em um cluster simulado do Kubernetes por meio do Minikube, onde o cluster simulado variará CPU e RAM. A última incógnita que simularemos é o número de réplicas para cada tamanho de cluster.

Por fim, discutiremos alguns insights sobre como melhorar a confiabilidade e escalabilidade de uma aplicação Ruby on Rails, é claro, com foco em nosso cenário, que é o Consul. Para isso, faremos uma análise estatística detalhada de como esse aplicativo lida com usuários simultâneos e como pequenas melhorias poderiam aumentar a disponibilidade do software. Esperamos, também, convencer o leitor da importância dos testes de carga e falar sobre as vantagens que o Kubernetes nos traz.

Palavras-chave: testes de carga, consul, cidades inteligentes

Abstract

This project studies how a Ruby on Rails application reacts through an increasing number of simultaneous users accessing the service. More specifically, we are going to study Consul, and propose improvements to lower the response time and increase the number of supported simultaneous users.

Consul is an Open Source application under GPLv3 developed by the Madrid city hall. Its goal is to ease the process of citizen participation in the legislation process, debates, and polls. But who is better to define Consul then itself? According to Consul's website, "Consul is the most complete citizen participation tool for open transparent and democratic government". The amazing part of this software is the community around it, with over 35 countries, 135 institutions, and 90 million citizens participating, not to mention over 1000 stars on GitHub, 700 forks, and 15000 commits.

Aside from the technical challenges, what drove me towards this project is its context under the concept of smart cities. Sticking with the definition, "A smart city is a city that incorporates information and communication technologies (ICT) to enhance the quality and performance of urban services such as energy, transportation, and utilities to reduce resource consumption, wastage, and overall costs". Consul comes right into play helping citizens participate more in the legislation process to help public authorities focus their effort where the problem is.

To understand the application behavior we will use a load test simulation software called Gatling to simulate user loads from a single user to a batch of simultaneous users during fixed time intervals. Consul will be running on a Kubernetes simulated cluster through Minikube, which will range CPU and RAM. The last unknown that we will be simulating is the number of replicas for each cluster size.

Finally, we will discuss some insights on how to improve the reliability and scalability of a Ruby on Rails application, of course, focusing on our scenario, which is the Consul. To do so we will be going through a detailed statistical analysis of how this application handles simultaneous users and how minor improvements could increase the availability of the software. We hope, as well, to convince the reader of the importance of load tests and talk about the advantages Kubernetes bring us.

Keywords: load test, consul, smart city.

Contents

Chapter 1

Introduction

1.1 Motivation

It is commonplace to say that society, in general, has been increasing in life speed. By that, we mean the communication has been increasing in speed continuously. On the search to be more productive or because people got used to things going fast. No one expects to wait anymore. Bringing to this study context, people expect websites to load quickly. As we are going to talk about, there are some studies around it. We will show, as well, that Google still uses website load time to measure performance on their top ranking websites.

The reader might not be satisfied with just people expecting quicker page loads. Then, we shall explain why load time is a great measurement tool to calculate an application performance. On a web server, there is an enormous amount of variables we have to deal with. For example, we have network packets in and out, memory, disk space, disk read/write speeds, latency, process power. Using machine specifications like this is not a great idea because someone else cant measure the data as well, so, for example, an search engine can't track which web application performs better to rank it better.

Page load time shows us a perfect combination of all those variables summarized on one metric that matters the most for the final user. Also, eases the process for other machine processes the data. If your web applications take too long to answer a request, you could leave the other machine on the other endpoint waiting, wasting resources.

We will explore all those metrics and also talk about virtualization implications, benefits and problems. Discuss and understand how each metric affects our page load time. We hope to achieve a better understanding of how to deploy a Ruby on Rails application. Make better use of available hardware to minimize costs and increase the number of answered users and their respective response times.

1.2 Why should we care about load time?

This chapter will try to convince the reader about the importance of the website page load's time. Let's start with the big picture to give us a scenario from where we are departing. It has been a few years since the world web traffic has surpassed desktop access. Here we have to split the problem into two parts; first, mobile connection across the globe is not as good as it is in major cities; second, most mobile devices do not have a big processing power. To put this under perspective, imagine an average mobile user, under those circumstances, having to use your website. If he has to wait like 10 seconds each time he clicks on a link, how long do you think this same user will last before he gets frustrated and abandon the

task?

Continuing investigating the same scenario but from the perspective of companies. Now imagine, let's say, Google's crawler working on indexing websites. Imagine that there is some other website which loads in 100ms each page, by the time the bot has crawled 6 pages of yours, it will have crawled about 600 pages of that other website within the same amount of time, and, consequently, budget spent, which will get a higher SEO[1] (Seach Engine Optimization) score? not even our bots tolerate long page loads, jokes aside, from Google's crawler point of view, there is no sense of spending a lot of time crawling one website when it can craw other sites.

But wait, there is more. If your web page loads faster, your device, in general, will spend less time handling that request. With a small payload, it will have lower code to process; consequently, the users and your CPU will have spent fewer resources processing that request. Again, lowering your bills with hardware and, also, saving user's hardware as well, and making you more sustainable.

Until now, we only have talked about the technical benefits of lowering your page load time. Now we need to talk about the psychological point of view. If you deliver your content faster, numerous studies show that your website will feel more professional. Going back to our imaginary exercise, a slow website gives the impression of an old computer, which links with unmaintained and abandoned software.

Narrowing down to our case, Consul. Our users will be navigating through different legislations processes, voting on polls. There is a huge amount of text to load on each topic this user tries to interact with, maybe images, not to mention the comments and debates. If our goal is to keep this user engaged, debating, things have to load fast; it must load fast. You can't keep track of a conversation or exploring the different process going on the platform if each time you open a link, or do any ordinary action, it takes a considerable amount of time.

1.3 How fast is fast enough?

We think there is enough motivation to prove that increasing page loads is an important matter. But then comes another question, how fast is fast enough? And here, there is no convention, but we can investigate and get a pretty good approximation. For example, according to Google, they aim to page response time's under 500ms. But, before we start talking about metrics and goals, we need to define metrics.

Figure 1.1: *Blue Corona's survey on how long would a user would wait for a page load against Google metrics according to his study (ref. Corona).*

[1] https://developers.google.com/search/docs/beginner/seo-starter-guide

Anyone who has done web development knows which is the document ready event, which is when a web page is ready for the user. Still, we must clarify that, and to do so, we will use Philip Walton's article, "User-centric performance metrics" (ref. Walton). For example, a web site that loads a few lines of javascript code and then starts firing Ajax calls to load the web site may have a faster response time, but we won't study this case here when we talk about page load time, we mean the time to load all page content. And here we did the first separation of our target group; we are not interested in the perceived load speed, for example, loading pieces of the page at each time, so the user could start interacting with the page before it fully loads, which is a strategy to attenuate the user frustration using a slow service.

Having set a target response time, we should aim to talk about our user's scenario. For this purpose, again, talking about Google's ranks, the top ranking websites on Google gets page loads time of under 3000ms according with Philip Walton which wrote the artcile "User-centric performance metrics" (ref. Corona). But the reality is not as good as Google's top-ranking web sites. The Unbounce page speed report says that in 2019, the average web site load time was about 15000ms and as we can see in figure 1.1, 2% of the Google's crawled web sites answers their requests under 3000ms, 13% got under 5000ms, which is the recommended maximum response time over 3G.

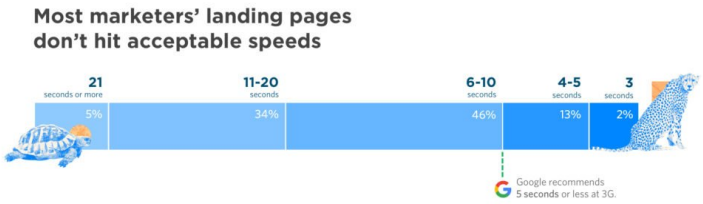


Figure 1.2: *Landing page response time for top Marketers at Google according to Blue Corona study (ref. Corona).*

Like everything else, there is always the other side, and the same report shows us that different users behave differently, for example, "Android users are more patient than iOS users. Of those who will wait 1-3 seconds for a page to load, 64% were iOS users while only 36% were Android users. Of those who said they'd wait 11-13 seconds, only 36% were iOS users versus 61% Android users." and they say as well that most of the users will, firstly, blame the internet provider of their equipment before blaming your web site. Still, we can sustain a platform based on a belief that someday will vanish.

	1 second	10 seconds	20 seconds
Lostness	.36	.59	.57
Frustration	2.6	2.6	1.6
Task Difficulty	3.4	3.1	2.8

Figure 1.3: *Lostness, Frustration, & Task Difficulty Means for Experiment 1 of Paula Selvidge (ref. Selvidge) study.*

We will be relying on a study by Paula Selvidge (ref. Selvidge) related to page load response time. In her experiments, she tried batches of users. The first batch was subject to response times of 1, 10, and 20 seconds. The second batch was submitted to 1, 30, and 60 seconds. The study's goals were to measure the degree of "lostness", task success, frustration,

Number of Participants	1 second	10 seconds	20 seconds
Successful	27	34	30
Not Successful	13	6	10
Total	40	40	40

Figure 1.4: *Task Success for Experiment 2 of Paula Selvidge (ref. Selvidge) study.*

	1 second	30 seconds	60 seconds
Lostness	.51	.50	.58
Frustration	1.9	2.8	2.8
Task Difficulty	2.3	3.0	2.8

Figure 1.5: *Lostness, Frustration, & Task Difficulty Means for Experiment 1 of Paula Selvidge (ref. Selvidge) study.*

and task difficulty as a function of delay time. To measure lostness, she got the ratio of the optimal number of nodes required to complete a task to the actual number of nodes visited by the user during the same task. The value presented on Figures 1.5 and 1.6 ranges from 0 to 1.00, where the closer it gets to 1, the less loss the user felt during the task.

Her study concluded that longer delays produce bigger frustration and that not only the frustration increases, but it increases faster than linearly. Another point that is worth mentioning is that the success rate on doing such a task did not vary according to the delay in response time, but, at the same time, users quit doing that task with a 60 seconds delay. The lostness was not affected by the increase in response time as well, but, surprisingly, her study says that the average user would tolerate a delay of about 20 seconds but no more than 30 or 60 seconds.

To cover the average user expectation, we will rely on a study from Scott Barber (ref. Barber) from PennState University, USA, that investigates how fast a website needs to be. He classifies the website performance into 3 groups, user psychology, system considerations, and usage considerations. We are interested in his conclusions about user expectations. As we have defined earlier, his study is interested in end-to-end response time, which the user perceives. What the study points out is that a "fast" web site is a page that loads under 3 seconds; a typical page would load in between 3 and 5 seconds, 5 to 8 seconds is considered a slow web site, at 8 and 15 seconds, the user starts to get frustrated, and above 15 seconds it is unacceptable.

We hope it is visible to the reader the importance (ref. Cruz) of page load time, and we had success in motivating you, using different points of view, to show that this study is of great importance. Also, by this point, we hope that it is also clear that we are aiming to provide a sub-second response time, trying to achieve 0.5 seconds, and would not tolerate a response time of over 3 seconds. So, our best-case scenario is handling 95% of our users under 0.5 response time.

	1 second	30 seconds	60 seconds
Success	30	26	27
Not Successful	12	14	15
Total	42	40*	42

Figure 1.6: *Task Success for Experiment 2 of Paula Selvidge (ref. Selvidge) study.*

Chapter 2

Background

2.1 What is Consul?

Who is better to explain what is Consul then Conul's official page itself? Quoting from their website, "Consul is the complete citizen participation tool for open, transparent and democratic government". The development started on 2015 July 15th; more technically, it is an open-source platform written in Ruby, based on Rails gem, which allows citizens to collaborate in legislation discussion, polls, debates, participate in budgets vote. As of June 2020, it has been implemented in 35 countries and over 135 institutions, allowing more than 90 million citizens to participate in public life through debates, polls and others structures. It has been actively developed at a GitHub repository, maintained by Madrid's town hall with over 100 contributors, 14000 commits, and 1000 stars on GitHub.

By default, the Consul team provides an Ansible script to deploy Consul to production servers. It's minimum system requirements for the production environment is a Ubuntu 16 OS with 32GB of RAM, a quad-core processor with 20GB of hard drive, and a Postgres database. For a staging environment, it is recommended a Ubuntu 16 with 16GB of RAM on a dual-core processor with 20GB of hard drive and a Postgres database. They also offer a Docker image for use with Docker, but the image is not ready for production or staging servers, as advised by their official documentation.

Simple plot of Consul directory structure:

```
/
├── app
│   ├── assets
│   ├── controllers
│   ├── helpers
│   ├── mailers
│   ├── models
│   └── views
├── bin
├── config
├── db
│   ├── dev_seeds.rb
│   └── seeds.rb
├── doc
├── lib
└── log
```

```
├── logs
├── public
├── scripts
│   └── entrypont.sh .2 spec
├── tmp
├── vendor
├── .codeclimate.yml
├── .coveralls.yml
├── .erblint.yml
├── .eslintrc.yml
├── .gitignore
├── .hound.yml
├── .mdlrc
├── .rspec
├── .rubocop.yml
├── .rubyversion
├── .scsslint.yml
├── .travis.yml
├── Capfile
├── CHANGELOG.md
├── CODE_OF_CONDUCT.md
├── CODE_OF_CONDUCT_ES.md
├── config.ru
├── CONTRIBUTING.md
├── CONTRIBUTING_ES.md
├── crowdin.yml
├── dashboard.yml
├── dockercompose.yml
├── Dockerfile
├── Gemfile
├── Gemfile.lock
├── Gemfile_custom
├── knapstack_rspec_report.json
├── LICENSEAGPLv3.txt
├── Rakefile
├── README.md
└── README_ES.md
```

If we take a closer look at the project structure, it is easy to see a classic monolithic Rails application. It relies on a simple Memcached[1] structure for local caching, but there is no implementation of a dedicated structured solution. We are going to use a dedicaded Redis[2] cluster, which will be required for one of the methods that we are going to focus on to increase the application performance. All essential files for application functioning are given an example file, such as the database connection, environment variables files, and mail connection.

Like almost all Rails applications, it uses Rubocop gem to track code style and code

[1] https://www.memcached.org/about
[2] https://redis.io/documentation

linters. Their official repository on GitHub uses TravisCI to execute unit tests, which are extensively applied. It is also given on the official repository the scripts for constructs, setup, and seed the database for development purposes. The project itself as the product is heavily focused on community input and discussion, having contributions worldwide raging for a great spectrum of use cases. Not to mention its extensive documentation of all its features. Still talking about the community around Consul, they provide a community forum based on an open-source project as well, named Discuss, where everybody can interact and solve the problem of their own related to Consul's implementation development. As you can see from the below plots extracted from GitHub insights, they are quite active.

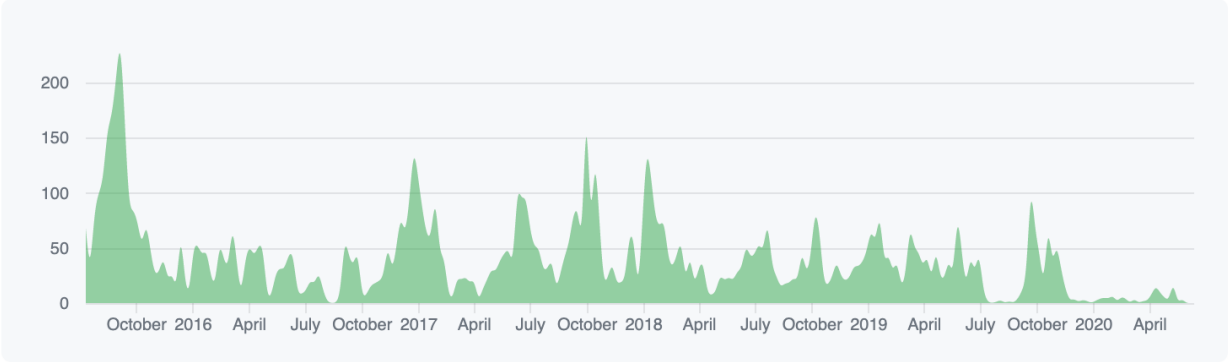


Figure 2.1: *Commits frequency of past year, excluding merges commits.*

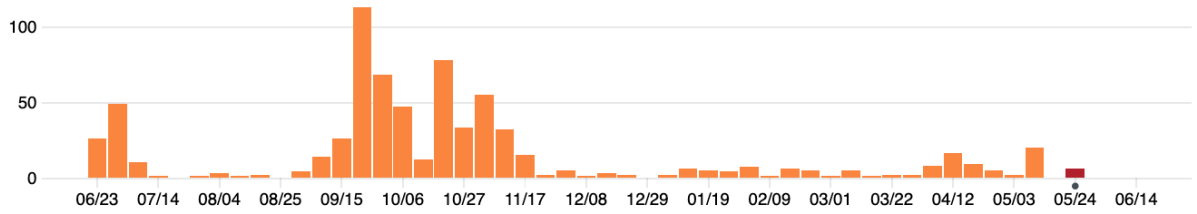


Figure 2.2: *Contributions to master.*

Consul is based on modulus, allowing third-party developers to attach pieces of code to solve problems related to their needs. By default, it allows users to interact with the application with a normal REST API and a GraphQL API. Still, only on read-only endpoints, it also provides default support for OAuth protocol. Finally, it provides a modulo to connect with the local town hall census without modifying the application code.

2.2 Why use containers?

Developing and deploying web application relies on different stages. Each of these stages depends on multiple libraries, depending on other runtimes and specific versions of each software. Configure all the dependencies and needed software to get the developer work environment running can be really hard, and reproduce this environment across a team of developers to make sure the development won't run into problems of matching dependencies versions and stack and even deploy this environment to testing, staging. The production server could be an even worse problem.

Quoting Solomon Hykes, creator of Docker, "You are going to test using Python 2.7, and then it is going to run on Python 3 in production, and something weird will happen. Or you will rely on the behavior of a certain version of an SSL library, and another one will be installed. You will run your tests on Debian, and production is on Red Hat, and all sorts of weird things happen". As the reader may already have discovered, developing software is increasing in complexity year by year, each time increasing the abstraction level.

As a community, we started using virtualization to easily set and reproduce the environment's configuration on which our software relies. Containers came into play just as the technology was getting attention and evolving, but that does not mean that virtualization became useless; each scenario needs a unique solution. Containers easy the job to apply those environments, assert that each piece of the software is the same across servers and developments laptops, making sure that the number of variables that could lead to a problem are reduced to the maximum possible.

Of course, we will lose some performance to increase our reproducibility, stability, security, and manage resource boundaries among applications. And talking about most of the web applications, that is exactly what it needs. Running an application on a real machine will always be faster than running over virtualization software such as KVM (Kernel-base Virtual Machine) and even faster than on a container. It is not difficult to find people using a container solution inside a virtualized system based on KVM. In further chapters of this text, we will show that the performance decreasing factor is acceptable compared to virtualization benefits.

Starting from the simpler of the benefits, scalability. Supposing, you are using a monolithic application and want to increase your availability to handle more simultaneous connections. The obvious solution is increasing your server's brute force vertically by increasing its attributes such as disk, RAM, or CPU cores. The other option is to increase it horizontally, add more servers to work in parallel to handle the increase in traffic. With this horizontal scale, you have to manage how each server communicates with other services and handle the requests. Here, the container orchestrator comes into play to do this job for you to increase using the second method.

Following Solomon Hykes, creator of Docker, as we already mentioned before, we need to make sure three different levels of reproducibility. First, we need that all member of the software developing team is running the same run times and dependencies levels, so the same code will behave the same way on different computers, like what we see in java, with the JVM (Java Virtual Machine), which allows it to run independently of the computer since it runs the JVM. Then, we need harmony between those developing environments and the testing environment, the staging server, and production ones for the same examples given before.

Talking about resource boundaries, we can see from the Figure 2.3 how software deployment evolves between the beginning of web applications and today. The production deploy architecture used to be all applications on the same server, which competes with each other for the resources. If a malfunction of one API, for example, used all the resources available on our server, the entire server would be compromised. And, entire system virtualization through Virtual machine services such as VMWare would increase the process's overhead.

Security is even easier to justify. If your application is containerized and separated into microservices and one of them is compromised, just this one will be compromised. If you are running on a monolithic application and running directly on the server, gaining access to part of your software could result in a whole server break down, compromising user security, and, even, data loss. The containers are isolated, sandboxed from the host OS (operating system), virtualized CPU, memory storage, and network resources at an OS level.

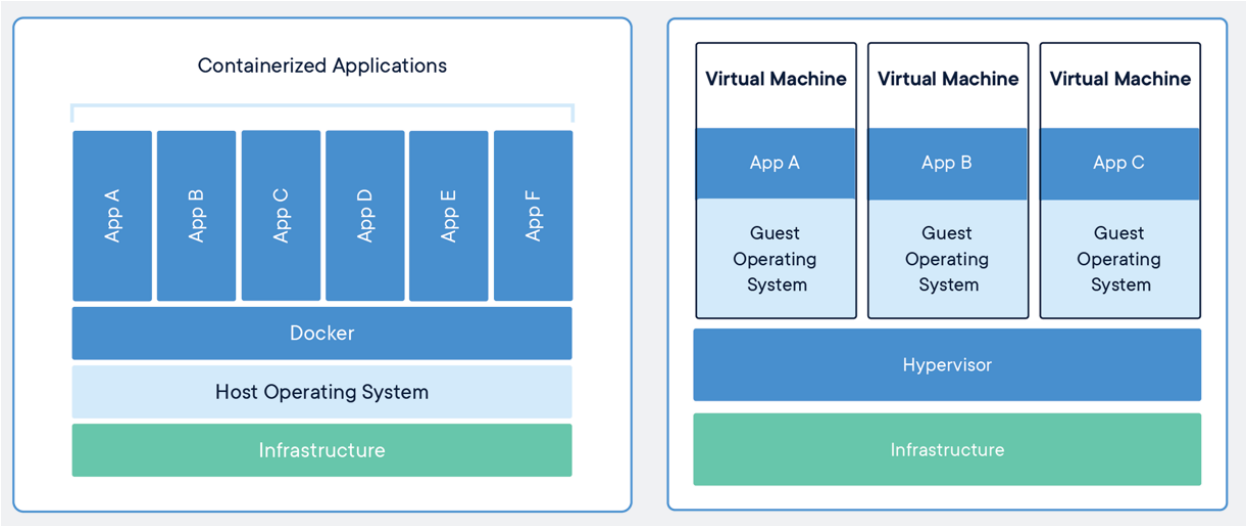


Figure 2.3: *Docker deploy architecture x Virtual Machine deploy architecture.*

2.3 Containers software

We had quite a few solutions from the beginning of the virtualization, such as Linux VServer, Solaris Containers, Open VZ, Process Containers, LinuX Containers, Warden Let Me Contain That For You, and last but not least, our industry-standard as of 2020, Docker. Most of the solutions mentioned and created before Docker, in 2013, were discontinued, and docker exploded in popularity and community support. Kubernetes, as known as "k8s," (ref. Mercl, and Pavlik) has over 2500 contributors, 65000 stars, and 24000 forks on GitHub, its main repository.

Docker and Kubernetes became the standard; the community is enormous, the variety of examples, people discussing them, and developing using one of those technologies are wide. That is our primary goal, we need to rely on a technology that is largely supported and well tested and capable of delivering what promises is; like Javascript, it always delivers what promises are. New promising technologies are emerging as of 2020, and the industry seems to be evolving again. Docker and Kubernetes are vastly supported by the community and has already proven to be capable of handling enterprise-level jobs, which all those other emerging software needs to prove yet.

As of 2014, Docker switched its virtualization library to one of its own, called "libcontainer" and quoting Hykes, again, from its release post on Docker blog, "First, we are introducing an execution driver API which can be used to customize the execution environment surrounding each container. This allows Docker to take advantage of the numerous isolation tools available, each with their particular tradeoffs and install base: OpenVZ, systemd-nspawn, libvirt-lxc, libvirt-sandbox, qemu/kvm, BSD Jails, Solaris Zones, and even good old chroot. This is in addition to LXC, which will continue to be available as a drive of its own." The libcontainer is a pure Go Library that accesses the kernel's container API's directly.

2.4 What is Kubernetes

From this point, we already understand where Docker comes to play at our development stack and what its job is, but where Kubernetes (ref. Barua) go in this scenario? And who is better to explain itself then Kubernetes? Quoting from Kubernetes documentation: "Kuber-

netes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.". From now we are one step further from understanding the big picture. Docker comes into containerizing our application, and Kubernetes is a software to manage these containers.

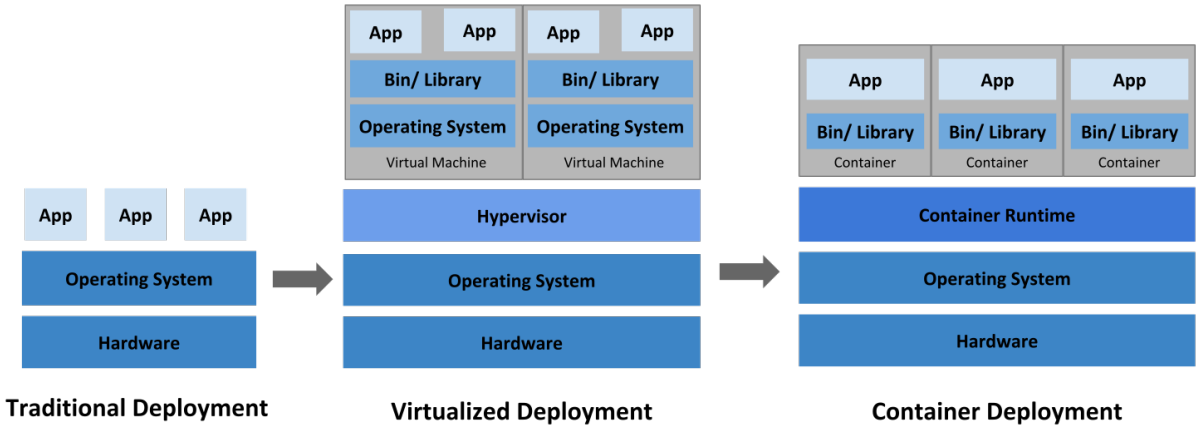


Figure 2.4: *Web application deployment evolution.*

Kubernetes emerges in a scenario where the need to handle containers failovers, downtimes, scaling, load balancing, storage orchestration, automated rollouts[3], bin packing[4], self-healing, secret and configuration management, and much more. In our study case, Kubernetes will load, balance all traffic, and distribute it across container replicas on different physical servers, expose our DNS name, assuring a stable environment on each server. Furthermore, storage orchestration is of the same importance for us since it will handle mounting different storage providers to handle our application needs. Taking advantage of the automated rollouts is essential to guarantee no downtimes during updates or rollbacks due to malfunctions of compromised updates; if we can deploy containers at the desired rate, checking their healthy at every step,, we can ensure that a broken container will not go into production. And, of course, bin packing allows us to tell Kubernetes the desired amount of CPU and RAM power each container will have at its disposal.

[3]https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a-deployment
[4]https://kubernetes.io/docs/concepts/scheduling-eviction/resource-bin-packing/

Chapter 3

Methodology and Experiments

3.1 What do we hope to achieve?

Our goal in this study is to identify scenarios where a Ruby on Rails application underperforms and discoverer what is causing such performance, thus leading us to investigate the software that gives some insights on building a better web application. Using the RAM and CPU allocation over Minikube's cluster, we can tackle our software's raw computation power at its disposal. Then we still need to understand how the replica set interferes with the application performance. For Example, Is it better to have four replicas to handle 1000 users, each one handling 250 users, or just 1 replica handling all? Does this make any difference at all?

From the other side, there is the total amount of simultaneous users and the time that those users will access our simulated cluster. Ranging these two variables, we can stress our application to discoverer how our hardware answers an increase in load power. The principle is simple: we start with small batches of users at small-time intervals, then we increase those numbers until we notice our cluster deterioration, signaling that we are starting to throttle the machine.

By simulating all those unknowns, we hope to understand how the Consul responds to user loads. With that in mind, we will explore possibilities on how to increase performance, allowing us to handle more users with the same hardware. But this analysis is not exclusively about Consul itself. It is about how a ruby web application escalates under a cluster environment and what the reader might be forgetting that we will analyze the replica set and understand how these applications behave in a cluster environment.

Let us dive into some examples to elaborate on what we hope to achieve by collecting these data. Imagine a scenario where 100 simultaneous users access the application for about 60 seconds with a simulated cluster size of 2 cores and 4096MB RAM. In that scenario, we can handle 90% of the requests. Now let us further increase our cluster size to 8 cores and 26000MB RAM. What is the expected behavior for our application? If with small amounts of computation power, we almost handled 100% of those 100 simultaneous users, with four times process power and 4.5 times RAM power. The job will be easy. You might have thought. But, as you may expect, the answer is not simple, and this is one of the scenarios that we explore and analyze during this study.

We are trying to demonstrate here that we can automate those steps and incorporate that test scope into our CI (Continuous Integration)[1] / CD[2] (Continuous Deploy) pipelines like what is proposed by Niklas Sundbaum (ref. Sundbaum) in his article where he defends

[1] https://www.atlassian.com/continuous-delivery/continuous-integration
[2] https://www.atlassian.com/continuous-delivery/continuous-deployment

that all software should incorporate those kinds of software to their pipelines.

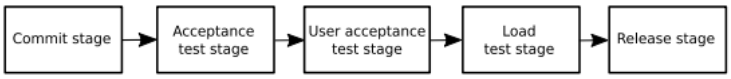


Figure 3.1: *Pipeline from Niklas Sundbaum (ref. Sundbaum) research.*

Further investigating his ideas, he proposes to investigate a way to deploy those tests, implement it over CI so it shows a clear and simple result, like passed and not passed, so that it can be automated into test pipes. And the proposed idea to evaluate load test performance is Control Charts like the example from his research in the figure below. The basic idea is to evaluate those chart regions to identify where the software performance deteriorates with proposed changes on an given commit, for example.

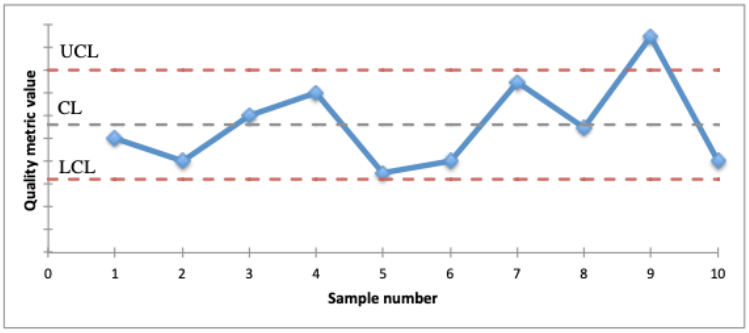


Figure 3.2: *Example of Control Chart pipeline from Niklas Sundbaum (ref. Sundbaum) research.*

3.2 Types of tests

According to S. Pradeep1, and Yogesh Kumar Sharma (ref. Pradeep, and Sharma) we can distribute the testing we apply to software to evaluate its capabilities before its launch across different test types. Those types are Load, Stress, Security, Smoke, Unit, Acceptance, Graphical user interface, Gorilla, and Performance testing. We are going to focus on the first two testing strategies.

The former, Load Test, which, again, accordingly to Pradeep and Yogesh is: "Evaluation of behavior of software with access by the huge load of users concurrently. The capacity of software is analyzed to handle the load of users" and, the latter, Stress test, "Analysis of the robustness of the software. It identifies the specific points where the software modules getting issues and evaluation under extreme conditions of system failure".

Looking for a different definition, we have the one from Andrei Proskurin (ref. Proskurin), which stands for Load test as "The system under test is exposed to the target load to verify that the performance targets are met under a production-like load." and Stress test as "The load on the system under test is ramped up until the application or supporting infrastructure breaks. The purpose of this test is to determine the capacity threshold for the system."

3.3 Example of workloads

Our goal here is to motivate the reader with a different projects and workloads implementation and understand how the test tools works on real-world scenarios and controlled

environment scenarios, going from software CI/CD pipelines to enterprise level applications witch requires to support a heavy load.

If we look at Andrei Proskurin (ref. Proskurin) research, his goal is to implement a loading test suit on a business level application developed using the Java Spring framework, which has industry-level standards. As his goals were to cover a large codebase, his main metrics were easiness to learn and implement these tools across the code, and the time it would be used to cover it all. His software stack was composed of Tomcat (application server), Oracle (database).

Another good example is the study conduct by Solange Paz1 and Jorge Bernardino (ref. Paz1, and Bernardino) where their goal is to evaluate different software testing tools and find what is the best one, but what is interesting is that to test the test tools, you actually have to push the tools into their limits so we can get a pretty good idea on how the would behave on heavy workloads on real environments.

A different perspective study is the Load Testing of Web Sites Daniel A. Menascé (ref.Menascé) explains how a website is built and what affects its performance the most. Also, he explains from a different point of view why implement load test, keep track of when to use it and implement it over CI/CD is so important. He also explains how load tests measure performance and how this relation, betwen load tests and performance improvements, stands.

Still following the idea of standardizing a load tests suite into CI/CD, we follow the study from Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora (ref. Zhen Ming Jiang, and Flora), Automatic Identification of Load Testing Problems, which talks about implementing a test suite into an enterprise-level application (Dell DVD Store, the JPetStore application), and uses Open Source software suites. Another interesting study from the same author is the Automated Performance Analysis of Load Tests (ref. Jiang *et al.*). But in this study, they try to narrow and solve common problems of an application under load by running software that analyses the application logs searching for bottlenecks.

Distributed functional and load tests for web services by Ina Schieferdecker1, George Din, and Dimitrios Apostolidis (ref. Ina Schieferdecker1, and Apostolidis) also uses an enterprise-level application as the subject of study to analyze and present a software testing tool and also discuss the importance of load testing your software. Still, at this time, the difference here is that the study analyses an older application with a heavy legacy code base and test styles to evaluate how it has evolved during the period.

Finally, the paper published by Rijwan Khana and Mohd Amjad (ref. Khan, and Amjad) tries to accomplish the same goal of this chapter, show the reader the importance of load testing web applications how performance is important to the end-user. The main interesting part of this study is that it uses real-world data from the production environment taken from its peak usage to measure the measurement tools and argue why it is important to test with real-world examples.

We are going to built a system for testing based on pieces from each research we mentioned earlier. We are going to use a combination of tools and scripts to develop a test approach that is usable on a CI/CD environment but still can be reproducible on the developer machine. To understand the application's throats we will use scripts in bash and NodeJS to create various scenarios to load int our test application, that will simulate the load. We will further discuss this topic on the next sections.

3.4 Tests scripts

We have already decided on the software we will test the Consul application, but we still need some code. First, the test script simulator, a Scala script, which controls the actions that Gatling[3] software will execute to simulate the load we are trying to test. We will not spend too much time on the test script, as it was auto generated by the record script that is part of the Gatling software with just simple page access. To get a starting point and simulate simple user navigation, the only exception is that we switched the fixed number of users and simulation duration to accept input from command line variables to automate the test process easily.

Simple plot of Consul directory structure:

/
- $test.sh$
- $tests_gatling.sh$
- $parse_results.js$

Our start script prepares the environment and launch the main script, which will perform the tests itself. At first, it is possible to see the launcher script prompts to get the sudo password, and that is because Minikube's tunneling daemon requires sudoers privileges. We need a self-sustainable script that could run on the cloud for a long period without prompting to ask sudoers password every couple of hours.

The main script is controlled by six main variables. The first one is $TRIALS$, which controls how many times Gatling will repeat the same test; *uamount* controls the number of simultaneous users each batch of tests will simulate. Similarly, *tamount* controls the user access duration, for example, if $tamount = 60$ and $uamount = 1000$, Gatling will simulate a batch of 1000 simultaneous users access the application during a period of 60 seconds. The second group is *ramount* and *camount*, which controls the number of replicas of the application's containers and the cores witch Minikube will use to simulate a Kubernetes cluster. For the least, the *mamount* controls how much memory will be available on our Minikube's simulated cluster.

First, we set the number of cores and memory we want Minikube to runs on, which will be used by Minikube to emulate each node on the cluster; that is, each core on the server will act as a node on our emulated cluster. We restart it to apply the new configurations. The next step is to patch the Minikube cluster IP to the Kubernetes interface, which helps the interaction between Minikube and Kubernetes simulate a load balancer service on the cluster and connect the outside world, not only inside the cluster itself.

The next step is to switch from Docker single machine environment to Minikube's environment, build Consul's image, and apply our configuration file $k8s - config.yml$. Now that we already have the environment configured, our tunnel to the external world setup, we jump into our "main" loop. First, we need to replace the newly generated cluster external IP address for this session with the Gatling scala script that we created before, which is done with the auxiliary of the *sed* command we can see on. Finally, launch the Gatling test application itself with our testing parameters and repeat it $TRIALS$ times. But before our script start Gatling itself, we need a little sleep time to allow Minikube to launch all the replicas and the containers, as we are using a single machine simulating a cluster; on launch time, we get a few $CrashLoopBackOff$ because all replicas on all cluster nodes are trying

[3] https://gatling.io/open-source/

to run its migrations at the same time thus the machine can't handle the job well, so we let it sleeps for 30 minutes to allow all its containers to crash and restore itself until its health reaches 100% during this time, which means all containers were able to run its migrations and start Rails application successfully.

Our main execution flow is divided into two sections, one for testing the Kubernetes auto-scale, as seen on, which we can tweak the settings for better performance and will be discussed in further chapters. The other execution flow is with fixed replicas amount, as shown in the figure, which increases exponentially until double the machine cores to avoid throttle our machine on the tests.

Our main problem was Minikube constant crash during the trials. As we perform a series of tests witch tweaks deferments states configuration for our application, for example, RAM, CPU cores, replicas set, after a certain amount of time advanced on the test set, Minikube stop responding, so at each interaction and after each trial we have to make sure Minikube is still running, everything is working as expected.

3.5 Parser script

After all the tests, we got a lot of result summaries. It is time to use the other auxiliary script, a NodeJS application, to parse all that result into a single CSV file of then summarized and sanitized to easy the analysis. The script itself is a javascript app running on NodeJS that opens all the folders on the tests directory and iterate through the files, extract the relevant data from it, and compile on a single line on the spreadsheet. By the time the script finishes its job, we will have a complete spreadsheet with all our test results summarized. The next step will be to load this CSV file into a Jupyter Notebook to take a closer look at the data generated.

The code is based on the main function, which transforms the raw data into structured data and calls two other parse functions to handle each appropriate file format, a javascript file containing statistics information and the index.html file.

Chapter 4

Load Test Software and tools

4.1 Technical features

Our main goal in this study is to analyze and increase cluster performance for handling multiple users concurrently, We are going to focus on loading times and requests handled. It is crucial that we know ahead of time how many users we can handle and the related amount of physical server we need to deploy to handle big cities such as São Paulo. We will discuss a couple of load testing software here and why We are using Gatling to do the job. We will focus on three main characteristics of the software, developer experience, performance, and maintainability.

We need software that is easy to use on the command line for single tests, scriptable; it needs to be executed on CI (continuous Integration) pipeline, it must be open-source and perform well. There are plenty of candidates to do the job, but We are going to focus on 5 candidates that demonstrate the good and bad qualities of each group of features. Our focus candidates are JMeter, Gatling, Locust, Drill, k6s.

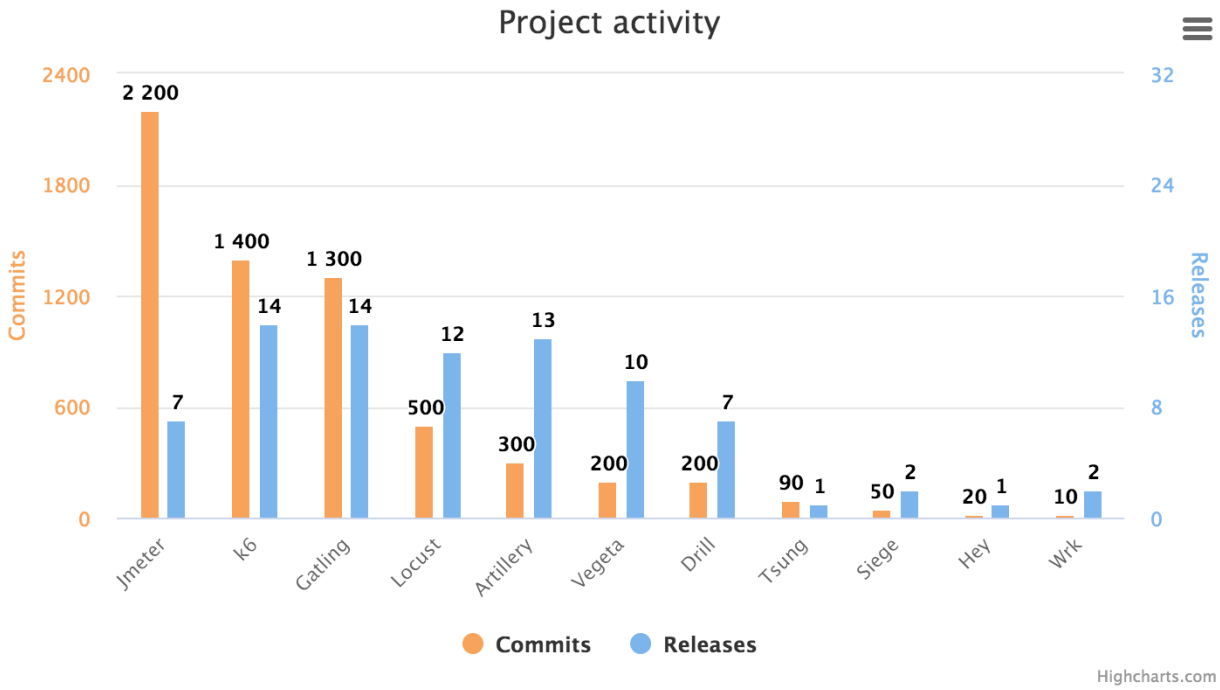


Figure 4.1: *Commits and releases on main repositories from Ragnar Lönn study (ref. Lönn).*

Starting with active development. The most important feature an open-source software would have is to be actively developed and have an engaged contributing community, which is also marked as a prominent soft load test tool by (ref. Pradeep, and Sharma) S. Pradeep and Yogesh Kumar Sharma. If we rely on software, we can not afoot it to be poorly developed, to analyze it we gonna check the commits and releases our these tools repositories. As you can see from the chart below, most of the market software is not as active as the first quartile. JMeter (ref. Doyle) witch is a giant of the industry, actively developed by Apache, a great lead followed by k6s, and Gatling is our best bet to rely on until here. If we look at the other candidates, we have Drill and Locust, but they do not look so actively maintained as the other, which is pretty critical.

Our first candidate is JMeter, but besides, it is actively developed, and Apache behind it is a good sign too; it is an old Java application. At the same time that Java is great and reliable, it is an old application, and to start a new application and use such old technology would not be the best choice; we could take advantages of the JVM (Java Virtual Machine) using newer technologies, and we don't too advanced tools to this study. On the other hand, following adversaries seems to be pretty good for the job. Both of them are twice actively developed as their following adversaries. K6s relies on AGPLv3, which is great, and Apache 2.0, which is the Gatling license. On the other hand, Locust goes with an MIT license, which is not a good sign for our project, and he wants to keep it as open as possible. Shining at the top, besides k6s, there is Drill, AGPLv3 as well (ref. Lönn).

Talking about coding, which one of them offers better possibilities using their environments? Suppose we start again from JMeter (ref. Loisel). In that case, it is, as mentioned before, a Java application, and a big one, which here is a score down because it would require more resources to run and test our ecosystem without taking advantage of its advanced capabilities. It is not scriptable, which is pretty bad as well. K6s here goes as our best options, as written in Go, a pretty performance-efficient programming language and runtime and scriptable in JavaScript, which is great too because it is such a versatile language. Last but not least, we have Gatling, which is written in Scala and scriptable in scala as well, which is a pretty downside of it since scala does not have a big community, but it relies on the JVM as well, and it is not as big as JMeter, which makes it very efficient. Besides being written in Rust, Drill is not scriptable, taking away too many possibilities of the table. Locust takes another hit here; it is written in Python and scriptable in Python. Besides all the amazing features python has, it is too slow and could throat our tests with bigger simultaneous user simulations.

Looking for a different approach, we see that Fernando, Monserrate, and Julio (ref. Fernando Maila-Maila1, and Ibarra-Fiallo) go for the JVM strengths and the golden standard for testing, the JMeter. We can think of Gatling and the evolution of JMeter in terms of modern software with, smaller code base, but it still gets the advantages Java, and the JVM bring to us. Still, we can see from their research table quoted here that Gatling got the best score out of all the analyzed tools.

4.2 User Experience

Finally, developer experience, and how good and easy is it to use all that software? All of our candidates got a command-line interface, they all have recording features to use on the browser, dynamically generated HTML report pages, Locust and k6s here display an outstanding feature which is exporting the generated reported as CSV and still got a pretty web user interface for reading and accessing the data. Besides JMeter's age, it gots a large plugin ecosystem that allows it to handle all sorts of jobs and tweaks. There is not too much

Software	a	b	c	d	e	Total
Cano WebTest [7]	1	1	1	1	0	4.00
CLIF [8]	**1**	**1**	**1**	**1**	**3.12**	**7.12**
D-ITG [9]	1	1	1	1	0	4.00
Fast Web Performance Test Tool [10]	1	1	1	1	1.56	5.56
Funkload [11]	1	0	1	1	0	3.00
Gatling [12]	**1**	**1**	**1**	**1**	**3.58**	**7.58**
Grinder – Java Load Testing Framework [13]	1	0.5	1	1	1.56	5.06
JMeter – Load and Performance tester [14]	**1**	**0**	**1**	**1**	**4.2**	**7.20**
MStone [15]	1	0	0	1	0	2.00
Multi-Mechanize – web performance and load testing framework [16]	1	1	1	1	0	4.00
OpenSTA – Open Systems Testing Architecture [17]	1	1	1	1	0	4.00
Performance Co-Pilot [18]	1	1	1	1	1.56	5.56
Pylot – Performance & Scalability Testing of Web Services [19]	1	0	0	1	0	2.00
SoapUI [20]	1	0.5	1	1	0	3.50
TestMaker [21]	1	0.5	1	1	0	3.50
Tsung [22]	1	0.5	1	1	1.88	5.38
Xceptance LoadTest [23]	1	1	1	1	0	4.00

Figure 4.2: *Fernando, Monserrate, and Julio (ref. Fernando Maila-Maila1, and Ibarra-Fiallo) table comparing the diversity of software testing tools.*

to differentiate all of them.

4.3 Performance

Just before we start talking about the test results, a disclaimer first. The tests were performed on a 4-core Celeron server running Ubuntu 18.04 with 8GB RAM as the load generator machine. For the target server, it was a 4-core 4Ghz i7 iMac with 16G RAM with hyperthreading.

If we take a look at the plot below, from Ragnar Lönn study on Open Source load test tools (ref. Lönn) where he compare the memory usage and maximum RPS (requests per second), it is easy to see that there is a lot of tools capable of generating more traffic then JMeter, Gatling of Locust, and Drills takes a fatal hit here, showing the slowest performance possible. But, we really don't need that much request per second. If we could generate like 200 requests/second would be over our needs for this study, our best bet tools are on their scope range. It is important to keep in mind that these numbers were generated tweaking some performances for some utilities, like k6s, Artillery, and Locust. Still, since those tweakings are available to the final user, it is ok to rely on them.

Talking about memory usage. As expected from a Java application, not to mention an old Java application, they eat all the resources we have. JMeter here uses most resources, followed by Gatling, as known as a modern version of JMeter. K6s here seems to be our best bet, but we still have some concerns; K6s is written in Python, which has drawbacks

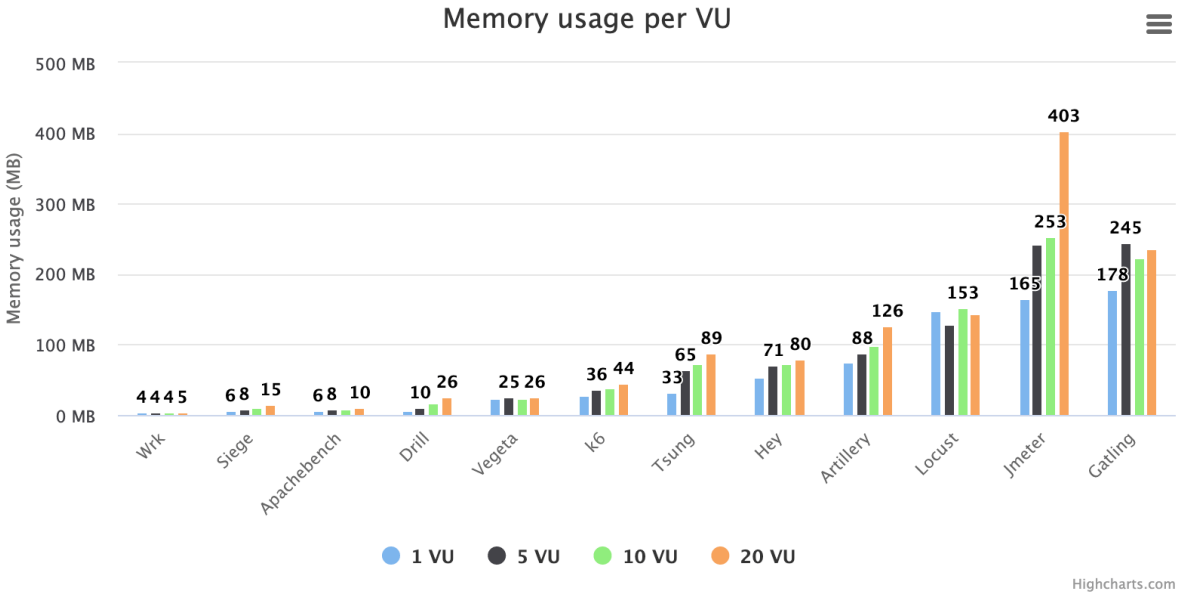


Figure 4.3: *Memory usage per Virtual User from Ragnar Lönn study (ref. Lönn).*

related to CPU performance. Since our testing server used for our tests has a nice amount of RAM, we are not so concerned about memory usage and more focused on CPU performance; ranking our possibilities here would be Locust, Gatling k6s JMeter.

Looking both, the memory usage per VU (Virtual User) and memory usage per request volumes show us the same results as before. Both Java application with better performance seems to use more resources than the other candidates' average. On the other hand, Gatling seems to be very consistent with its memory consumption and much better than JMeter, but both of them fail if compared to this study's other tools.

4.4 Gatling

For this study, we will go with Gatling because of its active development, a company running an enterprise solution behind it, and together with the community, its reliability provided by JVM; besides its low score on performance tests, Gatling got a company behind it, a big community helping the development which makes it easy to learn new technology, if more people are using it, there are more people to help you out with something you do no understand, as also mentioned in Comparative Analysis of Web Platform Assessment Tools by Solange Paz1, and Jorge Bernardino (ref. Paz1, and Bernardino). It is developed above the JVM; it is pretty easy to add the software to the CI pipeline across different computers and architecture. We can rely on that our software will behave the same and predict the results. Basically, our best approach would be Gatling, thinking about its longevity, durability, maintainability, learning curve, and performance.

JMeter, Locust, and k6s and good candidates, but JMeter is a pretty heavy software that may throat our tests due to the server capabilities. We would not need such a big ecosystem and plugins, not to mention its poor scriptable interface. K6s fails on a critical topic, it got a tiny community, and there is not too much content about it, making it difficult to learn and study already done user-cases. It is written in python, which for our purposes of simulation concurrencies users access to some URLs could lead to CPU throating and increased response times.

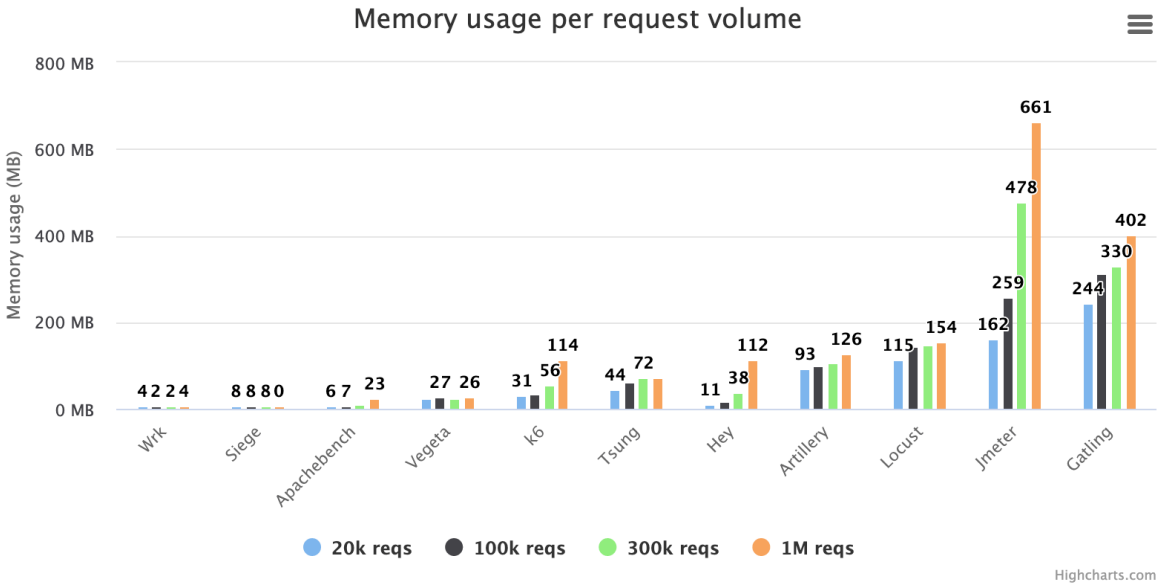


Figure 4.4: *Memory usage per request from Ragnar Lönn study (ref. Lönn).*

The main problem with locust is its development, which seems to be active and some periods but dead at others; on their repository, it is possible to see periods eighteens months of no updates, it is written in Python as well, and its license is MIT. Having said all of that, from now on, our reports will be all done using Gatling, what we can see as a great candidate marked as well by Andrei Proskurin (ref. Proskurin) on his research for testing java software.

Chapter 5

An analysis of Brazil's population

5.1 Motivation

For our study, we need to understand what kind our load our servers will have to handle. How many users will be using the application? What s the peak usage? To deploy a highly scalable web application is crucial to understand the expected load. For a simple example, if we would have a base use case of 10 to100 simultaneous users, we would need one instance of the Rails applications, but as the base users load increases, we would have to escalate the replica set or the nodes set.

Since this data is not available from Brazil's government, we will use population analysis to estimate the traffic that our application would have to handle.

5.2 Population analysis

For our analysis, we will use IBGE's (Instituto Brasileiro de Geografia e Estatística) demographic census from 2010. Brazilian capital's mean population is 1649841.73 people. But, there is something strange here. This number is too high; most of the city population is below 1 million people.

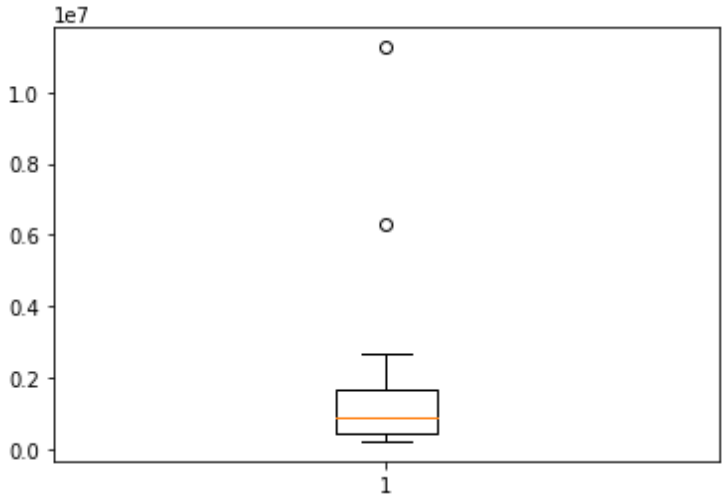


Figure 5.1: *Box plot of Brazilian capitals mean population.*

As we can higher in the boxplot, the expected population distribution gets 2 outliers; let's remove them and see what we get. Brazilian capital's mean population (without the 2

outliers) is about 1055080.67 people.

Now that we have a pretty clear understanding of the Brazilian population, it is needed to understand a bit more about São Paulo and its platform for contribution, called Governo Aberto.

We already know the largest cities in Brazil, which is our capitals, we are now focusing on our biggest city, São Paulo. But we are not looking for the overall population or access to the internet.

5.3 São Paulo's Governo Aberto analysis

Let's look to Governo Aberto's historical access data to see the traffic they need to provide. It is crucial to keep in mind that this data is from historical access of the Governo Aberto platform and reports, which does not mean it is the same traffic volume for the contribution platform. Still, it is as close as we can estimate data for the contribution platform on a state level.

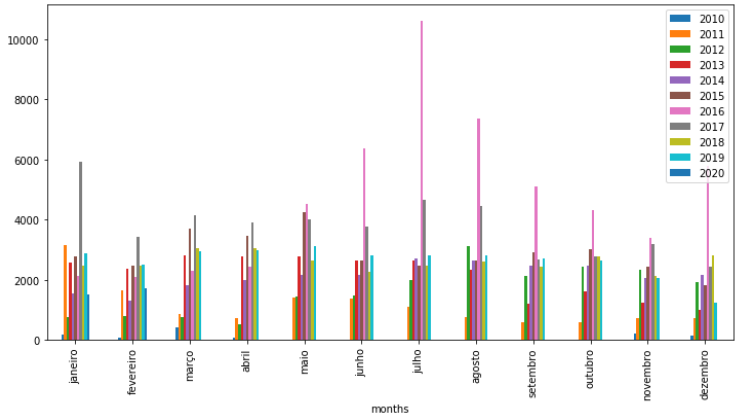


Figure 5.2: *Historical access during the past few years.*

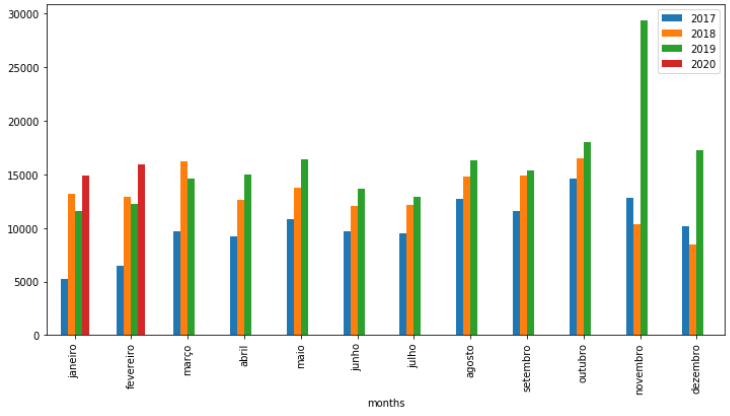


Figure 5.3: *Historical access during the past few years separated by months.*

As shown above, the catalog receives 1500 users a month with a peak usage at around 30000 users during the year. The first chart pops to our eyes that the web portal[1] receives as low as 12000 users a month at its peak capacity during the year.

[1]https://governoaberto.sp.gov.br

Chapter 6

An analysis of Consul's original code

6.1 Host machine and OS specification

Linux debian 4.19.0-8-amd64

Device	Class
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
Address sizes	36 bits physical, 48 bits virtual
CPUs	8
On-line CPUs	0-7
Thread per core	2
Core per socket	1
Sockets	1
NUME Nodes	1
Vendor ID	GenuineIntel
CPU Family	6
Model	58
Model name	Intel(R) Xeon (R) CPU E3-1230 V2 @ 3.30GHz
Stepping	9
CPU MHz	3637.993
CPU Max MHz	3700.0000
CPU Min MHz	1600.0000
BogoMIPS	6585.38
Virtualization	VT-x
L1d Cache	32k
L1i Cache	32k
L2 Cache	256k
L3 Cache	8192k

Flags	fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer es vnmi xsave avx f16c rdrand lahf_lm cpuid_fault pti dtherm tpr_shadow flexpriority ept vpid fsgsbase smep erms xsaveopt ida arat pln pts sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer es vnmi xsave avx f16c rdrand lahf_lm cpuid_fault pti dtherm tpr_shadow flexpriority ept vpid fsgsbase smep erms xsaveopt ida arat pln pts

Device	Class
Sytem	Computer
bus	Motherboard
memory	31GiB System memory
processor	Intel(R) Xeon (R) CPU-1230 V2 @ 3.30GHz
bridge	Xeon E3-1200 v2/Ivy Bridge DRAM Controller
bridge	Xeon E3-1200 v2/3rd Gen Core processor

6.2 Tests methodology

As we can see from the above commands, I'm working on an 8 cores machine with 31GB of RAM. I will use Minikube with default RAM (9GB) and then test it with 15GB and 25GB, leaving 6GB for the host OS.

For this study, and as we will use only one host machine with Minikube to simulate a cluster, at any moment of this document, when I'm referring to cores or nodes, I'm talking about the same thing. Each core on the host machine represents a node on a Kubernetes cluster.

I'm going to start simulating a cluster with 2 nodes and increase it by a power of 2 until it reaches 8 nodes. At the same time, we will test our cluster with 1, 2, 4, 8, 16 replicas of our application; 1, 10, 100, 1000, and 2000 users connecting during 1 second, 10 seconds, and 60 seconds.

6.3 Get to know the data

.

First of all, let's get to know what kind of data we are talking about. Check how the data is disposed of.

Data columns	Data Type
id	4935 non-null int64
simulation_duration	4935 non-null int64
number_of_simultaneos_users	4935 non-null int64
delayed	4935 non-null int64
cores	4935 non-null int64
memory	4935 non-null int64
replicas	4935 non-null int64
total_requests	4935 non-null int64
total_requests_ok	4935 non-null int64
total_requests_fail	4935 non-null int64
t_800	4935 non-null int64
t_800_1200	4935 non-null int64
t_1200	4935 non-null int64
failed	4935 non-null int64
minResponseTime_total	4935 non-null int64
minResponseTime_ok	4935 non-null int64
minResponseTime_fail	4935 non-null int64
maxResponseTime_total	4935 non-null int64
maxResponseTime_ok	4935 non-null int64
maxResponseTime_fail	4935 non-null int64
meanResponseTime_total	4935 non-null int64
meanResponseTime_ok	4935 non-null int64
meanResponseTime_fail	4935 non-null int64
standardDeviation_total	4935 non-null int64
standardDeviation_ok	4935 non-null int64
standardDeviation_fail	4935 non-null int64
percentiles_50_total	4935 non-null int64
percentiles_50_ok	4935 non-null int64
percentiles_50_fail	4935 non-null int64
percentiles_75_total	4935 non-null int64
percentiles_75_ok	4935 non-null int64
percentiles_75_fail	4935 non-null int64
percentiles_95_total	4935 non-null int64
percentiles_95_ok	4935 non-null int64
percentiles_95_fail	4935 non-null int64
percentiles_99_total	4935 non-null int64
percentiles_99_ok	4935 non-null int64
percentiles_99_fail	4935 non-null int64

Fine, we are going to focus our analysis on response time and failures. Our goal is to understand how to increase the rails application performance to handle as much as possible requests per Kubernetes pod.

6.4 Exploratory analysis

6.4.1 2 cores and 9000MB RAM

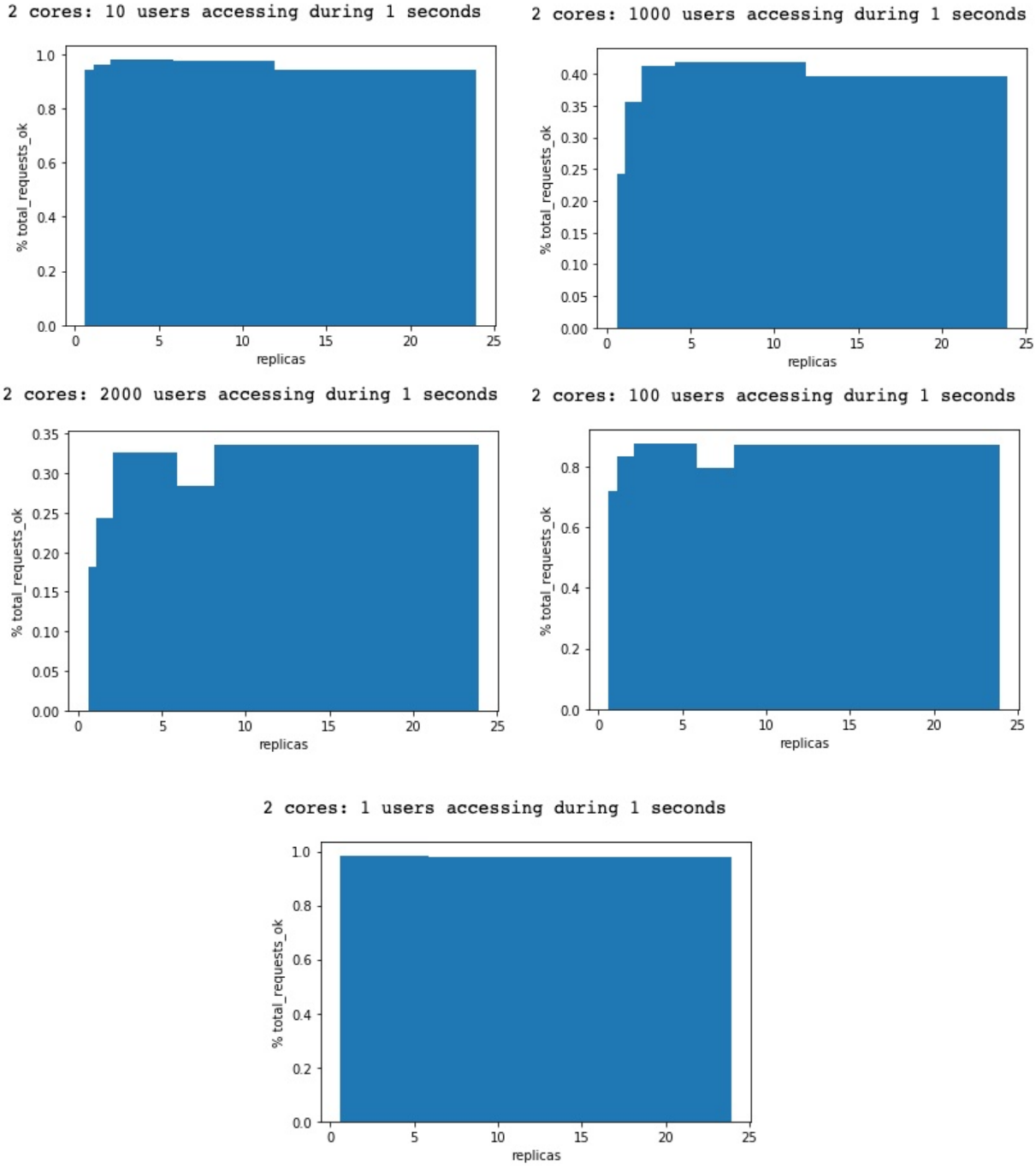


Figure 6.1: *Percentage of successful requests 2 cores, 9000mb. Accessing for 1 second.*

As expected, as we increase the number of simultaneous users accessing the application, the number of answered requests decreases as low as 20%. But, let us take a look from the other side, starting with simple math. We are talking about 2000 users accessing the application for 1 second, which means we have 12000 users each minute and a total of approximately 0.5 billion users accessing during the month. That is too much; let's not forget that we are using only 2 nodes on our cluster.

Let us see now how good was our response time, considering that if a request takes less than 800ms to return, it is the best scenario; if it takes between 800ms and 1200ms, we accept it as well, but if we got over 1200ms, we start to get worried.

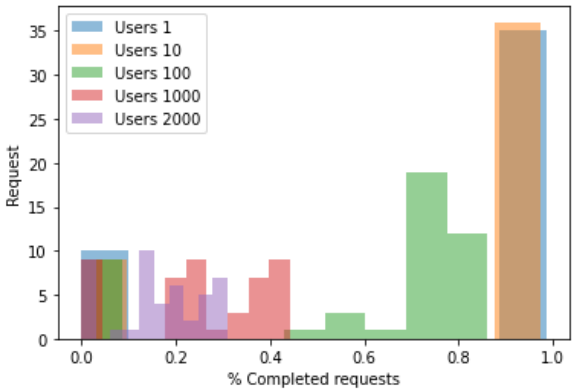


Figure 6.2: *Completed requests with 2 cores, 9000mb. Accessing for 1 second.*

As we can see from the above plots, most of the requests, which are completed, keep inside our margin for "good" response time (t $<$ 800ms). Now forget for a moment the number of replicas we are using on our cluster; let's check out how a 2-machine cluster could handle that amount of users accessing the application during 1 second.

As expected, we can check that we can handle not so many simultaneous user connections without computer power. Between 1-100 users/second, we can see that most of the requests were handled just fine by our cluster. We need to check out what happens if we split this user load for 10 seconds and 60 seconds.

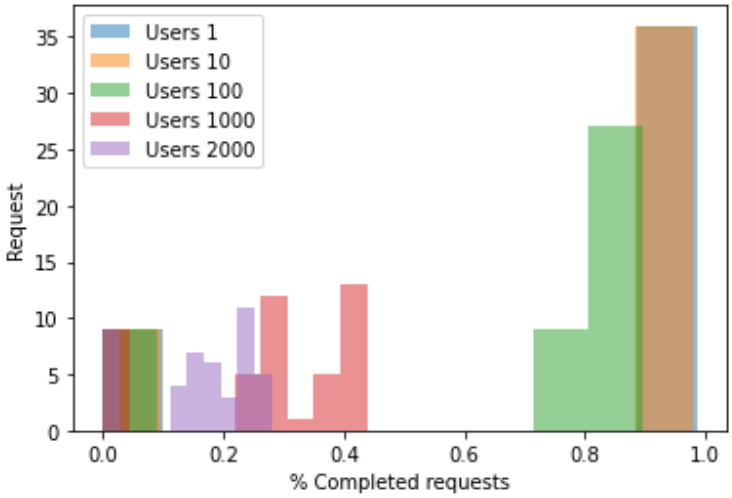


Figure 6.3: *Completed requests with 4 cores, 8192mb RAM for 60 seconds.*

We got a problem; we were testing with 2000 users accessing during one second, now we just moved to 2000 users accessing during 60 seconds, but as we can check on the above histograms that we only got a slight increase over the completed requests signaling us that we are reaching top performance for 2 nodes cluster, let's see what changes if we increase our cluster size.

6.4.2 4 cores and 9000MB RAM

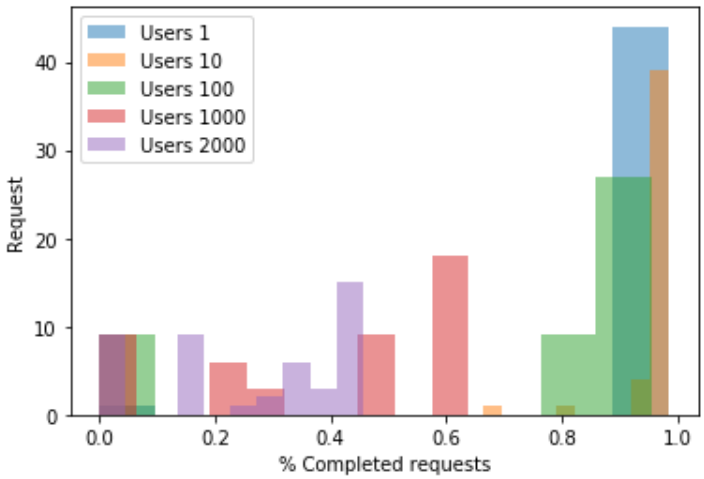


Figure 6.4: *Completed requests with 4 cores, 8192mb RAM during 60 seconds.*

What pop's up to our eyes immediately? We got a bigger number of requests being accepted at the same time. Our distribution got larger, and a larger amount of requests were answered. As we will see, not all of them got good response times, but the number of answered users increased significantly. Ok, let us check now the response times.

At this point, the user should have already understood what is happening and our goal. As we increase the number of available resources (cluster nodes) dedicated to our application, we get a slight performance increase, getting more users answered correctly. Their response times get slowly better. Jumping into our machine performance, handling all those users if we increase their time available.

Let us stop here for a moment. We are focusing on users during a 60 seconds access time. But our numbers are still not ideal. Our goal is to handle as much as we can and reach as close to 2000 users for 60 seconds. As we can see from the above chart, with 4 nodes, 8192MB total available RAM to the cluster for 60 seconds, we handle almost all requests at a max of 100 simultaneous users. Let's jump to 8 nodes.

6.4.3 8 cores and 9000MB RAM

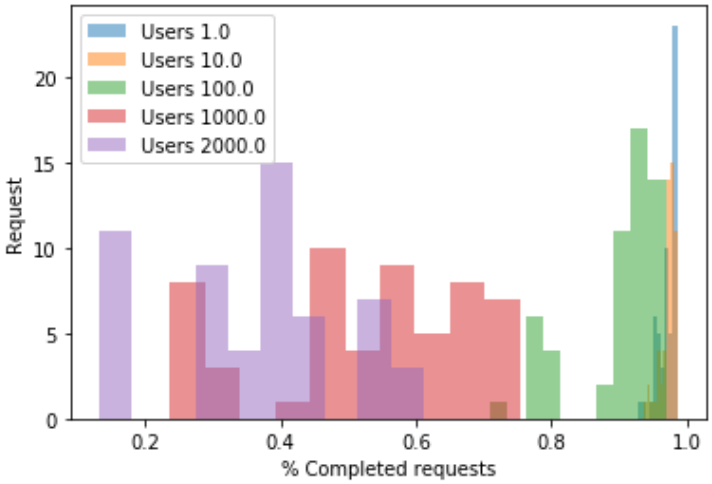


Figure 6.5: *Completed requests with 8 cores, 8192mb RAM for 60 seconds.*

Now, we should take a special look at our data. At this point in our study, we are using 8 nodes for our cluster simulation, and as the reader might remember, that is the number of cores that our server got. Consequently, we might expect that this is as good as we can get with this server. So, let's see how our cluster behaves with 8 nodes with 60 seconds.

Let us go step by step and take a closer look. First things, first. At 1 user, we handle 100% of the requests, but nothing new here; we are talking about just 1 user per minute. Surprisingly, at 100 users/minute, we still handle almost 100% of the requests, not all of them at our perfect timing response, but we are handling them. Considering the numbers, 100 users/minute are 144000 users/day, and approximately 4.3 million users a month if we look to a city of a population of 1 million people, as much as everyone accessing our application one time a week.

With our current setup, on a 2 nodes cluster with 8192MB RAM, we can handle 4.5 million simultaneous users a month (100 each minute) witch mean we can handle the expected population using our services three times a month, or 67% of then using it every week, but let us see if we can get better results.

But, if we remove the outliers cities, São Paulo and Rio de Janeiro, we got a population of 1 million inhabitants, which means we could handle all of them using our application once a week. Looking at 1000 simultaneous users, we can handle 80% of the requests just fine; in other words, 800 users wouldn't notice any downtime on our serves, but 20% of them will be angry.

Considering just 800 users simultaneously, accessing for 60 seconds, we are talking about 34.5 million users a month. Handling even the most populated city of Brazil, São Paulo, using our service 3 times a month.

If we look at the access data of the Governo Aberto portal's we see that if we handle 10 users a minute, we could handle almost 450.000 requests, which could serve just well the peak request capacity of 30.000 users/month. Here, in our example, we are using São Paulo's Open Government data, the most populated city.

As shown from the above plot, we managed to handle almost 80% of the requests of 1000 simultaneous users but around 50% of the 2000 users, getting even worse results, which shows us that if we get an overload of our server with peak usage, we should increase the number of nodes on the cluster. In contrast, we can see that it is quite simple for 100 simultaneous

users to handle the traffic. Now we are going to check what the response time for 1000 users was?

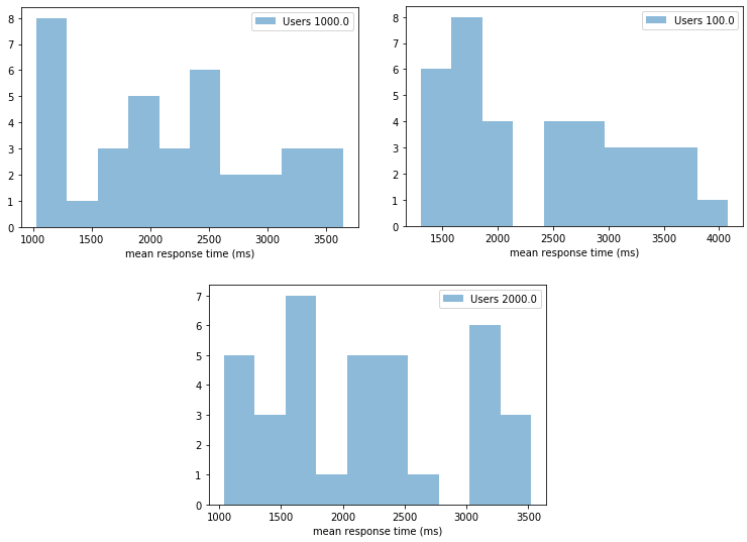


Figure 6.6: *Mean response time with 2 cores, 8192mb RAM for 60 seconds.*

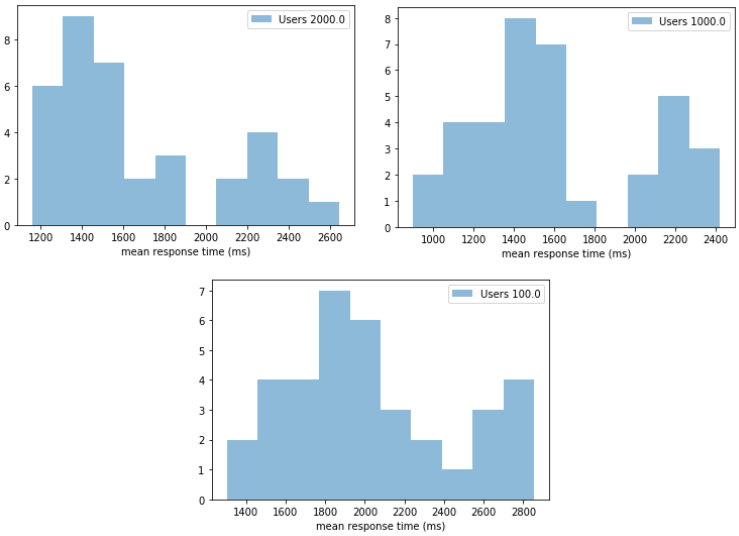


Figure 6.7: *Mean response time with 4 cores, 8192mb RAM for 60 seconds.*

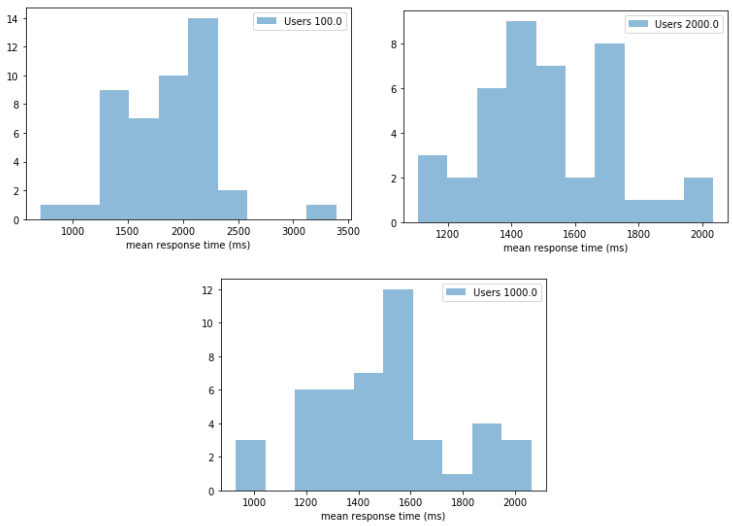


Figure 6.8: *Mean response time with 8 cores, 8192mb RAM for 60 seconds.*

As shown in the above examples, as we increase the simultaneous user access, the response time deteriorates quickly. Still, it seems to have a relation between the response time, the cluster process power, and the number of acting nodes. It is crucial to create an equilibrium between those resources, which requires further investigation.

6.4.4 8 cores and 16000MB RAM

We are investigating the cluster performance focusing on process power only (amount of cores of the cluster), but we were limited to 8291MB of RAM. What if we increase the cluster memory? We will check out what happens when we increase the cluster memory to 16384MB and 25000MB. It is wise to remember that the server on which we perform these tests only has 26624MB RAM, so we will leave 6000MB for the OS to handle our tests without page faults, causing it to slow down its performance.

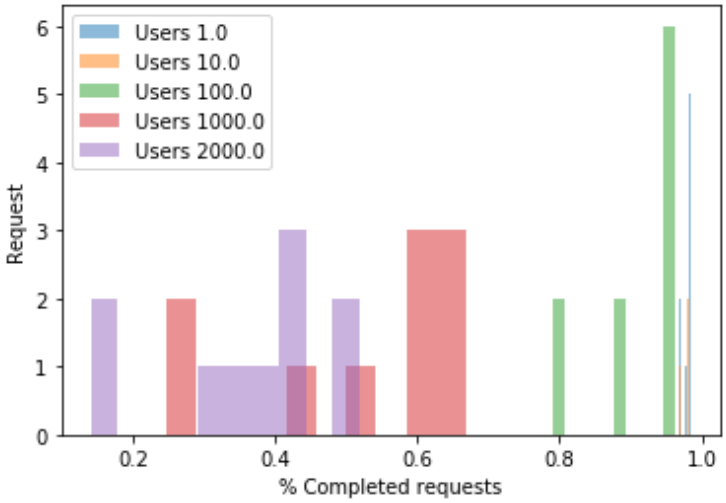


Figure 6.9: *Completed requests with 4 cores, 16384mb RAM for 60 seconds.*

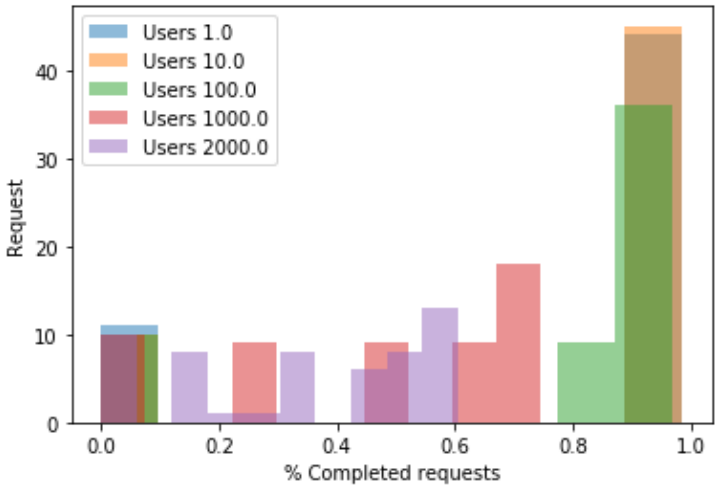


Figure 6.10: *Completed requests with 8 cores, 16384mb RAM for 60 seconds.*

From the above plots, observing the curves of 1000 users and 2000 users accessing for 60 seconds, we can see that deploy more replicas is not the solution. That is the opposite; as a developer handling the server, you must understand the sweet spot to get the best performance out of your cluster. The last plot shows us that for 4 cores, our best approach is to launch one replica per core/cluster node. If we were to look at the handled jobs, we managed to handle just fine between 50% and 70% of our requests, depending on the amount. Let us make ourselves conservatives and say that we can handle 500 simultaneous users/minute, leading us to approximately 21.6 million simultaneous users/month.

As the reader may recall, we doubled our core dedicated to the cluster that all cores available on our machine to simulate the cluster. However, we still keep the same amount of memory available for double the processing power amount. This leads us to a small amount of performance increase, around 10% for 2000 users. Still, nothing too much significantly doubled the process resources, and half of the memory is available to each replica. Thus, it is interesting to notice that we could maintain stability with more replicas; for example, the peak performance was approximately 10 replicas. Let's jump now to full cluster power, 26624MB of memory, and 8 cores. Finally, let us plot the mean response time and compare the changes that increase memory at each step.

6.4.5 8 cores and 26000MB RAM

Ok, we were able to increase the performance a little bit, but no too much, and now we are using approximately 3000MB memory for each node on our cluster, so what going wrong here? Here we face another technical issue; we are using the same machine to simulate our cluster and our test utility, which causes one tool to throttle the other; in other words, we can't guarantee the resources needed because if we give all machine resources to our cluster, our test application won't be able to simulate all those user sessions. And here is where and cant advance furthermore on our investigation. In our best scenario, we could handle around 700 to 1200 users simultaneously, if we took the mean, around 950 simultaneous users, leading to approximately 41 million users/month. Let us look at the mean response time; what can we say? Did it increase?

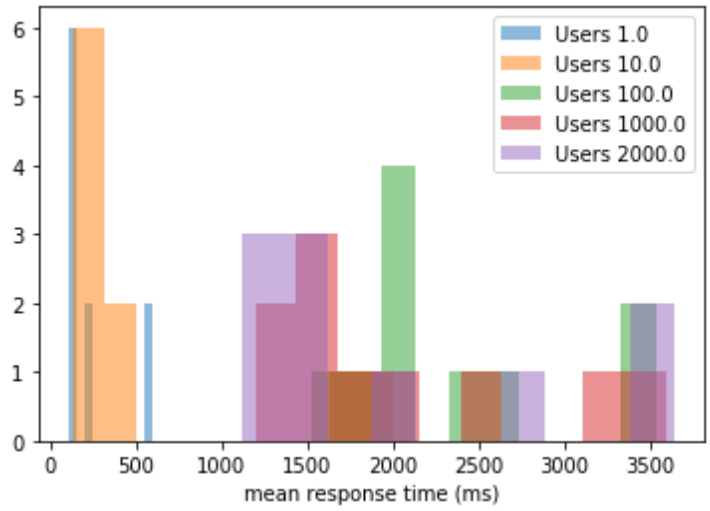


Figure 6.11: *Mean response time with 4 cores, 26624mb RAM for 60 seconds.*

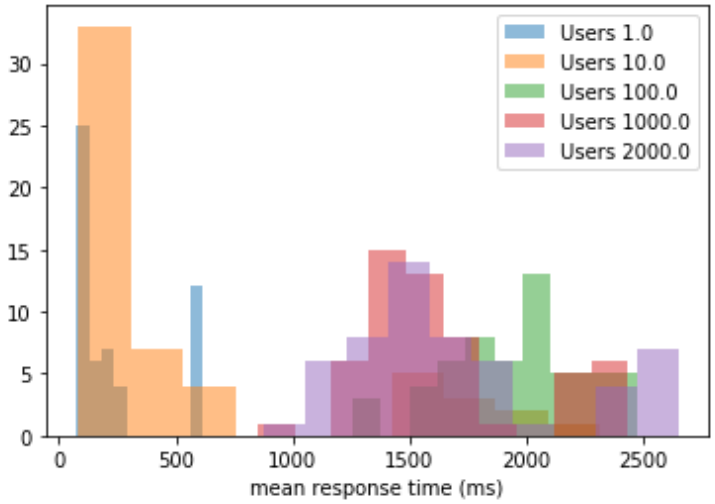


Figure 6.12: *Mean response time with 8 cores, 26624mb RAM for 60 seconds.*

6.4.6 Tests analysis

Now we have to split this task into 2 tasks; first, we are going to compare the results between 16GB and 26GB of memory to see what changed, separately for 4 and 8 cores, then we look at jumping from 4 to 8 cores at 26GB of memory.

Closing our analysis to 1000 and 2000 simultaneous users, for easy-of-understanding reasons, most of the requests were handled between 1500ms and 2500ms. Doubling the available memory allowed us to shift these charts slightly to the left, making it closer to 1000ms and 2000ms. Cheers, we are getting close to the optimum performance given the resources we have. Now observe the 8 core, between 16GB and 26GB of memory, the first response, for 16Gb memory, we handle most of the request from around 1000ms to 1800ms with a quarter of the requester being over 2000ms and as high as 3500ms, and again, increasing the memory shift our chart a little to the left, making the handling request response time lower. And we see a peak on 1500ms response time.

So, what do we see here? As we increase performance, we are going a little bit more user-handed but increase our response quality, making it faster for the answered users.

Talking about the changes between 4 and 8 cores, we see that our curve changed from a separate bar to more something that remembers a Normal distribution; in other words, we are better handling the same amount of users, not quite the same amount as we have seen before, like 10-20% more.

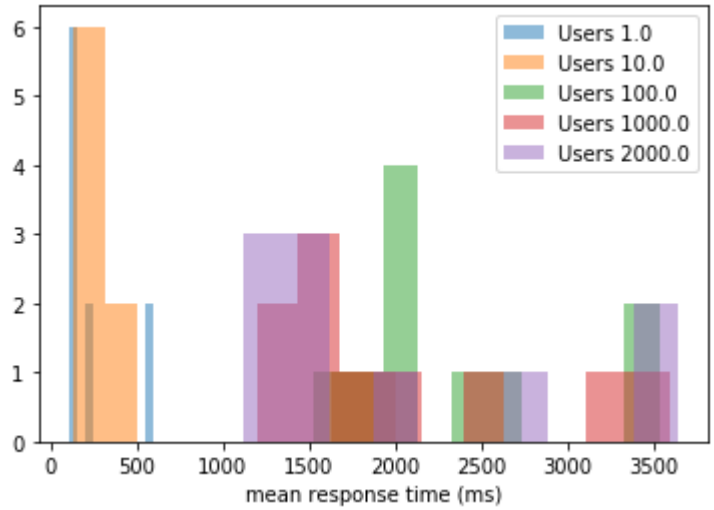


Figure 6.13: *Completed requests time with 4 cores, 26624mb RAM for 60 seconds.*

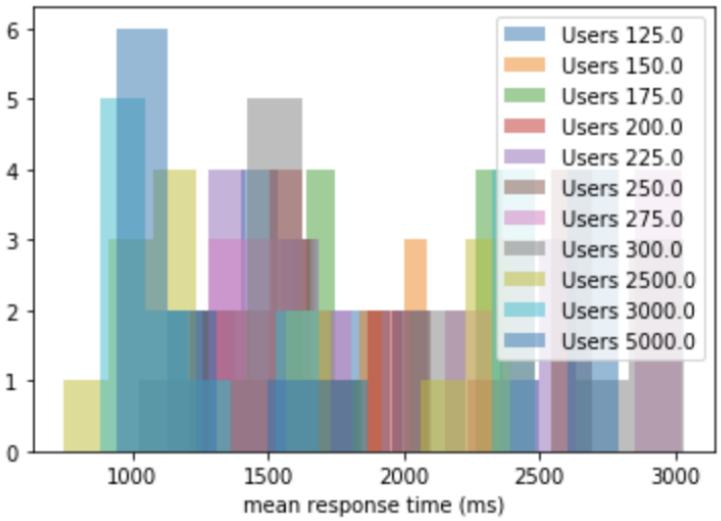


Figure 6.14: *Completed requests time with 4 cores, 26624mb RAM for 60 seconds.*

The above histogram summarizes how our application scale to handle those simultaneous users; at 2000 users, we can handle between 0.4 and 0.6 of the requests, and this performance increases a little bit as we double the cores, from 4 to 8, shifting from 0.3-0.5 to 0.4-0.6. On the other hand, if we compare with 16GB of memory, we had a wider range, going from 0.1 to 06; the difference here is how sparse those two distributions are on this interval.

The main problem here is that at 100 simultaneous users, we already dropped the percentage of handled requests. At 1 user, obviously, we handled 100%, but at 100 users, we dropped to something between 80% and 90% of the requests, but why is it dropping so fast? The cluster got 8 cores and 26000MB RAM for its disposal, our maximum hardware capacity, but the performance shows signals of deterioration too fast for the number of simultaneous users. Another interesting part of these 2 plot is that the proportions keep almost

linear when doubling the cores from 4 to 8. If you look at the first and second plots, one can notice that at 100 simultaneous users, we handle, on both 4 and 8 cores, around 80-90% of the requests. We can conclude that the core is not throating the cluster performance, at least for 100 simultaneous users. And here it comes, the big unknown, if we look at 4 cores and 8 cores with around 16000MB RAM, around 10000MB less than these last plots, the proportion keeps itself around 80-90% of the requests, so, let's recapitulate what happened here. We increased all available resources for our cluster; we doubled the core amount from 4 to 8 cores and increased the RAM amount from 16000MB to 26000MB, but the requested handle keeps the same on all 4 scenarios, around 80-90%. What is causing this? Let us plot a few histograms and see when we throttle 100 simultaneous users.

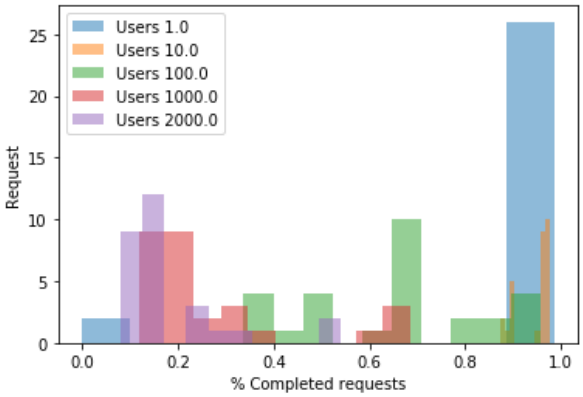


Figure 6.15: *Completed requests time with 8 cores, 4096mb RAM for 60 seconds.*

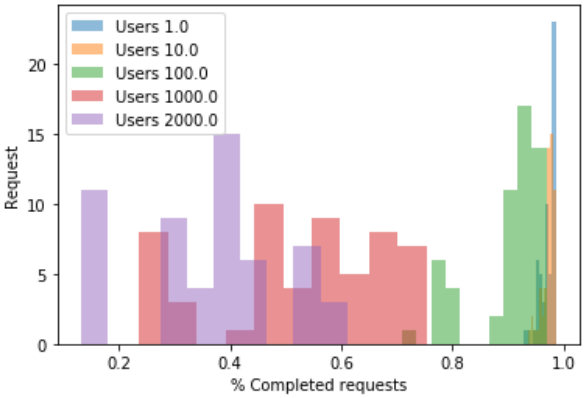


Figure 6.16: *Completed requests time with 8 cores, 8192mb RAM for 60 seconds.*

Ok, we need to make a stop here. We just plotted our cluster performance, handled requests, kept the core amount constant at 8 cores, and started lowering the RAM amount. At 4096MB RAM and 8 cores, we just dropped the percentage of handled requests with 100 simultaneous users, going from 80-90% to, on its worsts, 30%. We got a pretty big hit here, but we need to investigate; furthermore, let us keep the RAM now at 8192MB, where we know that it can handle 100 simultaneous users, and see how cores affect this response time.

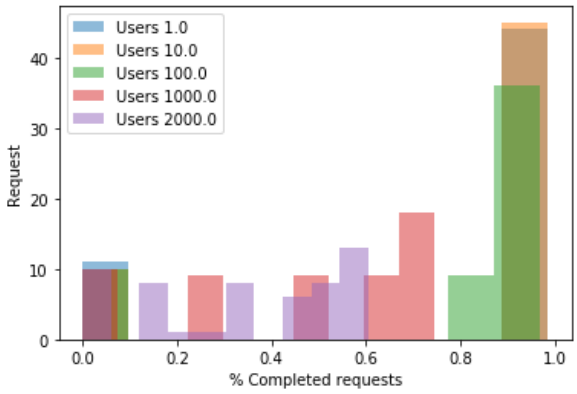


Figure 6.17: *Completed requests time with 8 cores, 16384mb RAM for 60 seconds.*

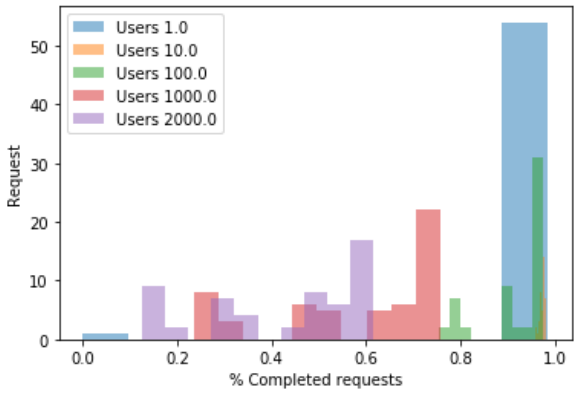


Figure 6.18: *Completed requests time with 8 cores, 26624mb RAM for 60 seconds.*

That is interesting; if we run our cluster as low as only 2 cores, as a 2 node cluster with 1 core each node, but with 8192MB RAM, the cluster keeps it static and handles the same amount of users. We still need further investigations because we can increase the cluster cores power from 2 to 8 and increase the memory from 8192MB to 26624MB, but the handled requests keep constant around the same interval. What happens if we lower the time interval? Until now, we kept our time interval constant at 60 seconds. The last 3 plots show us that, surprisingly, the number of handled requests at 10 users kept constant around that same interval, 80-90%. We need to plot the total number of requests to see if we are hitting the maximum request- s/second that we can handle.

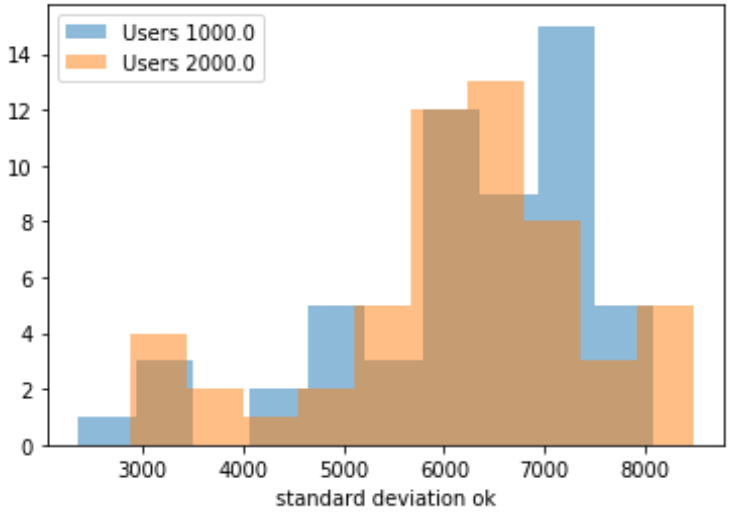


Figure 6.19: *Standard Deviation with 8 cores, 8192mb RAM for 60 seconds.*

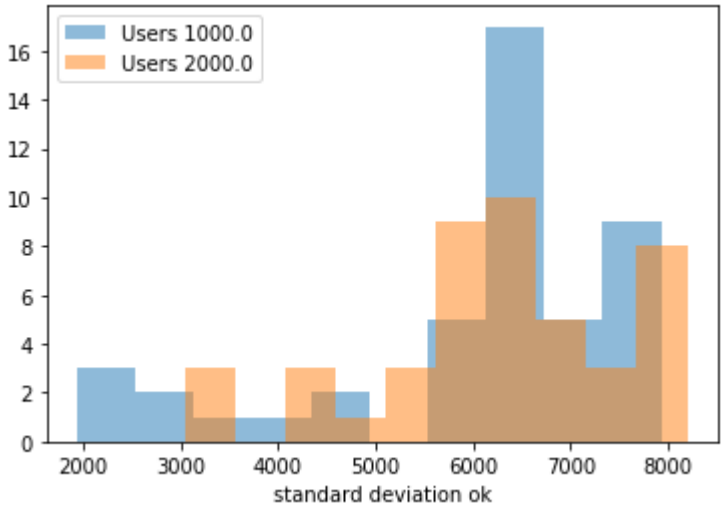


Figure 6.20: *Standard Deviation with 8 cores, 16384mb RAM for 60 seconds.*

As expected, the number of requests / second is not the problem here; we are far from its limits. So, what do we see here? The answer is simpler than the reader may expect. We are just degrading our cluster. If we want to serve over 95% of the requests, it is essential to increase our server. As we try to reach 100% requests, we need to increase the number of servers available to the cluster. And here, we will call the Pareto Principle, which states that "for many events, roughly 80% of the effects come from 20% of the causes". That is exactly what we see here; with 20%, we can handle 80% of the requests. Still, as we are moving to a better quality application, which tries to reaches 100% response tax, you will have to do the other 80% of the effort, and that is why to handle these last few requests, we need so much more servers. The closer we are to 100%, the bigger the number of nodes on the cluster.

To demonstrate what is happening here, we have to plot our user access for 60 seconds. Still, for a more thin gradient, for example, with 1, 10, 25, 50, 75, and 100 simultaneous users, so we can understand where our cluster drops from 100% request rate to 80-90% and find how our cluster behaves and handles optimum simultaneous users. We will make another stop on our analysis to take a closer look at what is happening here and its scenario. Still, for simplicity, we will keep only the 60 seconds scenario.

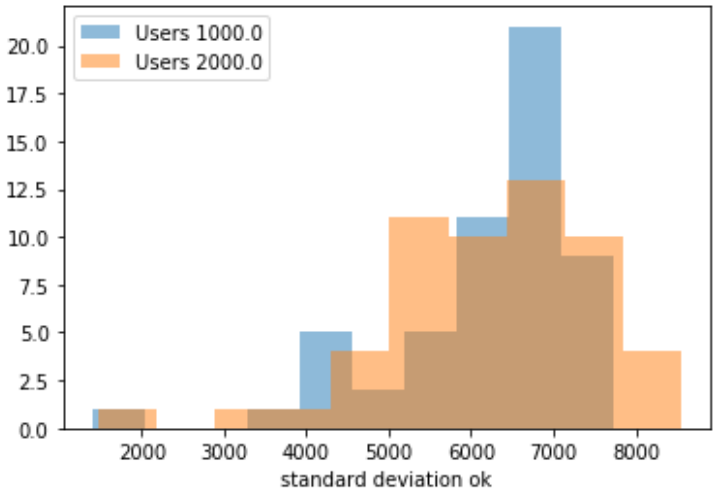


Figure 6.21: *Standard Deviation with 4 cores, 26624mb RAM for 60 seconds.*

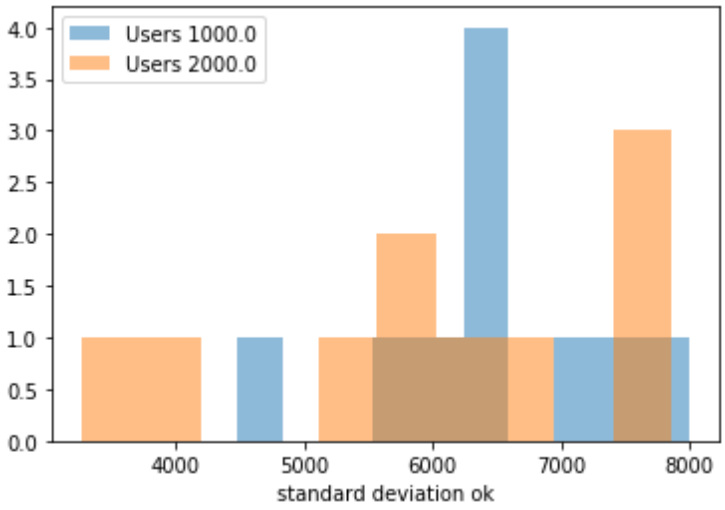


Figure 6.22: *Standard Deviation with 4 cores, 26624mb RAM for 60 seconds.*

The last 4 plot shows us the "softening" and "clustering" of our data point since we give more resources to our cluster, it accommodates more requests, but these extra requests come with a price. The mean response time increases over 1000ms, but what is causing this? As we can observe from the last plot, double the core amount gives a significant increase in requests responded successfully. Still, double the memory, or increase it by 50%, helps accommodate more requests at a more uniform rate, instead of handling some requests fast but others giving failure completely. To summarize, memory here plays an important spot in increasing each user's performance, allowing a uniform scenario across a bigger range of users. The cores/cluster nodes act on handling those requests; as expected, it is responsible for the processing power.

For example, by doubling the nodes handling requests, we got over 3 times the number of successful requests. Increasing memory allows the number of successful requests to increase by 3 points, mostly because of timeout requests. We had process power to handle more requests the RAM was fully used. So, increasing the available memory allows Rails to handle a little more requests. Still, our focus here is how it increases the performance for those handled users, making navigation flow softly.

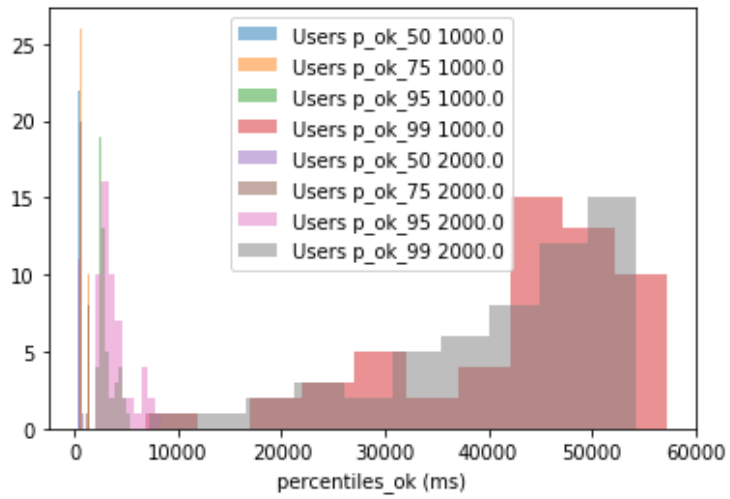


Figure 6.23: *Quartiles distributions for 50, 75, 95, and 99.*

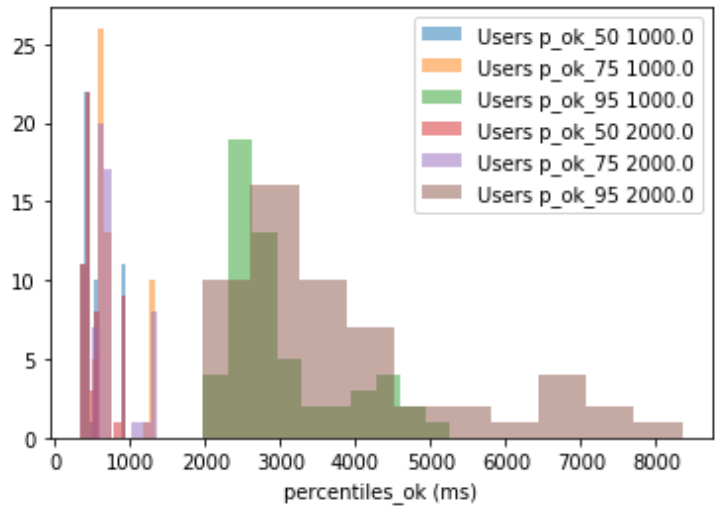


Figure 6.24: *Quartile distributions for 50, 75, and 95.*

The noticeable effect is that at 100 simultaneous users, we handle 100% of the request, as we increase to 100%, it sifts down to approximately 75% of then, and, ultimately, at 2000 users, we can handle only 50% of the requests but with an insanely high page load average time, on its peak, reaching over 60000ms with 2000 simultaneous users and a percentile of 99%. As we can see clearly from the above plot, 75% of the successful requests at 2000 are handled in about 1.5 seconds, which is pretty good. However, the last 24% suffers a lot with increase server load time, but still usable, and, for least, 1% of successful requests will have a page load time of over 60000ms, which is unacceptable. Here we need to conduct further investigation to find that sweet spot where our cluster handles the job.

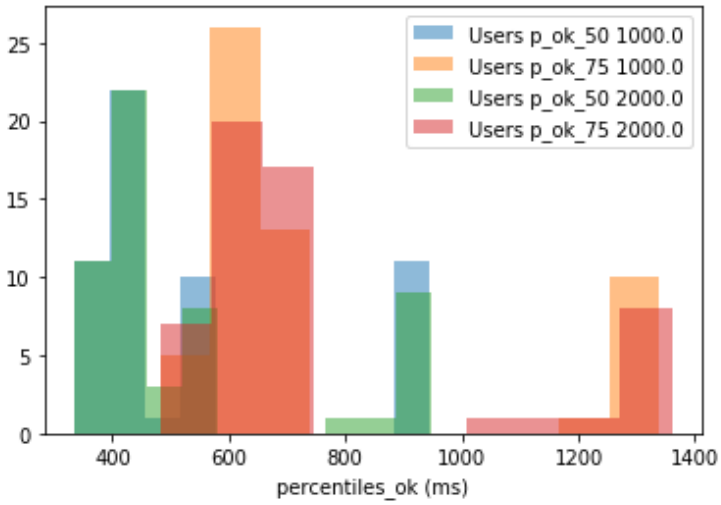


Figure 6.25: *Quartiles distributions for 50, and 75.*

We shall pay attention to that with 2000 simultaneous users; the 99 percentile is 40000ms and 60000ms. We have seen before considering that the average user should expect the page to load under 100ms and acceptably wait until 1000ms. Still, at 10000ms, this same user should abandon the task and think the page is broken; we are handling on a good condition around 75% of or target group, at the point that the 95 percentile shows us the timing of 2000ms to 4000ms which is considered a poor performance website but acetable by the average user. Summarizing our condition, we are handling pretty good 75% of the requests, leaving 20% with a bad experience, and the last 5% have completely abandoned us for this poor performance.

According to what we have been discussing so far, we are pretty close to handling those 2000 simultaneous users a minute; the problem is that we need to lower those page load response time, make navigation more reliable; our goal is not to increase the user load nor the amount of simultaneous users, and here we hit a giant wall, what we see is that our cluster is being throttled, we have enough CPU power to do the job but not enough memory, what happens is that the system starts to using the swap area which leads us to disk access and an increase in Read and Write operations. Recalling from the beginning of this chapter, we are using a hard drive that explains why we suddenly, at this last 1%, jump from under 10000ms to almost 60000ms. And for this, unfortunately, we will have to go sideways since we can't increase our memory power.

Chapter 7

Proposed improvements

7.1 Strategy to decrease page load time

As we are trying to deliver data as fast as possible to increase a web application's performance, our goal is simple: cache. We are going to implement some caching on Consul on some levels. First, we are going to cache only translations, i18n[1] related queries. Our second approach is more aggressive. This time, we will cache an entire page to see how it affects the performance since some pages like your website's index are frequently accessed. Sometimes, your application has to make hundreds of access to the database to get the information required to deploy an important and frequently accessed page. This seems to be a clear sign of a great fit for caching.

Accordingly to Instant Redis Optimization How to from Arun Chinnachamy (ref. Chinnachamy) the great advantage that the NoSQL database brought to the industry was fast data access. The purpose of using NoSQL, and the NoSQL movement that started to grow was the fast data access, and again, it Redis is the leading NoSQL database for fast data in the industry. Still, it is worth mentioning the options, which are Cassandra, MongoDB, Riak, CauchBase, Memcached. Another interesing study about caching tools performance is Performance Comparison between Five NoSQL Databases from EnqingTang, and Fan which focus their analysis on Redis, MongoDB, CauchBase, Cassandra, HBase. In their study, they highlight exactly what we are trying to investigate in this study. In the era of Big Data, mobile internet, and a huge amount of devices collecting data that need to be processed and disposed of to the client queries, performance becomes insentient. That is where NoSQL and cache come into play. In our scenario, we apply the same content on frequently accessed pages, which is a perfect scenario for caching. It can release the processing power required to deliver translations for static text or the whole page, if necessary.

Another interesting analysis is NoSQL Databases: Critical Analysis and Comparison from Adity Gupta, Swati Tyagi, Nupur Panwar, Shelly Sachdeva and Upaang Saxena (ref. Gupta *et al.*). The NoSQL model differences are document, wide columns, Graph, and single shared operation (where Redis shines. They also brought attention to the CAP theorem, named Brewer's theorem after computer scientists elaborated on Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services (ref. Gilbert, and Linch). We would like to highlight the figure 7.1 that emphasizes the differences and advantages of each data storage model.

[1]Rails Internationalization (I18n) API: https://guides.rubyonrails.org/i18n.html

Data model	Performance of queries	Scalability of data	Flexibility of schema	Structure of database	Complexity of values
Key-value store	High	High	High	Primary key with some value	None
Column Store	High	High	Moderate	row consisting multiple columns	Low
Document Store	High	Variable (High)	High	JSON in form of tree	Low
Graph Database	Variable	Variable	High	Graph – entities and relation	High

Figure 7.1: *Different NoSQL databases on basis of Non-functional features from from Adity Gupta, Swati Tyagi, Nupur Panwar, Shelly Sachdeva and Upaang Saxena (ref. Gupta et al.).*

Quoting, "Cap Theorems states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:"

1. "Consistency: Every read receives the most recent write or an error."
2. "Availability: Every request receives a (non-error) response, without the guarantee that it contains the most recent write."
3. "Partition tolerance: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes."

And that is where Redis used together with the default Postgres database used on Consul makes the difference. The standardized relational database provides all the benefits of ACID (which the reader can get a deeper understanding of reading Takai *et al.*) where Redis enable us to deliver fast read operations. On other words, Redis provides Availability where Postgres provides Consistency.

Finally, we want to cite Shanshan Chen, Xiaoxin Tang, Hongwei Wang, Han Zhao and Minyi Guo (ref. Chen *et al.*) that wrote Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis, which presents an extended analysis on Redis as fast in-memory storage access for web applications. On their analysis, it is proposed to create a "client-node" connection on the Redis cluster, on which the client connects directly to the correct Redis node where the correct data is stored, and accordingly to their article, they were able to increase the performance of the cached application by 2 times. Another interesting proposed strategy is a Master-slave Semi Synchronization over TCP, where it improves the consistency of Redis slaves/master data and performance by 5%.

We can see a diversity of studies relating the benefits of caching for application performance if we follow the study A Qualitative Study of Application-level Caching by Jhonny Mertz and Ingrid Nunes (ref. Mertz, and Nunes), we see a different approach, the authors have analyzed scenarios of application caching, ranging from automatic aching techniques, manual caching, background work caching, and the benefits of one or the other. For example, talking about background work caching we mean caching HTTP requests so the process checks for a cached requests before send the requests to the controller, as we can see in figure 7.2 extracted from Jhonny Mertz and Ingrid Nunes (ref. Mertz, and Nunes) article. Accordingly to the same study, over 80% of problems related to cache on famous open sources

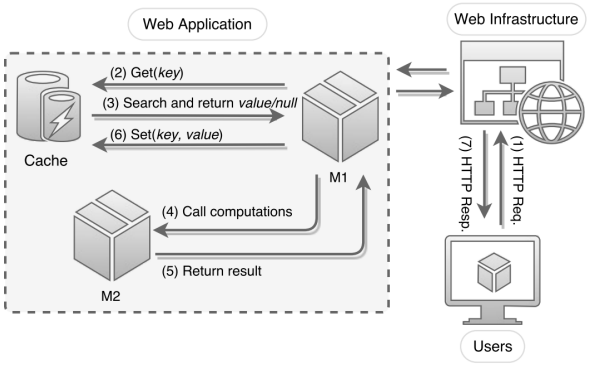


Figure 7.2: *Using Application-level Caching to Decrease the Application The workload from Jhonny Mertz and Ingrid Nunes (ref. Mertz, and Nunes).*

projects, such as ownCloud, Spree, Shopizer, OpenCart, Pencilblue, S1, Discourse, OpenMRS, PrestaShop, and Open edX are related to design and implementation (when grouped in categories), so what we can conclude from this study is that caching implementation is hard. Still, maintenance is easy and is a great starting point. Most of the time, enterprise-level solution for applications uses automated cache libraries which will prevent over 80% of the problems, and, in some cases, exceed 90% of the cache-related problems.

Looking for a more complex application like PWAs (Progressive Web Applications) that are normally built on top of the same HTML, CSS, and JavaScript technologies analyzed on Evaluating the Impact of Caching on the Energy Consumption and Performance of Progressive Web Apps by Ivano Malavolta, Katerina Chinnappan, Lukas Jasmontas, Sarthak Gupta and Kaveh Ali Karam Soltany (ref. Malavolta *et al.*). Skipping the energy consumption analysis, we can benefit from its performance analysis of PWAs from a cache that we can dramatically benefit from caching strategies where we see load items dropping from over 4 seconds to almost 0 seconds, which, from the user point of view, is instant.

The reader might be asking why Redis, and why cache, and before we were starting digging into numbers, let us go first to see why Redis can improve our applications exponentially. Accordingly to Thiao Macedo and Fred Oliveira (ref. Macedo, and Oliveira) that wrote Redis Cookbook, we should be using Redis to optimize different kinds of applications, shown in their book for different languages and scenarios. If we go even further, at rails 5.2 implementation, they added Redis Store built in the Ruby on Rails implementation thought the ActiveStorage announced by "dhh" (ref. dhh) on rails development blog.

7.2 Tests methodology and enhancements proposal

By this time, the reader might already have guessed correctly; our proposed enhancements are related to cache. For this study, we are going to use Redis it as a cache server. Following Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis (ref. Chen *et al.*), we are using a similar configuration based on a five nodes cluster. The idea here is to simulate a Redis cluster with 1 master node, with 2 slave nodes and two replicas for consistency so we can achieve a much similar production level environment on our study. Each of the nodes gets 1000MB of RAM available to cache using the LRU (Least Recently Used)[2] strategy. If we look at Analysis of a Least Recently Used Cache Management

[2]Web Cache Page Replacement by Using LRU and LFU Algorithms with Hit Ratio: A Case Unification: http://cloud.politala.ac.id/politala/1.%20Jurusan/Teknik%20Informatika/19.%20e-journal/Jurnal%20Internasional%20TI/IJCSIT/Vol%205/ISSUE%203/ijcsit20140503122.pdf

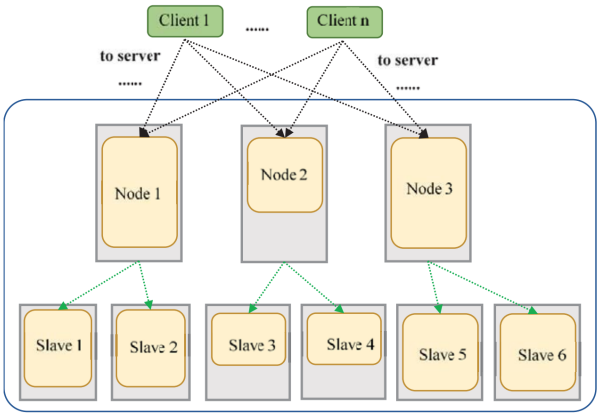


Figure 7.3: *Overview of a typical client-server structure for Redis distributed storage service from Shanshan Chen, Xiaoxin Tang, Hongwei Wang, Han Zhao and Minyi Guo (ref. Chen et al.).*

Policy for Web Browsers from Vijay S. Mookerjee and Yong Tan (ref. Mookerjee, and Tan) where they study this strategy for web browser caching, it perfectly fits our study case where we are displaying a commonly accessed page. This behavior happens a lot on our application. For example, on Consul, a platform for public debates is naturally expected to the page which lists the frequently accessed debates. Most of the users would be navigating from different debates and, frequently, accessing the same debates pages list to choose a new debate to participates in. The same logic applies to polls and the legislation process.

Talking specifically about Consul, but the concept fits most of the web applications. The majority of the page is composed of static content. And, by static, we are talking about content that does not have the necessity of being life; it could be cached for a few minutes if not an entire hour, or even in the cache for as long as the application stays alive, for example, caching assets like CSS and js files. But there are still important sections of Consul that need to be live, like the forum, the comment, and discussion section; people need to interact with each other in real-time for the discussion to be productive.

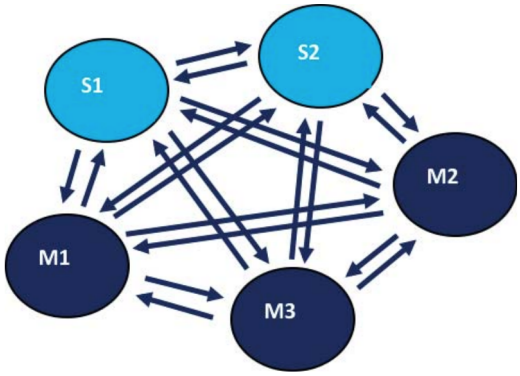


Figure 7.4: *A decentralized design for Redis by using Gossip protocol from Shanshan Chen, Xiaoxin Tang, Hongwei Wang, Han Zhao and Minyi Guo (ref. Chen et al.).*

These five static containers acting as a Redis cluster are delivered using the same host machine, using the same simulated cluster over Minikube, so, at or max capacity, we will have 26000MB and 8 cores dedicated to our entire cluster, including the rails application, the database, and the Redis cluster. If Redis it gets 5000MB at its peak capacity, it leaves Consul and the database to escalate until 21000MB of RAM. Of course, the reader might be asking what we saw earlier in this study that we lacked RAM to increase our performance,

that processing power is not the issue here, and we would need more memory. Still, we actually deploy less memory to the application and the database, so what is the problem? Counter-intuitively, by giving up 6000MB of memory to our Redis cluster, we are freeing our rails application of process power and data allocation and retrieval, which leads to an increase in responses, and page load times.

Our test strategy will be composed of 2 experiments, divided into the same categories and groups of previous studies over Consul's original code. First, we will cache only translations and i18n related queries and files; second, we will cache entire, frequently accessed pages, as previously mentioned. In a further section, the reader will notice, and we are talking about simple cache, our first strategy, and more detailed implemented cache, our second proposed analysis. The rest is simple; we are going to investigate different ranges of memory and CPU power, from 2 cores to 8 cores and from 1024MB RAM to 26000MB RAM; the only difference this time is that the cluster, and, consequently, host machine, will be shared with the application, database, and Redis cluster.

7.3 Explanatory analysis - Simple caching

7.3.1 2 cores and 8000MB RAM

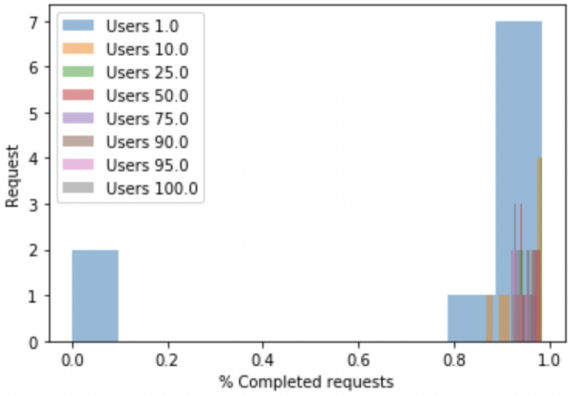


Figure 7.5: *Completed requests with 2 cores and 8gb RAM with ramp users accessing during 60 seconds.*

Let us stop here for a moment. We have a 2 cores cluster, only 9000MB dedicated to running the simulations, and ramp users access during 60 seconds. The reader might remember from the last chapter that we will focus on or analyze one to one hundred simultaneous users because, after that, we start to throttle the host machine. If we were to look at the whole range until 2000 users, we would STILL be trapped on the same problem as before, 80-90% of handled requests; let us keep a closer look at a smaller range of users, focus on 1 to 100. On the above chart figure7.5, we can clearly see that our requests keep inside a range of 86% to 100% requests handled; what is interesting is that with 50 simultaneous users, we can keep up to 94% requests handled, keeping our application on a 5% failure acceptance ratio, but remember, we are running with 9000MB and 2 cores only. Before we scale up to 4 cores, we need to confer the response time on such hard environment conditions.

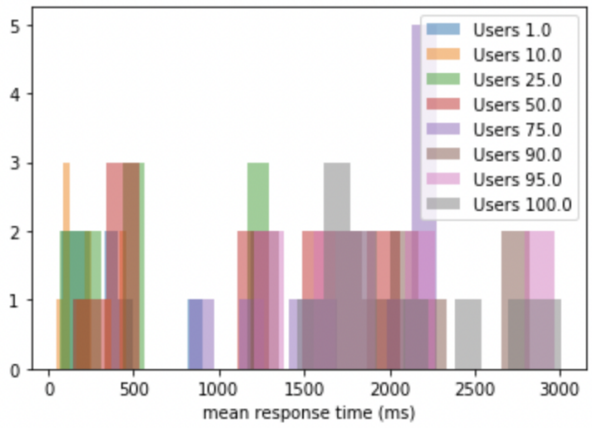


Figure 7.6: *Mean response time with 2 cores and 8gb RAM with ramp users accessing during 60 seconds.*

Looking at the mean response time on the figure 7.6, we see two different situations. The first one is that we got outlier 1 second response time measurements. Second, we got lower responses time because a smaller number of requests were handled. Still, the one that was got better responses time; in other words, the users who were lucky to get a request handled by the server was able to navigate faster, but the number of failures is large enough to make your user abandon the task, as we have seen previously on "how long a user waits for a website." This is a clear example of a throttle system so the reader could get a comparison base for core improvements that may affect our performance. Our system clearly benefits from simple caching strategies, but we still can not fully picture the benefits and downsides because our cluster is throttling. Let us scale up the processing power to 4 cores, still keeping the 9000MB of RAM, to see how it affects our performance.

7.3.2 4 cores and 8000MB RAM

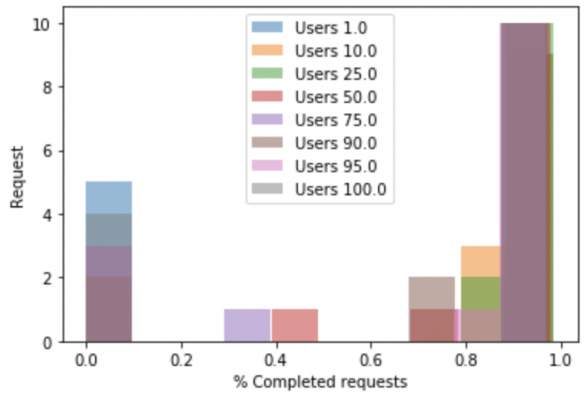


Figure 7.7: *Completed requests with 4 cores and 8gb RAM with ramp users accessing during 60 seconds.*

We just doubled the cluster process power from 2 to 4 cores (still half of our total size), and we got 96% handled requests for 50 simultaneous users for 60 seconds. Jumped from 86%-94% almost uniformly distributed for 100 simultaneous users to 94% completed requests with a small fraction of the requests distributed on a range of 86%-94%. The most interesting part is that we are at 9000MB RAM, and or Redis cluster uses up to 6000MB of that same

memory. We need to keep investigating, but the first signal is that relieving process power and dedicating some memory for caching dramatically improve the application's performance. As our previous research has shown us, a cache is indeed a great tool to improve performance, and, as mentioned before, we are trying to keep it simple to avoid caching bugs or cause more problems then solves which the number one problem when we are talking about cache, as our previous research have shown us, caching adding another level of complexity to an application that needs to be managed by developers.

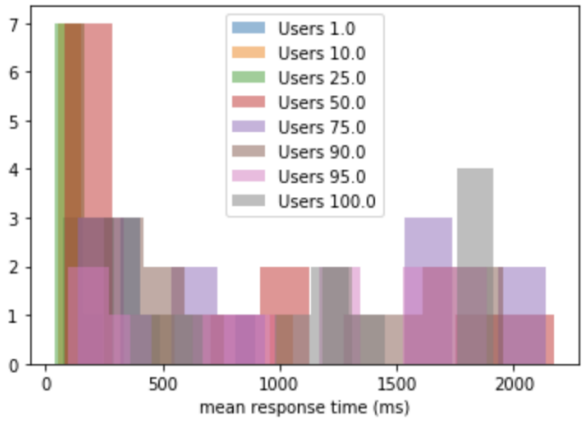


Figure 7.8: *Mean response time with 4 cores and 8gb RAM with ramp users accessing during 60 seconds.*

As seen before, we have got a great overall performance with 4 cores for 100 simultaneous users. In the worst scenario, we could lower our max mean page load time to 3000ms, which is 2000ms lower than the previous iteration with 2 cores, as the reader can see in the figure 7.8. That is time to double to process power again and see what happens. But look what is happening here, except for the minimal amount of users, like below 25. We see almost a normal distribution here, as the number of users increases during a small period of time, accessing and using the same resources, cache stat acting. We got the performance improvements we were talking about, and we just hit the 3 seconds page load time that was quoted from Google's research early on this study for 100 simultaneous users.

Before we jump into our top process power capacity, let us look at the first half of the chart figure 7.8, considering that RAM is our main issue here. We are on a small cluster, we were able to deliver great response time up to 25 simultaneous users, keep all of then under 1000ms and the reader might want to look again at the figure;7.7, we were able to deliver such good response time to 99% of those users which is as perfect as it is possible to if we take into considerations occasional failure due he network itself.

Ok, we are sure the improvements are great, but what have we achieved so far? How better it is from 2 cores? Did we get actual improvements Doubling of performance? This is a simple question; let us compare both plots. First, let us look at figures 7.7 and 7.7show us the completed performance, and we can see that the performance keeps static at 98%-99%, but what about response time? Looking for figures 7.8 and 7.8this time we see great improvements, jumping from the mean response time of 3000ms to as low as 500ms mean response time, which is a 6-time improvement in performance.

7.3.3 8 cores and 8000MB RAM

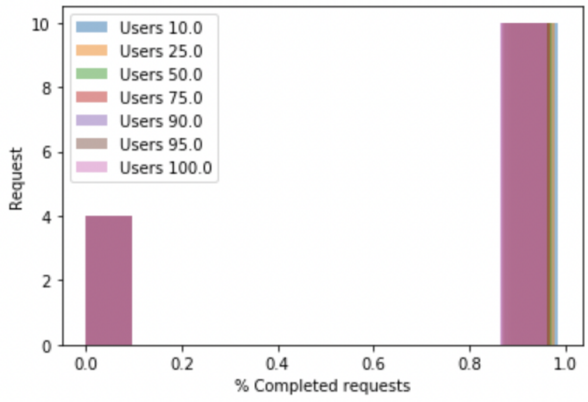


Figure 7.9: *Completed requests with 8 cores and 8gb RAM with ramp users accessing during 1 second.*

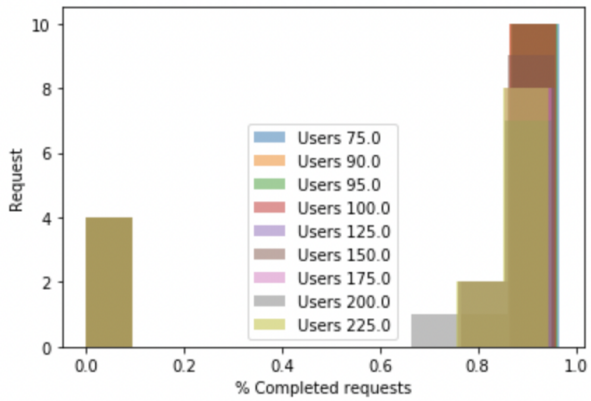


Figure 7.10: *Completed requests with 8 cores and 8gb RAM with ramp users accessing during 10 seconds.*

This is our top process power capacity. We will be increasing only the RAM to investigate its performance, which is also, until now, what is most affecting response time counteracting cores power, which improves the number of requests we can handle. Now, look at the figure 7.9, but it makes it a little hard to analyze such a plot, so let us concentrate on the figure 7.10. Now that we completed the group with the outlier, we can see that we could handle most of the requests at almost 92% of them for 100 simultaneous users. Interestingly, with up to 75 simultaneous users, we were able to deliver over 95% of the requests. It is about 1 second time period, just as a comparison base for further analysis with increasing memory.

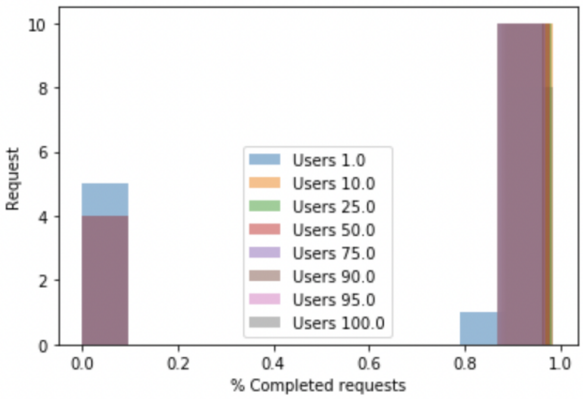


Figure 7.11: *Completed requests with 8 cores and 8gb RAM with ramp users accessing during 10 seconds.*

In the figure, 7.11we see a plot for ram user access during 10 seconds, and on the figure,7.12 the same tests but for a period of 60 seconds. We can clearly see that for both plots, figures 7.11 and 7.12 we handled over 90% of the requests for most users, and that increase the time duration increase from 1 second to 10, and then 60 seconds have shifted our comparison point slightly. As the readers might want to check for themselves, comparing both plots, we see a more concentrated number of users over 90% when we look at the 60 seconds chart. Looking for those 3 plots, we see that, as expected, when we increase our time range, we can accommodate more simultaneous users, and it is not a process burst power needed o a shot period of time. Still, a well-distributed range of users accesses a long period o time.

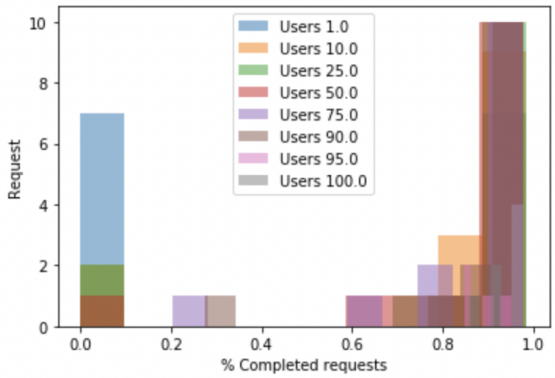


Figure 7.12: *Completed requests with 8 cores and 8gb RAM with ramp users accessing during 60 seconds.*

The problem with those burst access like on figure 7.11 is that simulating 100 users accessing a web application during 10 seconds is the same as 600 users accessing the same application during 60 seconds and see this heavy payload on our servers that the just dropped our capacity of handling simultaneous users at the tax rate of over 99% to 75 users to less than 95% to 25 users, and that is the same reason we have a wider group of requests closer to 82%. If we were to consider the plot on the figure,7.9 we could clearly see our outlier metrics justified for such a short period of time since we are simulating all those users accessing the application at the same second, removing it, and looking at the plot on the figure 7.10 we clearly see one-second burst access break a little bit more requests but the

overall performance, when talking about the number of handled requests, keeps the same, but the performance related to responses times drops in fast.

But before we jump into response time, let us remember the plots on figures 7.5, 7.7, and 7.12 which representing the number of completed requests for 2, 4, and 8 cores, respectively. We jumped from poor numbers related to handled requests and in some cases handling only 65% of them, but all the way up to 8 cores we were able to tweak those number into higher percentages, going up to 92% of the users for 90 simultaneous users and over 86% for our ideal goal of 100 users.

Now let us talk about response time and how those are affected by an increase in the time range, from 1 to 60 seconds and, of course, comparing with 4 and 2 cores. We will focus on the plots on figures 7.13, 7.14, and 7.15 which measure response times for ramp user access during 1, 10, and 60 seconds.

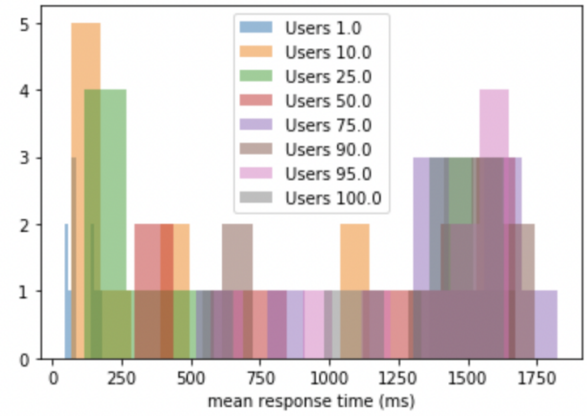


Figure 7.13: *Mean response time with 8 cores and 8gb RAM with ramp users accessing during 1 second.*

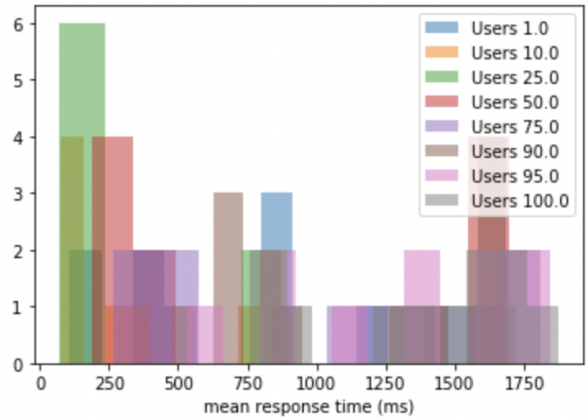


Figure 7.14: *Mean response time with 8 cores and 8gb RAM with ramp users accessing during 10 seconds.*

As e have seen before, lower our time interval has a small impact on the number of requests handled and, as the reader is about to check for his own here, those timestamps get a small impact, as well, on response time measurements. Start looking for the plot,7.13 we see pretty poor performance, response time going up to 2500ms distributed all over the spectrum with lower amounts of simultaneous users getting better response time, and as we increase the number of simultaneous users, those same requests increase their duration to over 2000ms and almost 2500ms.

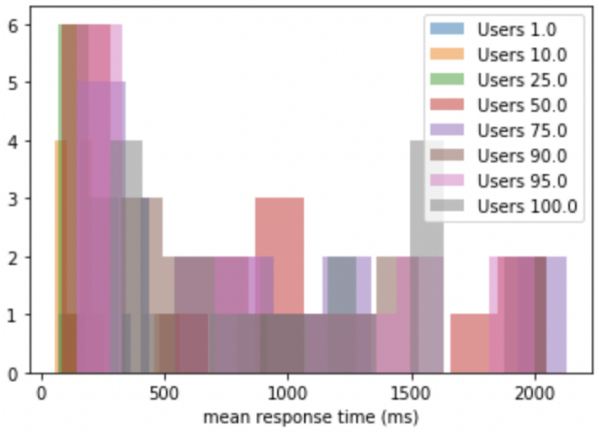


Figure 7.15: *Mean response time with 8 cores and 8gb RAM with ramp users accessing during 60 seconds.*

Before we make another performance level jump, the plot 7.15 tips us of the behavior we saw before. If Consul original code, we reach a level we see a softening effect where our cluster starts to fit our users better. We start seeing more uniformly distributed access plots. From the last plot we see we can keep 50 simultaneous users under 1000ms, 75 users under 1500ms, and, of course, keep all those 100 users under Google's 3-second rule of thumb mentioned here many times. However, we are racing to under 1000ms performance, and we still can triple our memory capacity.

Keep on our exercise of backtracking, look at figures 7.6, 7.8, and 7.15, for 2, 4, and 8 cores respectively we see that we start with a peak performance under 5000ms, which is good but was not good enough. When we jumped to 4 cores, we lowered those numbers by the entire 2000ms, going to 3000ms at worst performance, meaning we improve our response time by 60%. Then we doubled again, this time to 8 cores, but, now our performance just lowered to 2500ms at the worst-case scenario, and again, as before, those last stage of performance seems to be the hardest to achieve, remembering me of Pareto principle, where the last 20% of performance improvements will take 80% of the effort.

7.3.4 8 cores and 16000MB RAM

We are almost reaching our host machine top capacity for simulating our cluster. Let us jump into 16000MB of RAM and 8 cores with 5 nodes Redis cluster. Until now, the whole analysis was focused on process power and the number of cores; in our simulation scenarios, cluster nodes, as known as each core simulating a single core machine running one or multiple containers. As the reader might remember, we could see some response time improvement as we increase the processing power. However, it is still important to remember the processing power is mostly responsible for the number of completed requests; the RAM is responsible for response times. having said that, we are now going to shift our analysis's target by seeing more improvements in response times than the number of completed requests.

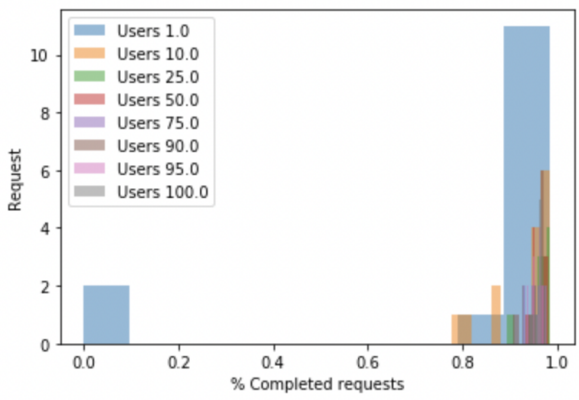


Figure 7.16: *Completed requests with 8 cores and 16gb RAM with ramp users accessing during 60 seconds.*

Now let us look at the figure;7.16, we have a great overall performance of up to 90 simultaneous users to deliver over 95% of the requests. The interesting part is that we see most of 100 users distribution over 94% requests ratio with its peak concentration at 97%. Still, we can clearly see a small fraction of our requests between 80% and 85%, which is intriguing and need further investigation.

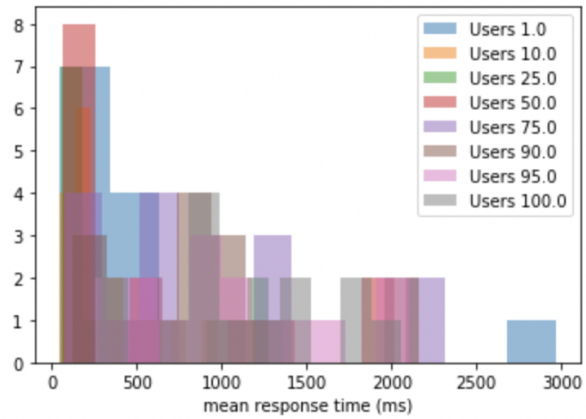


Figure 7.17: *Mean response time with 8 cores and 16gb RAM with ramp users accessing during 60 seconds.*

On the other hand, focusing on mean response time, on the figure,7.17 we see that in the worst-case scenario, we are still on 2500ms. Comparing the results we see here to the ones shown on the plot 7.15 immediately brings to our attention that the performance did not increase. What is interesting that we see here is that we double the memory. Still, the chart just bounced around its mean; it is like the extra 8000MB of memory could ear us just a few milliseconds, and its event 1% shift in completed requests improvements. It is like we are reaching some kind of roof where we can go through, which is a signal that we are throttling or need much more power to gain those few extra percentages.

Before we make another performance jump, we want to take a closer look at plots 7.17 and 7.15, comparing each group of users. Starting from 10 simultaneous users, we see that the distribution kept almost the same, under 500ms but with a small group around 800ms. At 25 users, we can keep under 500ms witch is the same as the 1000ms limit for the plot 7.15. Going up to 50 users is our limit under 2 seconds limit, but we still see the same pattern repeatedly. With 75 simultaneous users, things start to change, we still keep all requests under 2000ms

with 16000MB of RAM, but at 8000MB, we see a slight fraction of the requests above the 2000ms threshold. Over 90 users start to face problems, the distribution range goes from 1800ms to 2500ms on both plots, but they are more uniformly distributed in the figure 7.15.

7.3.5 8 cores and 26000MB RAM

Finally, here we are, 2600MB RAM and 8 cores with 5 nodes Redis cluster with 6000MB RAM, this is our peak cluster performance, so our analysis on basic cache reaches our best scenario. Further in this study, the reader will encounter an analysis of advanced caching where we will cache specifics parts, pages, or routes of the application and compare the performance gains versus the increase in complexity add to this granularity cache strategy.

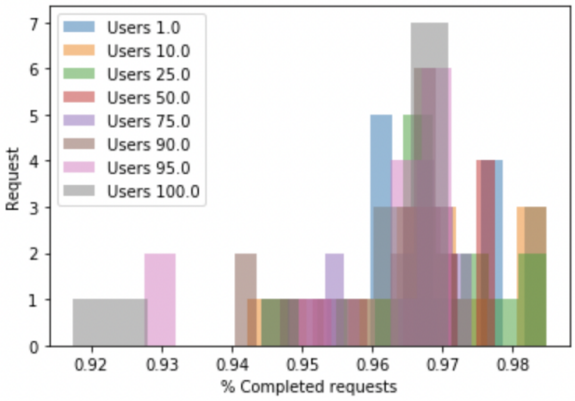


Figure 7.18: *Completed requests with 8 cores and 26gb RAM with ramp users accessing during 60 seconds.*

Let us start our analysis with a plot 7.18 and compare it with our previous results from plots 7.16, and 7.12. looking at the above, we see improvements, but soft ones. Let me explain; looking at the figure 7.18 we see that for up to 50, we handled 100% of the requests; at 75 users, we see some outliers around 94% to 98% of the requests; we see a pretty interesting thing here, our target group, 100 simultaneous users, just shift around 95%, entering our 5% acceptance ratio.

As we can see, different from the last step where we doubled the RAM amount from 8000MB to 1600MB, this time, increase another 10000MB of RAM we can handle all those users, we still have some outliers but at 100 simultaneous users, we tests are doing over 26000 requests during this time period which make it practically impossible not to have any timeouts of outliers requests due to multiple factors like routing.

The difference that we see here in the figure 7.18 is small. Before, on the figure 7.16, and 7.12 we saw a significant amount of request around 80% to 90% percent, and doubling the memory power, as we have seen before, was not enough to show us great performance improvements, which was rather frustrating. Still, here we see just a small fraction of 100 user distribution landing on lower requests, and we can safely assume us on inside our safe limits. Those requests out the cluster around 95% are outliers, and we will see further investigation on this outlier behavior in the next section.

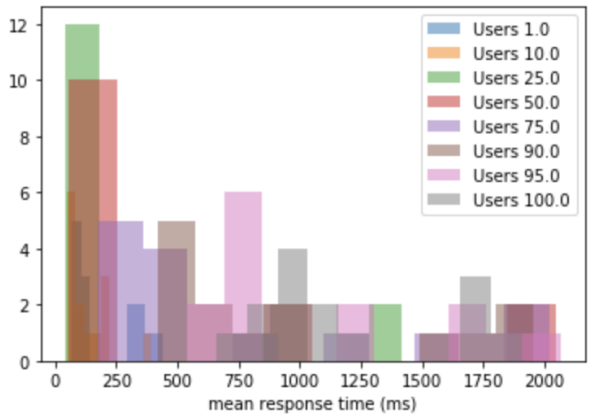


Figure 7.19: *Mean response time with 8 cores and 26gb RAM with ramp users accessing during 60 seconds.*

We need to take a closer look at the mean response chart is plotted in the figure 7.19. Starting from 10 users, here we see no much difference between 26000MB (fig. 7.19), 16000MB (fig. 7.17), and 8000MB (fig. 7.15). The same behavior applies to 25 simultaneous users, which leads us to one conclusion, to those levels we reach peak performance and no matter how we increase our cluster we will still see those same results. The interesting is that we keep seeing some outliers where, even at 50 simultaneous users, we can see some requests lasting longer than 1500ms, which is an incredibly high wait time for such a small number of users with that processing power, which also explains why at 100 simultaneous users we see some requests seems not being handled.

At 75 users, we see slight improvements, our mean response time distribution peaks migrate from 1000ms-11000ms to 900ms, but as seen before, it is not a great difference and could be due to small environmental changes. On 95 users, we keep on the same behavior across the 3 levels of RAM, 7.15, 7.17, and 7.19. Going to the border of our study case, at 90, 95, and 100 users, we face the same problem as we did on the section before with 16000MB of RAM; the increased amount is not large enough to allow us to see bigger differences here. Still, it shows us that to increase users' last gap; we have to increase the effort by doing a more granular cache strategy.

If it is worth adding another level of complexity to the application by doing manual caching for partials and pages, routes, or requests, which is the main problem of caching? This is what we will be investigating in section 7.4. Until now ,we were trying to keep things simple and avoid adding another layer, which will certainly bring more bugs to our application and add more needing maintenance, so we kept only automatic caching and translations caching. But before we advance on manual caching, let us take a step back and look from afar into our simulations. We need to get an overall understanding of these experiments and this time focusing on the global conditions.

7.3.6 Test analysis

Before we jump into a general analysis of our case scenario, we would like to use the plot 7.20 and 7.21 to demonstrate why focus on simulations with a higher number of users is of no need for us in this study. First, let us look at the plot 7.21 we can handle the request pretty well, all of the tested groups keep under 3 seconds limit set by Google that we already mentioned. But, if we pay attention to the plot 7.20 we can spot the problem right on the way. Our request handled peak is 80% for 500 users, increasing all the way up to 2000 simultaneous users make our requests rate drops to as lower as 20%-30% which is

even less than 1000 simultaneous users rate, which is between 50% and 70%. The problem here is that even on 500 simultaneous users our acceptance ratio is pretty low, we can not miss one out of 5 requests.

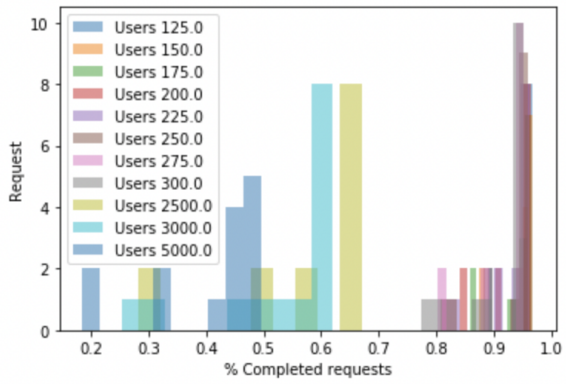


Figure 7.20: *Completed requests time with 8 cores, 26624mb RAM for 60 seconds for over 100 simultaneous users.*

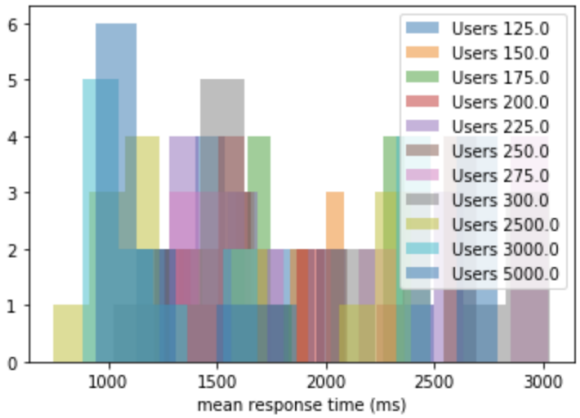


Figure 7.21: *Mean response time time with 8 cores, 26624mb RAM for 60 seconds for over 100 simultaneous users.*

The reader might have noticed that we had brought to our attention that we had outliers numerous times during these plots, now we need to take a closer look into these outliers and see how sparse is these experiments. And here we will divide into 4 groups, percentile 50%, 75%, 95%, and 99%. Let us start our analysis with the 50% percentile and go all the way up to 99%.

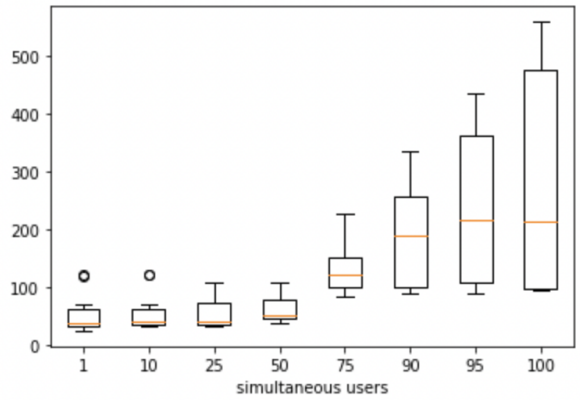


Figure 7.22: *Response time in milliseconds for 50% requests.*

If we pay attention to the figure 7.22 we can see that as we increase the number of simultaneous users, and, consequently, the raw amount of requests shooted through the server, the number of outliers increase, and, like we have seen on other plots (figures: 7.19, 7.17, 7.15, 7.8, and 7.6) we can clearly see the distribution clustering over 90% but some request close to 80% e that is out outliers that we were talking previous on this study, as we were guessing before, now we are sure that this measurement could be excluded focusing on the correct results.

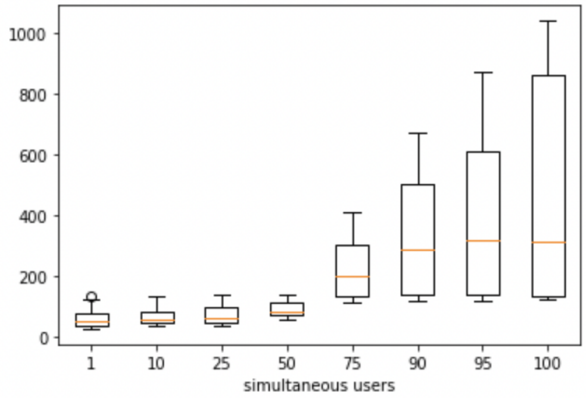


Figure 7.23: *Response time in milliseconds for 75% requests.*

The plot 7.23shows us exactly the problem we saw in this whole analysis, and at the same time, proves to the reader what we have been discussing. We reached 100 simultaneous users with over 95% completed requests ratio, but there are outliers, and there will always be, which is why we have seen strange results across our plots. But as we decrease the interval width, we start seeing a different behavior.

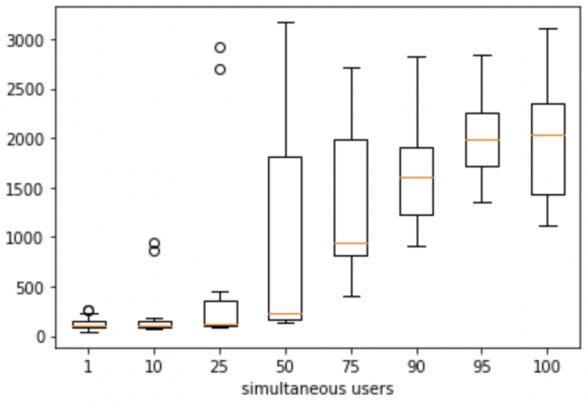


Figure 7.24: *Response time in milliseconds for 95% requests.*

We need to stop here on the plot 7.24. Differently from 7.23 and 7.22 we see here on 7.24 outlier all the way up to 95 simultaneous users. Still, at 100 simultaneous users, we see a bigger interval but no outliers. Of course, we exclude 1 simultaneous user since that is not much data to be analyzed. if the reader takes a closer look at this last boxplot, we see that the upper half is sparse precisely to the median line. In contrast, the lower half is denser, backed up by outliers only on the upper half (which represents higher response times). This confirms that our small percentages of not completed requests have higher response times is due to outliers on the HTTP interface.

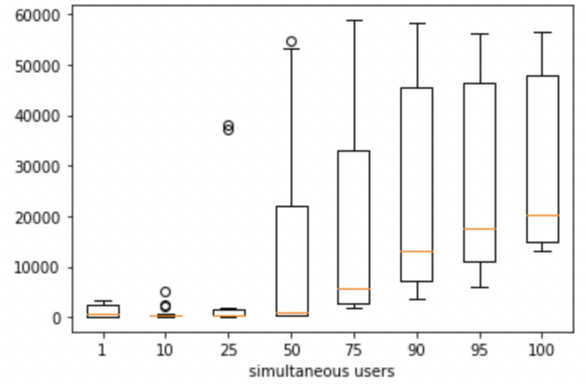


Figure 7.25: *Response time in milliseconds for 99% requests.*

We can analyze the 95 and the 99 percentiles together as the behavior is similar. In the figure 7.25 we start to see fewer outliers as the length of the interval has sort. But, the interesting part here is to notice how the response time (y-axis) has increased. On the plot 7.22, for 50% of the requests only, the response time was 800ms. As we increase to 75%, the response time increased to 1200, which is still excellent. Going to 95%, things start to deteriorate, response time, counting the outliers is up to 3500ms, and without then, it would be 5000ms, for 100 users. And finally, for 99%, we see that the response time is up to 60000ms, or 60 seconds, which is incredibly high, and this behavior happens even on 25 simultaneous users. If the reader could pay attention to 10 simultaneous users on the figure 7.25, the reader might notice that even on 10 simultaneous users, we see responses time of over 20000ms (on outliers), and this give us a hint that no matter how big our cluster be, it will always be some requests that will break of taking longer responses time, and that is due to our protocol, we can not do anything about it.

At this point, data become very dense, so we are going to divide into 4 groups, again, at the same 4 groups, 50%, 75%, 95%, and 99% percentiles plots, printed out on figures 7.26, 7.27, 7.28, and 7.29 respectively. Let us begin with the plot7.26.

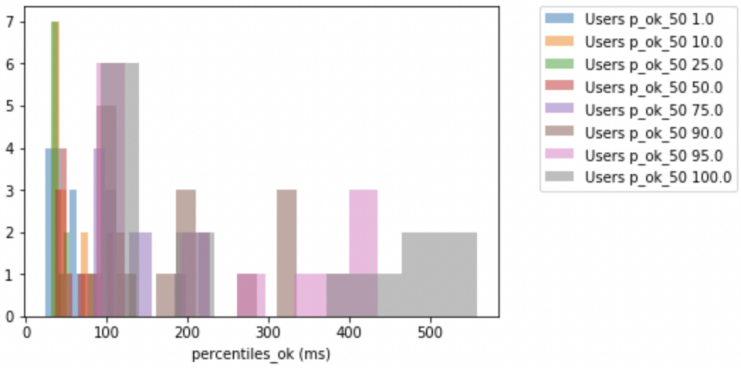


Figure 7.26: *Completed response time histogram for percentile 50%.*

If we focus on the plot, 7.26we can see the same behavior we spotted earlier and confirmed our guesses that there is a small percentage of outliers on the requests. We can see that most of the requests for 100 simultaneous users land on 200ms, some on 400ms, and our outliers, about 800ms. The interesting part is that we see this behavior for 95, 90, and 75 users, look for the upper half of the plot labels. If we look to the lower half, from 10 to 50 users, the results are clustered around 100ms to 200ms, but the outliers for from 300ms to almost 500ms.

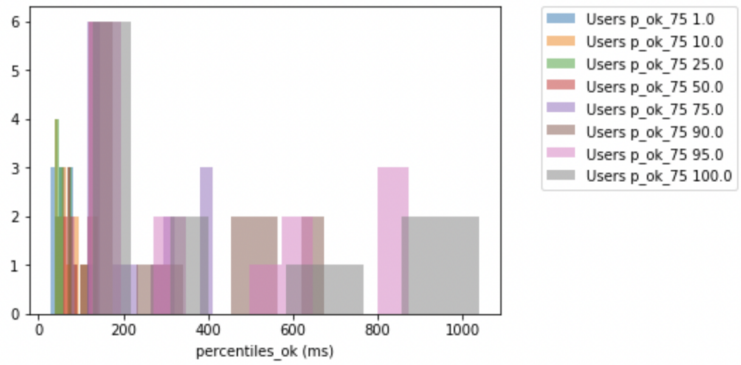


Figure 7.27: *Completed response time histogram for percentile 75%.*

As we have seen on the boxplots earlier on the figure 7.22 and 7.23now on figures 7.27 and 7.26now with response time on horizontal axes. As the reader might have already spotted, for 100 users, up to 75% of the requests we handle it with an excellent response time of 100ms to 200ms, and even the outliers only increase those numbers up to 800ms.

If the reader remembers from our previous research, we should never take more than 5 seconds to load a web page because we could lose SEO ranking, and the user might leave the task behind. And here, for the 95% percentile, we just hit this limit and increase it by 40%, increase our maximum response time to almost 3500ms, which is 500ms over our desired limit.

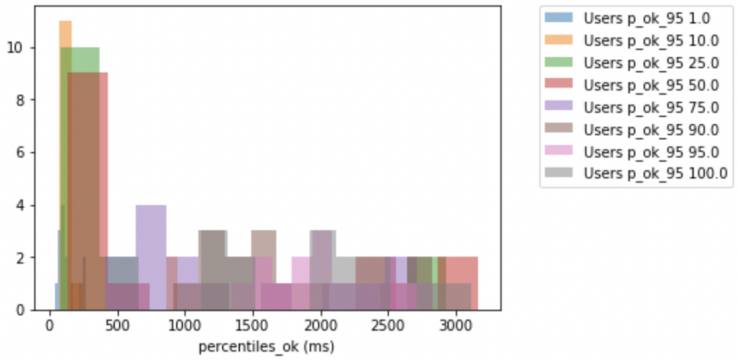


Figure 7.28: *Completed response time histogram for percentile 95%.*

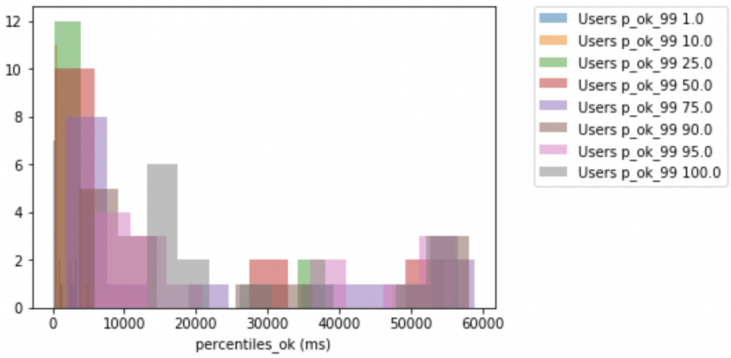


Figure 7.29: *Completed response time histogram for percentile 99%.*

Finally, if we pay attention to the plot 7.29, differently from all other plots, to handle 99% of the requests, we have to increase our cluster resource on an impracticable level, it is just not worth it. If we remember together, we are with 8 cores, 26000MB RAM, and citeRedis cluster disposal of our application. For over 10 simultaneous users, the response time is higher enough to make our website got a low score on the SEO index and make all of our users tired of using the service. Think about the Pareto principle, this last 1% would be worth only for big companies in technology industries, but not what most of the enterprise-level applications would have the technical or financial capacity to handle.

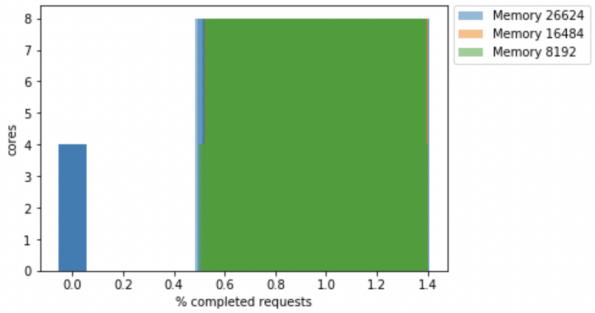


Figure 7.30: *Number of completed request x number of cores.*

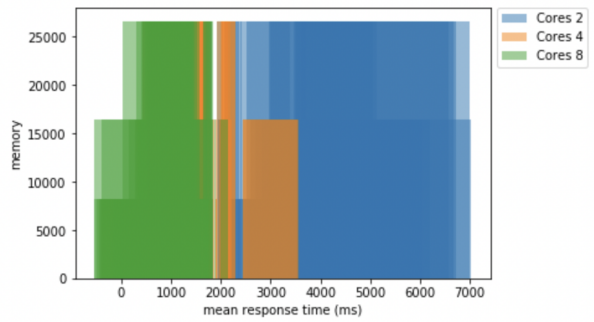


Figure 7.31: *Mean response time for completed requests x memory amount.*

Before we finish this section, we would like to look at the plots' number of completed requests and mean response time variations according to cores and RAM 7.30. As we can see from the above plots, as we increase memory and the number of cores, we reach 100% requests. On the other hand, focusing on the plot 7.31, we can see how response time decreases over resources increase. The plot 7.31 clearly shows us that around 8 cores, response time stick around 2000ms, against initial 6000ms-8000ms for 2 cores. Talking about process power and cores, we can better understate its effects by looking at the plot 7.30 where we can clearly see 1024MB of RAM running 0 requests, but as we increase memory, we reach over 90% requests. Similarly, we can spot for 8 cores, numbers of requests being dramatically impacted by RAM. Now, let us jump into advanced caching, where we will be using partials and page caching to identify if the benefits are worth the increase in complexity.

Finally, we would like to point out to 7.32 hat memory is the main responsible for lowering the mean response time. We second of that would be the replica set; as we increase the number of replicas, we cannot decrease mean response time and increase the number of handled requests. Following the number of handled requests, cores play an important role in memory, but memory is definitely one of the main ones responsible for performance.

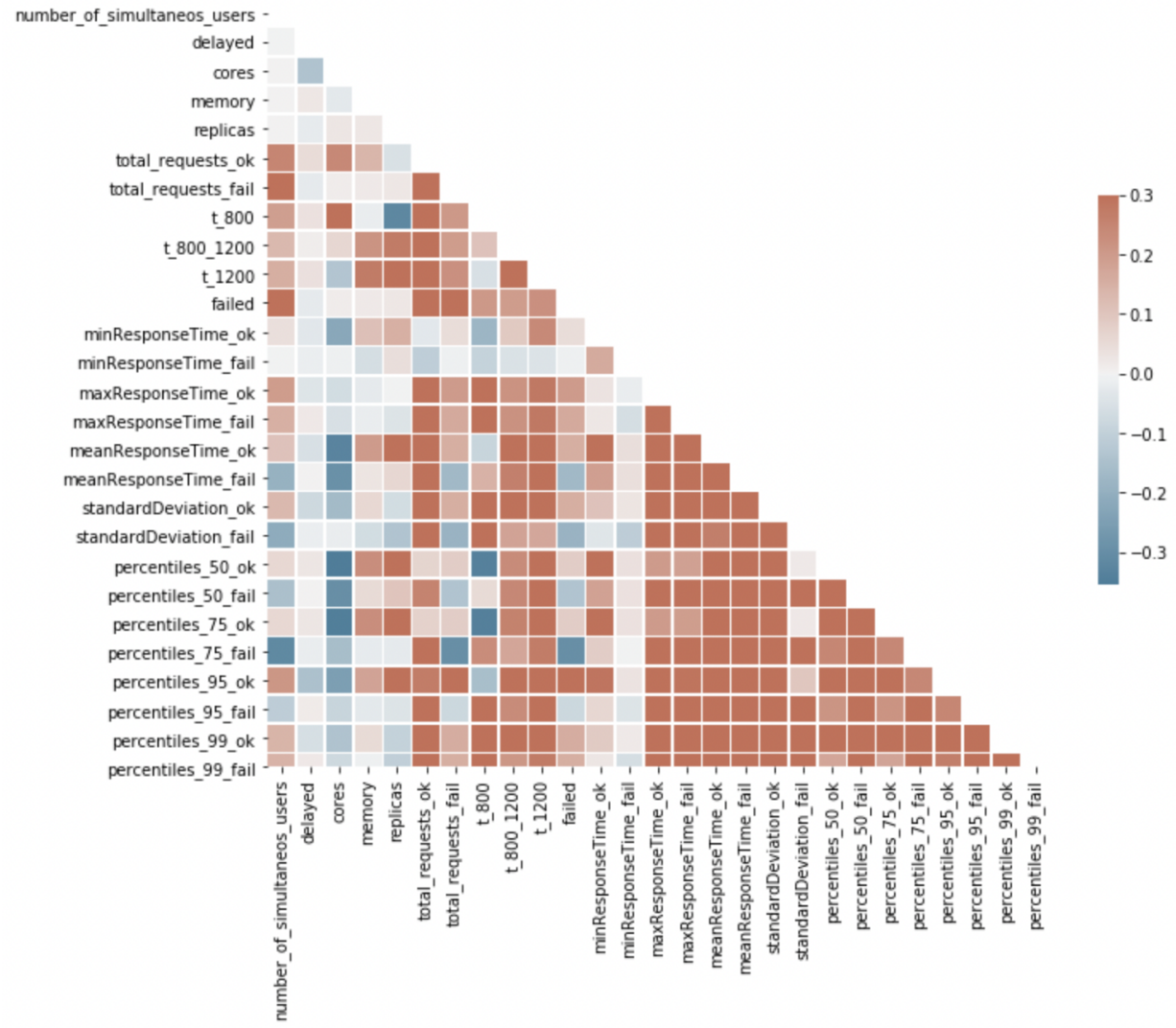


Figure 7.32: *Correalation heatmap.*

7.4 Explanatory analysis - Advanced caching

7.4.1 2 cores and 8000MB RAM

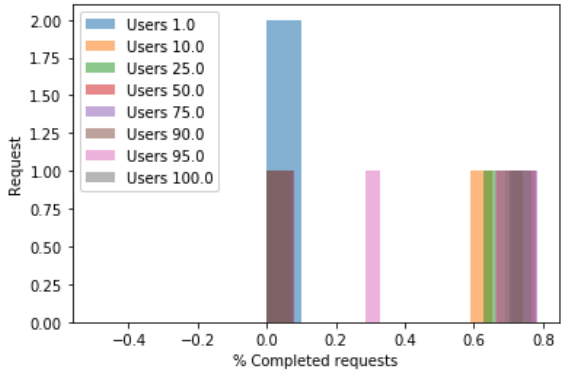


Figure 7.33: *Mean response time with 2 cores and 9gb RAM with ramp users accessing during 60 seconds.*

Accordingly to what we have been studying so far, we will focus this part of the analysis on 1 to 100 simultaneous users. Unlike the last section, where we relied on rails gem's automatic caching to make things simpler a bug-free, we will use advance caching strategies where we select specific parts of the code in the cache. This section will use a cache of partials of views, query results, and compare to the last section results and hope to discover if it is worth increasing software complexity is given performance improvements.

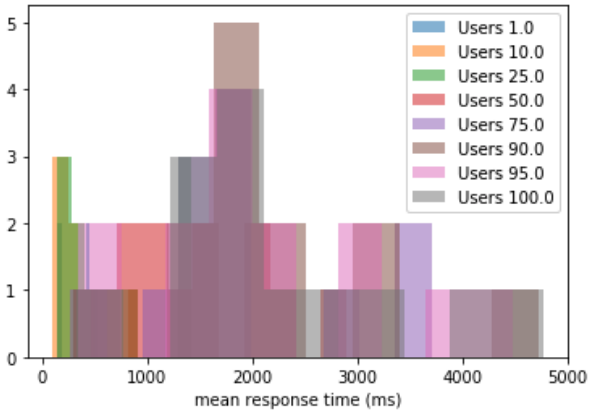


Figure 7.34: *Mean response time with 2 cores and 9gb RAM with ramp users accessing during 60 seconds.*

If we start by looking at the plot,7.33 we see that for 50 users, we can handle over 94% of the requests; for 25 users, this number increase by up to 98%, while 100 users keep between 86% and 94%. If the reader remembers, from the last section, on plot 7.5, we see many similar results, where the number of completed requests keeps very close to each other. Talking about response time, we should compare 7.6 and 7.35, and here we see something interesting. Here, with a more complex cache scenario, we see a higher response time than previous tests. With the increased overhead to cache given the number of requests, we could not see benefits from the requests, and with a lack of resources, we end up with less performance. And this shows us in some environments, it is not ideal for increasing cache complexity.

7.4.2 4 cores and 8000MB RAM

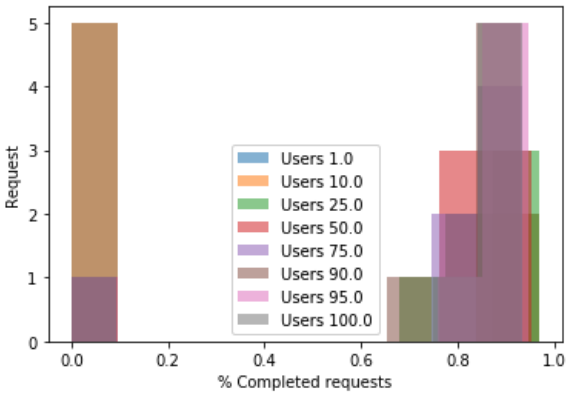


Figure 7.35: *Mean response time with 4 cores and 9gb RAM with ramp users accessing during 60 seconds.*

With 4 cores, we are still far behind ideal process power, and as we can see on the plot 7.35 and 7.7 that, and keep on the same complete requests interval. If we compare to previous experiments with 2 cores, we see a slight shift in requests interval were on 100 users, outliers requests spot between 88% and most of the requests keeps between 92% and 96% requests, for 50 users, the number of completed requests increase up to 98%, a small improvement.

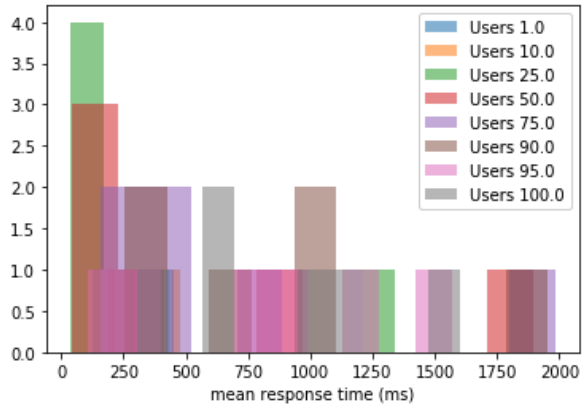


Figure 7.36: *Mean response time with 4 cores and 9gb RAM with ramp users accessing during 60 seconds.*

If we pay attention to the plot,7.36 we see the same interesting behavior. We do not have enough resources to be worth advancing, and we actually see the worst performance results. Unlike previous tests, on plot 7.34, doubling process power makes us able to decrease response time peak to 3000ms, but it is still 1000ms bigger than previous experiments with basic caching.

7.4.3 8 cores and 8000MB RAM

Unlike the previous analysis, now we are at our peak process power capacity on our simulated cluster. Let us start our analysis by comparing the results here and the ones from the previous section.

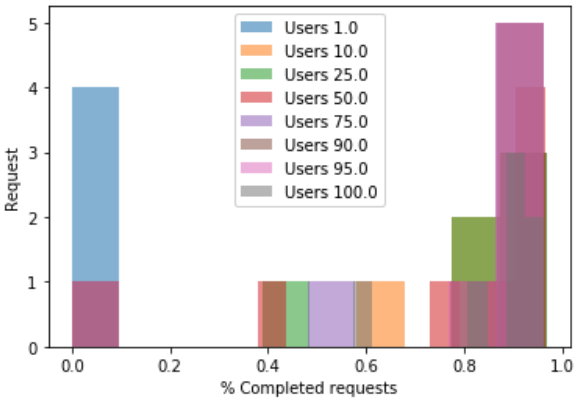


Figure 7.37: *Mean response time with 8 cores and 9gb RAM with ramp users accessing during 60 seconds.*

Now stop, let us stop here a moment and look at the plot 7.37 compared to 7.12. We see similar results on both experiments, which suggest that we need a requirement of at

least 8 cores to even the results we see on simpler caching strategies, up to 50 simultaneous users there a 98% completed quest rate. If we do not consider the outliers, requests for 100 simultaneous users are over 94% rate.

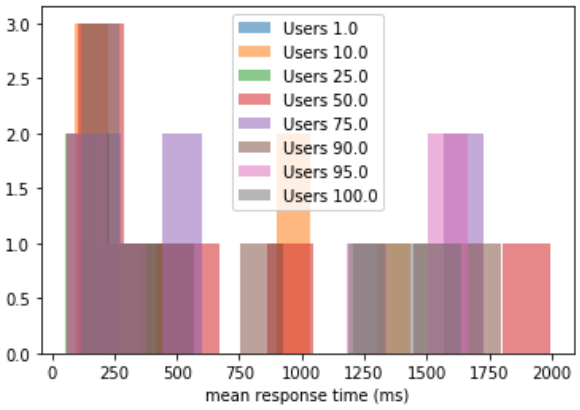


Figure 7.38: *Mean response time with 8 cores and 9gb RAM with ramp users accessing during 60 seconds.*

Plot 7.38 make it even clearer to understanding the outliers, here we see requests for 1 simultaneous user taking over 2500ms, and this is due to caching all the content on the first request, and as we have a lot of more requests to cache, we see an outlier of over 2500ms. On the other hand, by looking at our plot and excluding the outliers, we see the same 2000ms response time for all the way up to 100 users, which is similar7.15. Let us make another jump, this time into 16000MB RAM.

7.4.4 8 cores and 16000MB RAM

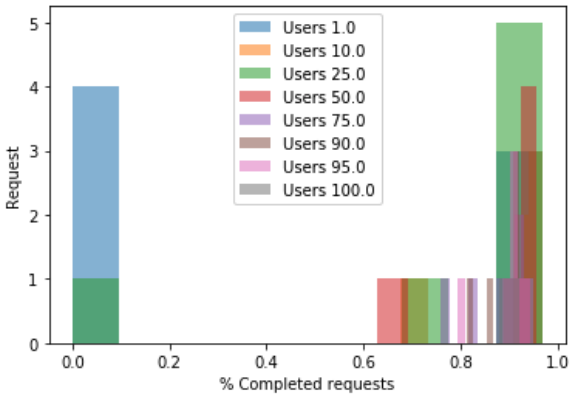


Figure 7.39: *Mean response time with 8 cores and 16gb RAM with ramp users accessing during 60 seconds.*

Following the behavior we have been following on previous experiments, the percentage of completed responses on the plot 7.39, excluding outliers, keep above 94% for all users. If we compare to the plot 7.16 we finally see an improvement of like 1%-2%, which is frustrating, give the wide increase in complexity on this cache strategy. According to our previous research, the problem here is that the cache has been flagged as one of the main reasons behind new bug sources, and a 1% to 2% increase in the number of responses time does not seem like a scalable solution.

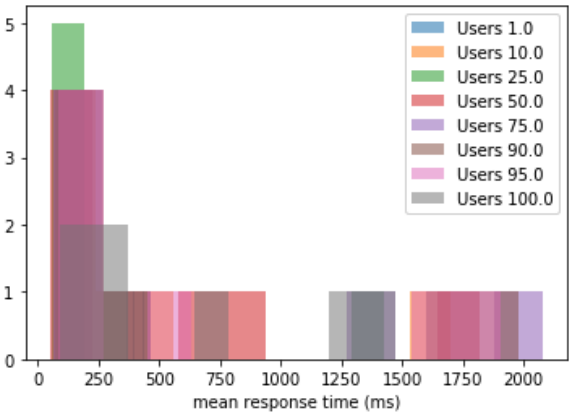


Figure 7.40: *Mean response time with 8 cores and 16gb RAM with ramp users accessing during 60 seconds.*

By looking into the percentage of completed requests, we showed the reader that it is not worth the complexity of advanced caching when talking about the completed request number. Let us see on mean response time, start by comparing the plot,7.40 and 7.17 we see an interesting behavior; on the plot,7.17 we see the peak response time of 2500ms, and now, on the plot 7.40, the peak response time is 2000ms, which is against what we have seen so far. But the interesting here is that there are no significant improvements in mean response time; add that to the fact that there is only 1% to 2% improvement in the percentage of completed requests, we start to see signs that advance caching simply is not worth. Let us jump into our peak performance at 8 cores and 26000MB RAM to finish this analysis.

7.4.5 8 cores and 26000MB RAM

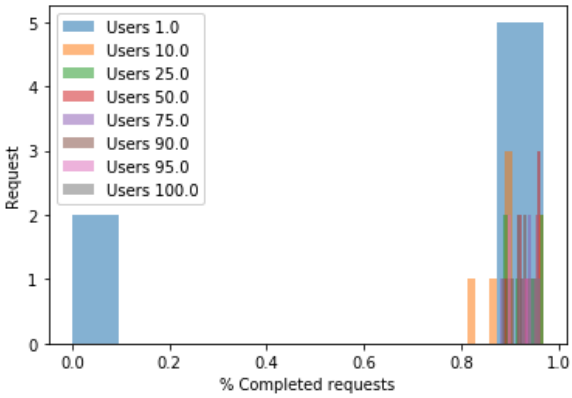


Figure 7.41: *Mean response time with 8 cores and 26gb RAM with ramp users accessing during 60 seconds.*

Finally, here we are, our peak capacity, 8 cores, 26000MB, and advanced cache; this is the last step of our research. Now we will focus on 7.41 and compare against 7.18 to discover ones for all if it is worth advance caching since we already saw that the advanced-cache could be dangerous in small environments. Starting from the plot,7.41 we see the same behavior 7.18 where there is a small percentage of outliers, and the number of completed requests keeps above 96%, and for smaller batches of users, they keep around 98% and 100%, nothing new.

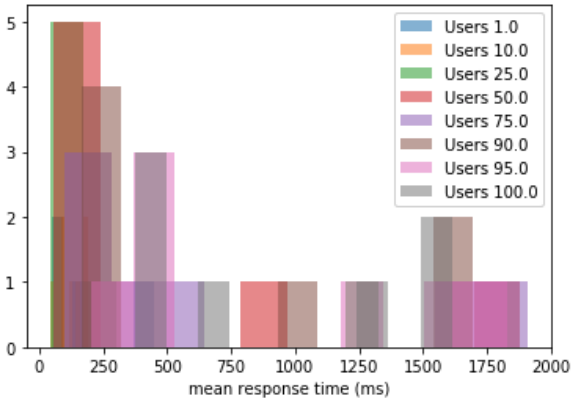


Figure 7.42: *Mean response time with 8 cores and 26gb RAM with ramp users accessing during 60 seconds.*

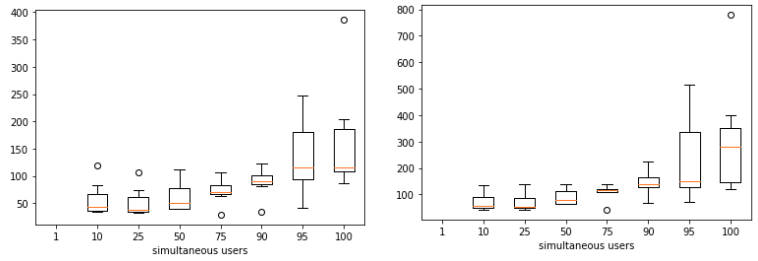
Skipping to mean response time, let us take a look at 7.42 and compare against 7.19. Now on response time, we see a small difference. Before we saw a sparse distribution of requests response times, now we see a more fixed and small range for each user group. Nonetheless, we see a smaller response time; all requests keep on the 2000ms time range, which shows us that we could not lower this metric with our available hardware.

If the reader pays attention, up to 25 simultaneous users can hold a mean response time as lower as 250ms, which is fantastic. But as we escalate just up to 100 simultaneous users, which is 4x the number of simultaneous users, the mean response time jumps from 250ms to 2000ms, 8 times the initial response time. In other words, the mean response time is increased at a faster rate than a linear correlation. On the other hand, we are still under the 3 seconds rule of thumb of Google's SEO recommendations. Now we are going to make a general analysis of the test distribution.

7.4.6 Test analysis

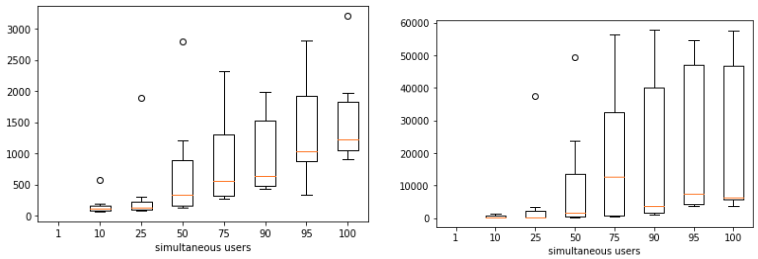
Comparing boxplot 7.43a and 7.22, referring for percentile 50% of the requests, we see the same behavior and the same mean response time interval. Figures 7.43b and 7.23 as on boxplot for 50% show the exact time interval and outliers groups. As the reader might already expect, the same behavior happens when we are talking about 95% of the requests on plots 7.43c and 7.2499 on plots 7.43d and 7.25.

Now we will look at the bar plot of completed requests and mean response time to see if we can spot any differences. By looking into 7.44, more precisely, on 7.44a and comparing against,7.31 we see, again, the same mean response time intervals and outliers, which show us that advance caching has proven itself not to be worth the complexity. The reader might be wondering if it is the answer to all scenarios, and be sure that it is not; the case we see here is that we are already at or lower response time as possible given our hardware.



(a) *Response time in milliseconds for 50% requests.*

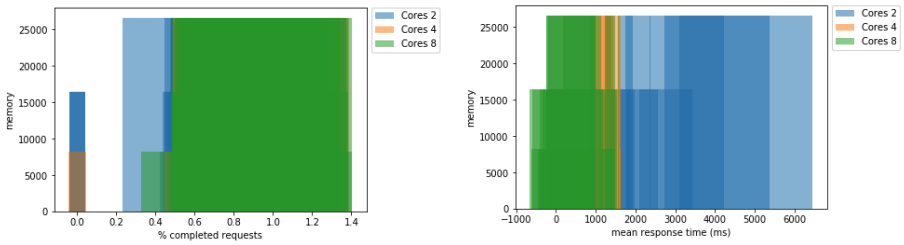
(b) *Response time in milliseconds for 75% requests.*



(c) *Response time in milliseconds for 95% requests.*

(d) *Response time in milliseconds for 99% requests.*

Figure 7.43: *Boxplot distribution of mean response time for advanced caching*



(a) *Mean response time for completed requests x amount of completed requests*

.

(b) *Mean response time for completed requests x memory amount.*

Figure 7.44: *Bar plots for completed request and mean response time grouped by cores*

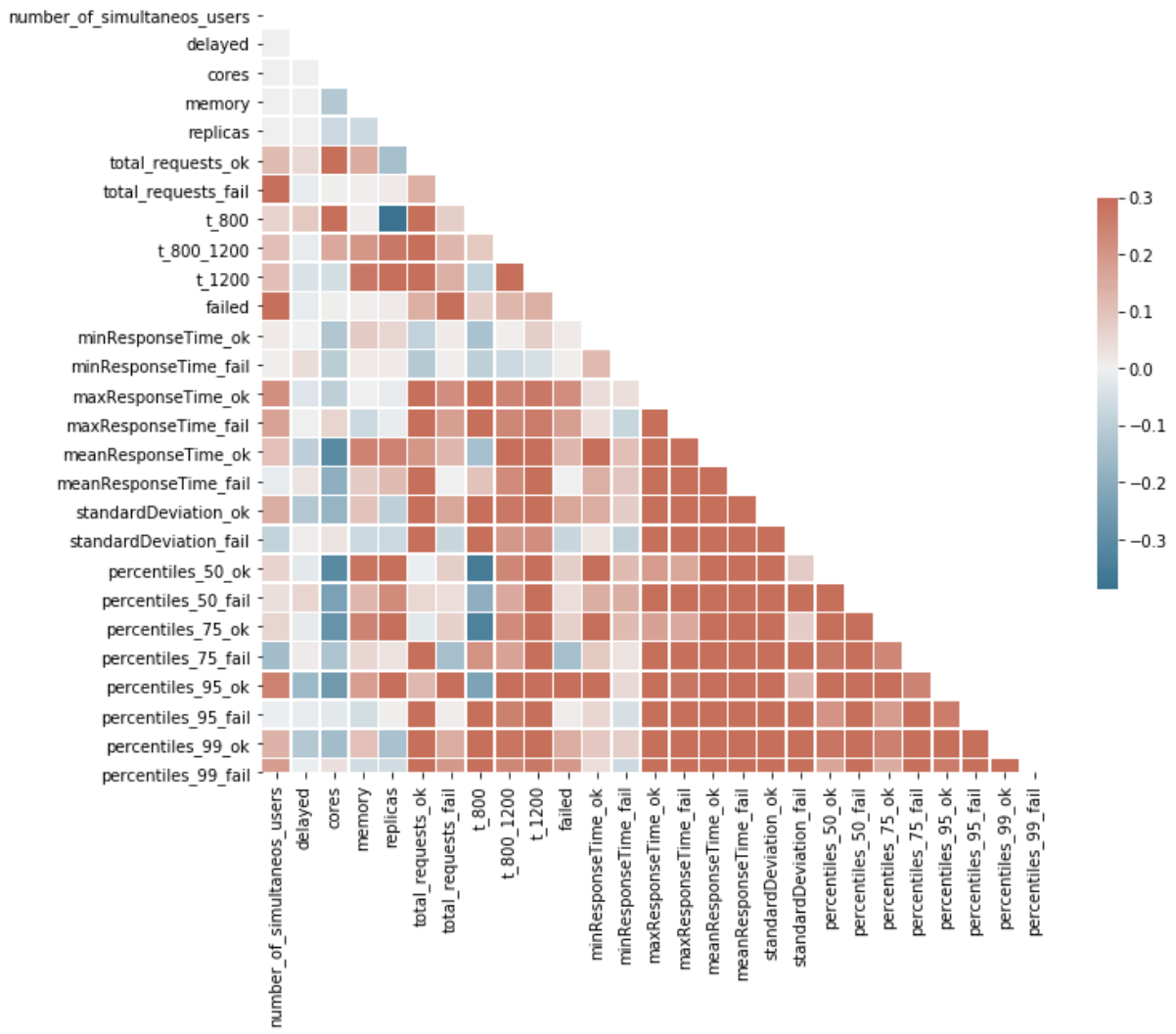


Figure 7.45: *Correalation heatmap.*

By looking at heatmap 7.45, we see a few interesting correlations, first for replicas and number of handled requets and memory and mena responresponse, as we have been guessing until now. On the other hand, process power and cores aparently do not have so much influencce on all its results, but the problem here is that our dataset is limited from 1 to 100 simultaneous users. if we compare previous 7.32from basic caching, we see cores have less impact on response times, but there is almost the same influence on total requests, as we have already mentioned.

Chapter 8

Conclusion

We have already concluded all our experiments and wet extensively thought all of them. The raw data and the Jupyter notebook, which produce these experiments, are available to download in the appendix A. To organize the ideas behind this study and what we have accomplished so far, we will recapitulate the most important parts and how they correlate to each other.

We started our research by understanding why performance is important and why we should care about our application's response time. Next, we jumped into fastness scales and how fast is fast enough, and what is a good response time. Then we started studying defenses between architectures for deploying a web application based off Rails; it was using Ansible script and a monolithic server against using a Kubernetes approach with horizontally scalable cluster, where we discussed the benefits of using Kubernetes against its main disadvantage, which is losing performance but gain scalability.

Talking about how fast is fast enough and response time, it is important to recall our standards. Our research stated that an acceptable mean response time would be 3000ms, and a gold standard would be 1500ms. We have discussed that we would be handling 95% of the requests and would not care about the appeared response time ad would not use the Ajax method to give an impression to the user that our response time is lower; we measured real response time and page loads.

Our workload test environment was composed of a couple of variables. The number of cores, RAM amount, replicas set, talking about cluster parameters. We ran load tests using Gatling to stress those variables, which ran ramp users access during 1, 10, and 60 seconds for batches or 1 to 2000 simultaneous users in intervals of 25 users each. We used a couple of scripts in Scala, Bash Shell, and NodeJS, and the help of Gatling software. Our tests consisted of tweaking these parameters in cluster size and load test payload. Our goal is to use cache solutions such as Redis to increase our performance time to keep inside our golden standard of 1500ms, but 3000ms is still acceptable.

To maintain a comparison base and understand which kind of stress our software would encounter, we had to do a simple analysis of Brazil's population. Recalling what we have studied about our country city populations and governmental web portals access data, we could estimate an average expected access load. Our study has shown us that São Paulo's platforms, Governo Aberto[1] had received at its peak access, 30000 users during November of 2019, but averaging 1500 users each month. On the other hand, the federal website for retrieving population's data, Portal Brasileiro de dados Abertos[2], received at its peak 12000 users in July of 2016. Now that the reader had remembered our goal, the standard we set

[1] http://www.governoaberto.sp.gov.br/
[2] http://dados.gov.br/

to achieve, and comparison bases on Brazil data access portals data, let us jump into the results.

Our statistical analysis starts by looking at Consul original code implementation but deploys through a Kubernetes stack instead of a normal monolithic virtual server. We deploy that code using the provided Docker image but with our own implementation of Kubernetes architecture. With our simulated cluster ready and our stack deploy to Kubernetes, we ran our tests using bash and Scala scripts using Gatling. Then parse generated results using other NodeJS scripts, which gave us a CSV file to summarize. The readers can find out on our repository to simulate for their own.

By analyzing the results, we have understood that Consul, as a Ruby on Rails applications, dramatically relies on the replica set and the memory for performance, with cores playing a smaller role but still needed to increases performance, but not throttling our simulated cluster capabilities. On the other hand, increasing the replica set also increases our cluster's memory usage, which leads us to memory being the main responsible for lowering the mean response time, thus increasing our cluster performance.

Jumping into the numbers, we saw that the original code was able to handle the requests with just a small margin above our golden standard of 1500ms with an average mean response time of 2000ms for 100 simultaneous users but 500ms for cached pages loaded directly from the file system and just around 2500ms for 2000 users. It is also important to remember that most of our requests were clustered around 1500ms ranging all users intervals. But what is important is to pay attention to all plot shapes; all histograms remember a Normal distribution, where it starts with a small and increases to its peak, and then the curve smoothly decreases.

Caching translations, session data, GETs, and HEADs requests allowed us to handle 95% of the requests within our golden standard or 3000ms. But not only that, in better scenarios, we were able to lower our response time by half, allowing page loads of 500ms. If we were to talk about handled requests, our numbers jumped to almost 100%, excluding outliers due to network problems. On higher simulations, with like 3000 or 5000 simultaneous users, the numbers dropped slightly, but we were able to handle 75% of the requests within a page load time of, as low as, 500ms. Caching allows us to delivery speedy page loads; the problem is that we start to throttle with process power to handle all open connections to our machine.

The proposed improvements were all about caching and if it was worth increasing complexity to increase performance. To do so, we divided our tests into two groups; first, we analysis simple caching, just leaving Rails to cache what it "thought" was important and caching translations. We jumped into the caching page and partials to identify if performance increases and if it is worth the performance difference due to managing cache state on small pieces of code updates. It is also important to keep in mind that our research has shown us that cache is one of the main causes of bugs in enterprise level applications.

We did pretty much the same stress tests for the original consul code, one batch for simple cache strategy and another for the advanced caching. First, looking at our plotted results, we saw that there were not too many improvements to our application level. We saw improvements for handling 100% of the requests on 100 simultaneous users. The plots became more stables, but we did not see much improvement in the average mean response time; the stability came from a smaller standard deviation on the cache version than the original code.

The results discussed above are what we expected from our research. Cache helps out by eliminating some access the Rails application does to the database. This shows us that cache is an important ally in a situation with small resources. It is crucial to leave some of the resources for a cache server. It will dramatically improve your application. The other

scenario is where you have huge amounts of simultaneous users on your applications and problems like store session data. It can dramatically improve by leaving those access to the cache instead of a regular SQL database.

On advanced caching problems, the benefits are even less worth it. If you are managing huge enterprise-level applications with thousands of simultaneous user access and a team to maintain your application, you will benefit from advanced caching. Our analysis has shown us no significant improvements when compared to the basic caching scenario. If we keep in mind that advanced caching is one of the main sources of bugs on the enterprise-level application, when can infer that caching page, partials, query results all over the code to decrease even more the database access and lower our resources is not worth the work you will have to spend on those upgrades, unless you are managing high traffic servers.

Keeping it simple, if you are low on resources, or if you are managing enterprise-level applications which does not have much traffic, it is important to keep caching on a basic level. Let automated caching utilities do the job, do not write "cache code" on your application, and already see improvements on your application. The other scenario is managing an application with 5000, 10000, or even more simultaneous users, which heavy load on databases servers. In this scenario, you must use advance caching to relieve as much query's to the SQL databases, like PostgresSQL, as possible, as querying databases is significantly slower than access caching databases like Redis.

Utilities like actionpack-page_caching gem[3], which automatic caches all GETs and HEADs requests, or automatic caching utilities given by rails itself like translations for i18n, or session data caching are the best utilities to improve your web application without increasing complexity. If you are interested in small advancements, the reader might want to try caching partials and query results for simple queries for heavily accessed pages, which could relieve database access and process power needed. Still, it is vital to keep caching simply. Advanced caching, for complex requests and actions, is dangerous because it could lead your development team to spend more time-solving bugs caused by the cache, improving your application, or solving all other bugs and issues that are already on the road-map.

Finally, it is also important to understand how you will build your Kubernetes architecture. For example, our experiments have shown us that one Postgres database instance could handle just a fine 5000 requests simultaneously, so, probably, you would want to use 2 or more instances or your Rails application to connect to the same Postgres database. Another interesting understanding is how you will build your Redis cluster and your Rails replicas connected to the same cluster. For example, how much data are you storing, how many Redis instances would satisfy your Rails cluster's needs? You should not build a Kubernetes cluster with 1-1-1 instances, 1 rails replicas for 1 Postgres replica, and 1 Redis replica. To maintain instability, we have seen that it is important to keep a Redis cluster of at least 5 nodes for replicas and master-slave behavior, and, as Redis is using LRU strategy, if you leave small amounts of memory available for caching, you will experience a huge amount of page faults, due to caching being constantly swapped, which will leave on the worst performance then it was without caching.

Think about the replica set; it is better to leave Kubernetes to manage the replica set; you should leave threshold based on CPU and memory for when the container reaches 75% of CPU usage or 75% of memory usage it should escalate the replica set, then if the memory of CPU is as low as 25% the cluster should decrease the replica set. To take advantage of horizontal escalators' full potential, you should use a host solution that proved automatic cluster management that can add or remove machines to your cluster, allowing Kubernetes to increase its power even more or decrease your costs application as needed.

[3] https://github.com/rails/actionpack-page_caching

Appendix A

Code used for this simulation

Repository with insignts for Consul's performance improvements:
https://gitlab.com/lbenicio/consul

Repository with scripts for simulation automation:
https://gitlab.com/lbenicio/consul-tests

Repository with raw data generated on this study:
https://gitlab.com/lbenicio/consul-tests-raw-data

Consul original source code:
https://github.com/consul/consul

The tests and scripts were developed on debian as host OS.

Docker:
https://docs.docker.com/engine/install/ubuntu/

Minikube:
https://kubernetes.io/docs/tasks/tools/install-minikube/

Java:
https://launchpad.net/ linuxuprising/+archive/ubuntu/java/+packages

Gatling Load Test Tool:
https://gatling.io/open-source/

Bibliography

Barber() Scott Barber. How fast does a website need to be? Mention on page. 4

Barua() Hrishikesh Barua. A comparison of some container orchestration option. Mention on page. 11

Cassandra() Cassandra. https://cassandra.apache.org/. Mention on page. 45

CauchBase() CauchBase. https://www.couchbase.com/. Mention on page. 45

Chen *et al.*() Shanshan Chen, Xiaoxin Tang, Hongwei Wang, Han Zhao and Minyi Guo. Towards scalable and reliable in-memory storage system: A case study with redis. Mention on page. 46, 47, 48

Chinnachamy() Arun Chinnachamy. Instant redis optimization how to. Mention on page. 45

Corona() Blue Corona. How fast should a website load? Mention on page. 2, 3

Cruz() Ryan Joshua Dela Cruz. Why website speed is very important? Mention on page. 4

dhh() dhh. Rails 5.2.0 final: Active storage, redis cache store, http/2 early hints, csp, credentials. Mention on page. 47

Doyle() Kyle Doyle. Gatling vs jmeter vs the grinder: Comparing load test tools. Mention on page. 20

EnqingTang, and Fan() EnqingTang and Yushun Fan. Performance comparison between five nosql databases. Mention on page. 45

Fernando Maila-Maila1, and Ibarra-Fiallo() Monserrate Intriago-Pazmiño1 Fernando Maila-Maila1 and Julio Ibarra-Fiallo. Evaluation of open source software for testing performance of web application. Mention on page. 20, 21

Gilbert, and Linch() Seth Gilbert and Nancy Linch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. Mention on page. 45

Gupta *et al.*() Adity Gupta, Swati Tyagi, Nupur Panwar, Shelly Sachdeva and Upaang Saxena. Nosql databases: Critical analysis and comparison. Mention on page. 45, 46

HBase() HBase. https://hbase.apache.org/. Mention on page. 45

Ina Schieferdecker1, and Apostolidis() George Din Ina Schieferdecker1 and Dimitrios Apostolidis. Distributed functional and load tests forweb services. Mention on page. 15

Jiang *et al.*() Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann and Parminder Flora. Automated performance analysis of load tests. Mention on page. 15

Khan, and Amjad() SRijwan Khan and Mohd Amjad. Performance testing (load) of web applications based on test case management. Mention on page. 15

Loisel() Jérôme Loisel. Jmeter vs gatling tool. Mention on page. 20

Lönn() Ragnar Lönn. Open source load testing tool review 2020. Mention on page. 19, 20, 21, 22, 23

Macedo, and Oliveira() Thiao Macedo and Fred Oliveira. Redis cookbook. Mention on page. 47

Malavolta *et al.*() Ivano Malavolta, Katerina Chinnappan, Lukas Jasmontas, Sarthak Gupta and Kaveh Ali Karam Soltany. Evaluating the impact of caching on the energy consumption and performance of progressive web apps. Mention on page. 47

Memcached() Memcached. https://www.memcached.org/. Mention on page. 45

Menascé() Daniel A. Menascé. Load testing of web sites. Mention on page. 15

Mercl, and Pavlik() Lubos Mercl and Jakub Pavlik. The comparison of container orchestrators. Mention on page. 11

Mertz, and Nunes() Jhonny Mertz and Ingrid Nunes. A qualitative study of application-level caching. Mention on page. 46, 47

MongoDB() MongoDB. https://www.mongodb.com/. Mention on page. 45

Mookerjee, and Tan() Vijay S. Mookerjee and Yong Tan. Analysis of a least recently used cache management policy for web browsers. Mention on page. 48

Paz1, and Bernardino() Solange Paz1 and Jorge Bernardino. Comparative analysis of web platform assessment tools. Mention on page. 15, 22

Postgres() Postgres. https://www.postgresql.org/. Mention on page. 46

Pradeep, and Sharma() S. Pradeep and Yogesh Kumar Sharma. A pragmatic evaluation of stress and performance testing technologies for web based applications. Mention on page. 14, 20

Proskurin() Andrei Proskurin. Adapting a stress testing framework to a multimodule security-oriented spring application. Mention on page. 14, 15, 23

Redis() Redis. https://redis.io/. Mention on page. 45, 46, 47, 48, 49, 55

Riak() Riak. https://riak.com/. Mention on page. 45

Selvidge() Paula Selvidge. How long is too long to wait for a website to load? Mention on page. 3, 4, 5

Sundbaum() Niklas Sundbaum. Automated verification of load test results in a continuous delivery deployment pipeline. Mention on page. 13, 14

Takai *et al.*() Osvaldo Kotaro Takai, Isabel Cristina Italiano and João Eduardo Ferreira. IntroduÇÃo a banco de dados. Mention on page. 46

Walton() Philip Walton. User-centric performance metrics. Mention on page. 3

Zhen Ming Jiang, and Flora() Gilbert Hamann Zhen Ming Jiang, Ahmed E. Hassan and Parminder Flora. Automatic identification of load testing problems. Mention on page. 15