Improving the scalability and performance of a rails application: A case study with Consul

Leonardo A. B. Santos[1], **Vanessa M. Tonini**[1], **Alfredo G. v. Lejbman**[1]

[1] Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo (IME-USP) – São Paulo, SP – Brazil

{vanessametonini,gold}@ime.usp.br, leonardo.araujo.santos@usp.br

Abstract. *This research studies how a Ruby on Rails application reacts to an increasing number of simultaneous users accessing the service. More specifically, we are going to study Consul, and propose improvements to lower its response time and increase its number of supported simultaneous users. These propositions will need containerized application parallelism and cache request techniques, which will lead to a faster response time, with fewer bare-metal servers to sustain the same amount of load. On average, we were able to increase the request rate by 60%, allowing more users to access the resources and faster page loads.*

1. Introduction

1.1. Motivation

If we think at the machine level, year by year, companies release faster processors [8]. Mobile internet and data plans are getting more accessible on under development countries [9]. Concerning the application level, the operating system gets updated annually, due to techniques like Continuous Development. Applications also get updated to increase their performance constantly. Therefore, pushing the content at faster rates as well. Day after day, users expect applications to work more quickly.

On a web server, we have to deal with an enormous amount of variables: network packets in and out, memory, disk space, disk read/write speeds, latency, process power, and much more. However, it is difficult for all these parameters to find a model to track the performance of a web application. Page loading time shows us a perfect combination of all those variables, as stated by [7].

In a more complex scenario, the situation is even worse: if a service takes too long to answer a request, other services may have to wait, which causes a waste of resources; if web applications take too long to answer a request, you could leave the other machine at the other endpoint while waiting, and also wasting resources. We may also quote [6], where he analyzes the Rails' performance, starting from Ruby compilations, to caching databases.

> In recent days, network applications and websites have exploded in numbers. Applications should be flexible, dynamic, and fast, with no unnecessary delays. (...). These tools are also supposed to ease the development process for the developer, balancing ease of use with performance for the perfect combination.

In consequence, we aim to achieve more awareness on how to deploy a scalable Ruby on Rails application. Simultaneously, we expect to use the available hardware to minimize costs and increase the number of answered users and their respective response times. Considering that, the understanding of how to increase Ruby on Rails' performance is crucial.

We aim to answer the following Research Question: Is it possible to improve the performance by improving the number of answered requests and their respective times?

1.2. Organization

First, we will discuss target metrics, and explore the current industry standard. Then, we present scalability concepts, and introduce the variables involved in our analysis. To demonstrate those topics, we will use Consul[1] as our sample application. Afterward, we will discuss current bottlenecks on Consul implementation. To achieve that, we will elaborate on our experiments and get to the measurement data.

Then, suggested improvements, in which we will discuss caching structures and horizontal scaling Consul by using Kubernetes and containers. In this part of the text, we will replicate the same experiments and compare the results to understand the performance gains. Thereby, it is possible to elaborate on the role where Kubernetes benefits us and where caching could assist the application to perform better.

Finally, we will discuss our study limitations and conclusions. At this point, we will compare both scenarios and explain some strategies to increase the performance of a Ruby on Rails application. The goal here is to take advantage of the horizontal scalability with maximum resource usage by applying simple cache techniques.

2. Background

2.1. Why Consul?

Consul[2] is a Free/Libre and Open Source Software (FLOSS) — licensed on AGPL 3.0, and written with Ruby on Rails. A Consul instance creates a complete citizen participation tool for open, transparent, and democratic government. It allows citizens to collaborate in the legislation process, polls, debates, and participate in budget voting.

The development of this tool started on July 15th in 2015 as an initiative of Madrid's town hall. The source code is open to the world in a public repository. Today, Consul Foundation is the main supporter and counts with voluntary core developers, and more than 100 contributors; resulting in a hectic GitHub repository with more than 14000 commits, and 1000 stars.

According to the official website, over 35 countries and over 135 institutions already implemented Consul, allowing more than 90 million citizens to participate in public life through debates, polls, and others structures. When it comes to Brazil, 4 instances of Consul are running at the municipal and state levels to foster citizen participation, including São Paulo[3], the biggest city in the country with more than 11 million citizens[4].

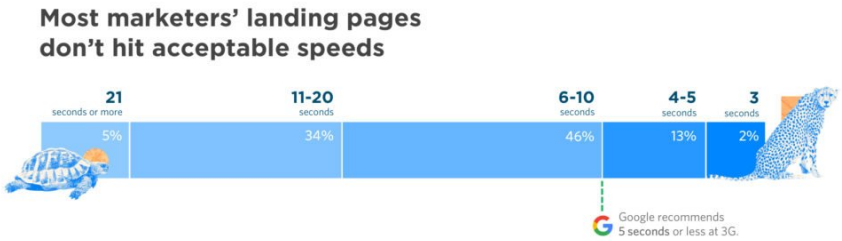
[1] https://github.com/consul/consul

[2] https://consulproject.org

[3] https://participemais.prefeitura.sp.gov.br/

[4] https://censo2010.ibge.gov.br/sinopse/index.php?dados=21&uf=35

(a) Survey on how long a user would wait for a page to load against Google metrics.



(b) Landing page response time for top Marketers at Google.

Figure 1. Blue Corona's study [3]

2.2. How fast is fast enough?

We think there is enough motivation to prove that increasing page loads is an important matter. However, it brings up another question: how fast is fast enough? Actually, there is not an agreement, but we can investigate it and get a satisfactory estimation: for example, according to Google, their aim to page response time is under 500ms. But, before we start concerning the metrics and goals, we need to define metrics.

For web developers, it is possible to recognize the moment when a page has been completely loaded by the browser and is ready for the user to interact with it. The "DOM-ContentLoaded" event of the DOM (Document Object Model) characterizes this moment, which can be observed programmatically by JavaScript[5]. Still, we must clarify that matter, and to do so, we will use [11].

After setting a goal, we should aim to discuss our user's scenario. For that purpose, we may mention Google's ranking, where the top-ranking websites get a page loading time of under 3000ms, according to [11]. Nevertheless, the reality is not good as that. The Unbounce page speed report[6] says that in 2019 the website loading time was about 15000ms. As we can see in Figure 1b, 2% of Google's crawled websites to respond to their requests under 3000ms; and 13% got under 5000ms, which is the maximum recommended response time over 3G connections.

[5]https://blog.logrocket.com/custom-events-in-javascript-a-complete-guide/
[6]https://unbounce.com/page-speed-report/

On the other hand, the same report shows us that different users behave differently. For instance, "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 would wait 11-13 seconds, only 36% were iOS users versus 61% Android users." The report also states that most of the users will, firstly, blame the internet provider for their equipment before blaming the website.

2.3. Scalability

Our goal in this study is to identify scenarios where a Ruby on Rails application underperforms and discovers what is causing such performance, thus leading us to insights on building an improved web application. To do so, we are going to use a simulated Kubernetes cluster through Minikube[7]. Minikube is a small CLI (Command Line Interface) which is the suggested way to achieve a production-like environment for Kubernetes on a development machine (single machine).

Using RAM and CPU allocations 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 of them handling 250 users, or just one replica handling it all? Does it 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. By ranging these two variables, we can stress our application to discover how our hardware answers to 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, announcing that we are starting to deteriorate the application response times.

By simulating all these situations, we expect 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. This analysis is not exclusively about Consul itself, though. It is about how a Ruby web application escalates under a cluster environment.

3. How to scale up Consul?

Following the current documentation[8] of the Consul, it is suggested to deploy the full stack of the application on a bare-metal server. It is also important to mention that in big cities it is recommended a separate server for the database. The suggested deployment method leads us to initially rely on vertical scaling of our resources, which causes wasting resources while not on peak usage.

To solve these problems, we will use containers and a Kubernetes cluster, which still does not solve all our problems. Most of the user requests are for consuming data by navigating on the application, and most of the time, the server has to deliver the same content to multiple users. Rails application also stores user session and data cache to manipulate each user's interaction, which happens to be used all over the application.

[7] https://minikube.sigs.k8s.io/docs/

[8] https://docs.consulproject.org/docs/

If we had to perform all those steps in all requests, we would be wasting resources as well. Our goal is to scale multiple replicas and keep the number as low as possible. For that, we need a cache server for storing; fast access to all those data; and avoid re-processing the same requests.

To scale up Consul, we will have to convert the Ansible script[9], provided by the official documentation, to Docker and Kubernetes implementation for enterprise-level application. Also, there are the deployment scripts for a scalable PostgreSQL database; Redis server; and File System server used for files uploads as profile pictures. This is different from a bare-metal application, where the database does not scale and cannot store data files locally. On a distributed system, all this data need to be available anywhere our server spawns and also to be independent of the machine which runs the container.

4. Methodology

Our study will be based on four scripts to complete the experiments. The Scala script will be used as load simulation software; NodeJS to analyze generated results; Bash script to control the simulation software executions; and Python to analyze the final data. Therefore, to generate traffic we need a loaded software, which we will use Gatling[10]. It is an Open Source, cross-platform, load generator software, which also supports enterprise companies. It is great for production environments, similar to what we can see on CentOS and Red Hat on the Linux OS level.

Variable	Description
$TRIALS$	controls how many times Gatling will repeat the same test
uamount	controls the number of simultaneous users each batch of tests will simulate
tamount	controls the user access duration (ramp user access mode)
ramount	controls the number of replicas of the application
camount	controls the number of cores in the simulated cluster
mamount	controls how much memory will be available in the simulated cluster

Table 1. Main script variables to control the experiments.

Our main script controls the six main variables. The first one is $TRIALS$, which controls how many times Gatling will repeat the same test; while $uamount$ 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 accessing the application during a period of 60 seconds.

The second group consists of $ramount$ and $camount$, which control the number of replicas of the application's containers and the cores in which Minikube will use to simulate a Kubernetes cluster. At last, $mamount$ controls how much memory will be available in our Minikube's simulated cluster, as summarized in Table 1

Replicating a real-world scenario, we are simulating users accessing the cluster using a Gatling script which makes a combination of HTTP requests. In our environment,

[9]https://github.com/consul/installer
[10]https://gatling.io

we are making HTTP requests of type POST, GET, OPTIONS, and PUT. Gatling will measure those requests data like mean response time, completed requests ratio, and status codes. Finally, it will make the requests for us and combine the data into a report of HTML files which will be parsed by our script.

Concerning our stack environment, we will do our experiments on a simulated Kubernetes cluster using Minikube. Powering the simulated cluster there is an eight-core Intel(R) Xeon(R) CPU E3-1230 V2 @ 3.30GHz, bare-metal server, with 32 GB of DDR3 memory on x86_64 architecture. The server get 32K L1d and L1i cache, 256K L2 cache, and 8192K L3 cache.

5. Current Bottlenecks

First, we need to present our study environment. In this section, we will be discussing simulations of 1, 10, 100, 1000, and 2000 simultaneous users accessing the application with ramp access mode during a period of 1, 10, and 60 seconds. Our cluster simulates 2, 4, and 8 cores. Also, there were 3 steps for memory: 8 GB, 16 GB, and 26 GB. Inside our cluster, we used 1, 2, 4, 8, 16, and 32 replicas of the application. Each one of these variables represents a scenario and a simulation. Each simulation runs 10 times, on the other hand, Gatling simulates 132 to 25000 requests to our application. Given we are using a controlled environment, we are dealing with less variability. Our next figures summarize our experiments.

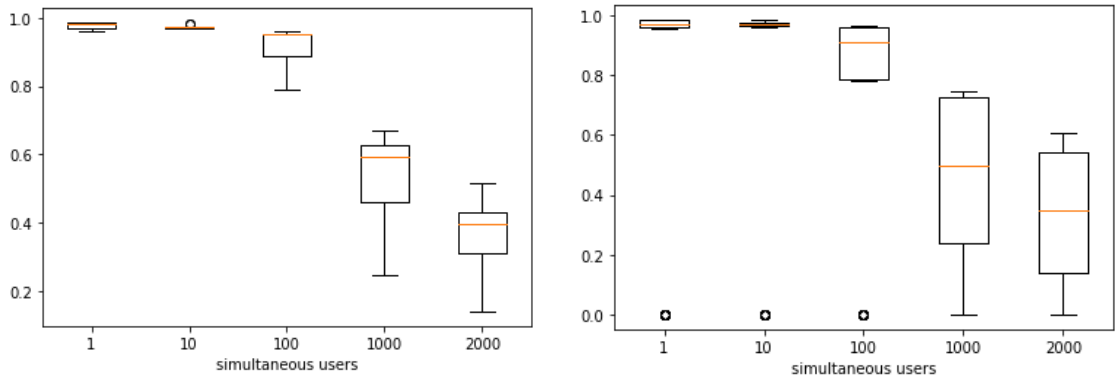
We will be using, most of the time, histograms of mean response time and amount of completed requests as our main metrics to understand the application behavior. But, why is that? Each Gatling simulation does from 120 to over 25000 HTTP[11] requests of type POST, GET, OPTIONS, PUT and DELETE. Each simulation generates a report, but we are doing 10 simulations for each trial. So, in order of magnitude for our worst case, we have 250000 requests to analyze.

The number of completed requests, and the mean response time for each group of simulated users by its frequency, are excellent ways to summarize our data and understand how Consul performs. For example, for 1 simulated user, accessing during 60 seconds, we have around 132 requests for each simulation, on 10 trials, we would have around 1320 requests. It is important to remember that one scenario refers to the amount of RAM, amount of cores, number of replicas, user access, and time during which those simulated users accessed the application.

As we increase the simultaneous user access, the response time deteriorates quickly. 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 a balance between those resources, which requires further investigation.

We managed to handle almost 60% of the requests of 1000 simultaneous users but only around 40% 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. At this point, we will check what the response time for 1000 users was.

[11] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods



(a) 4 cores **(b)** 8 cores

Figure 2. Completed requests with 16 GB RAM during 60 seconds.

The problem here is that we are handling only the third quartile of the requests and still resolving to high response times. What is throttling our environment performance is memory. In contrast to the low resource usage when we look at the CPU power, memory is heavily used by our application, which caching solutions will greatly benefit us.

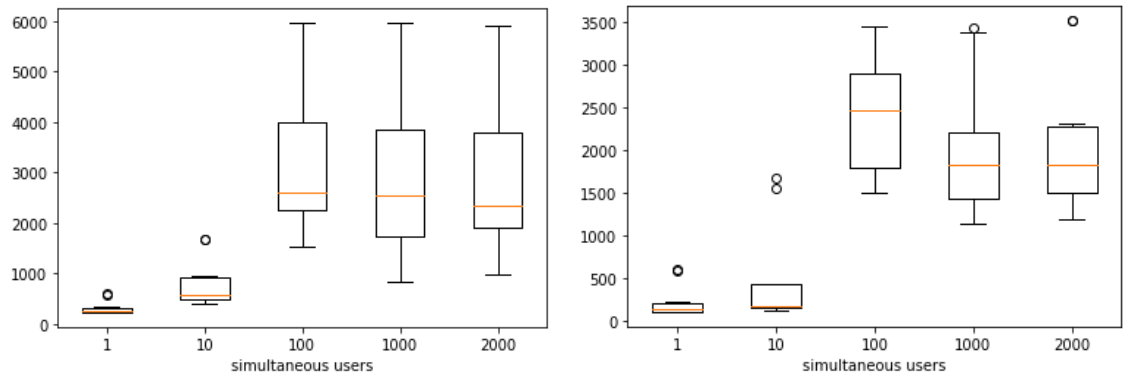
5.1. 8 cores and 16000 MB RAM

From the Figures 2a, 2b, 3a, and 3b, observing the curves of 1000 and 2000 users for 60 seconds, it is clear that deploying more replicas is not the solution; it is actually the opposite. As a developer handling the server, we must understand the spot to get the best performance out of our cluster. The Figure 2a 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 well around 50% and 70% of our requests, which would change with different amounts. Still, could we use less hardware to deploy this same amount of requests?

We doubled the core dedicated to the cluster. However, we still keep the same amount of memory available for doubling the processing power amount, which 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. Now, we will skip to full cluster power, concerning 26624 MB of memory and 8 cores.

5.2. 8 cores and 26000 MB RAM

Since we were able to increase the performance, we used approximately 3000 MB of memory for each node of our cluster. Here, there is 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 cannot guarantee the resources that are requested because if we give all machine resources to our cluster, our test application won't be able to simulate all those user sessions. Therefore, we reached a point where we have not been able to advance further in our investigation. In our best scenario, we could handle around 700 to 1200 users simultaneously; if we consider the mean, around 950



(a) 4 cores **(b)** 8 cores

Figure 3. Mean Response time with 26 GB RAM during 60 seconds.

simultaneous users, it would lead to approximately 41 million users/month. Regarding the mean response time, what could we conclude from it? Did it increase?

6. Proposed Improvements

6.1. Kubernetes

Developing and deploying web applications relies on different stages. Each of these stages depends on multiple libraries, depending on other runtimes and specific versions of each software. Setting up all the dependencies and the required software to get the developer's work environment up and running is a difficult task. Reproducing the same required environment for a team of developers, or other environments as testing, staging, and production can become an even more difficult task. 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. Starting from the simplest of the benefits: scalability. If one is using a monolithic application and wants to increase their availability to handle more simultaneous connections, the obvious solution is increasing the server's hardware vertically by increasing its attributes such as disk, RAM, or CPU cores. The other option is to increase it horizontally, by adding more servers to function in parallel to handle the increase in traffic. At this moment, the container orchestrator works to increase it by using the second method.

Given a containerized application and separated into microservices, if one of them is damaged, it will be the only one. The containers are: isolated; sandboxed from the host operating system (OS); have virtualized CPU; memory storage; and network resources at an OS level. Here it is important to assure we are following best practices, assuring

isolation and sandbox of the containers. Kubernetes emerges in a scenario where there is the need to handle container failovers, downtimes, scaling, load balancing, storage orchestration, automated rollouts[12], 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. It will also expose our DNS name, assuring a stable environment on each server. Furthermore, storage orchestration is equally important for us since it will handle installing 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 malfunctioning of damaged updates; if we can deploy containers at the desired rate, checking their functionality 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.

6.2. Redis

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 in Consul on some levels. First, we are going to cache only translations, i18n[13] 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 as our website's index are responsible for the most accesses. Sometimes, your application has to make hundreds of accesses 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.

Following [2], the great advantage that the NoSQL database brought to the industry was fast data access. The purpose of using NoSQL was the fast data access, and again, Redis is the leading NoSQL database for fast data in the industry. Another interesting study about caching tools performance is [4], which focuses their analysis on Redis, MongoDB, CouchBase, Cassandra, and HBase. In their study, they highlight exactly what we are trying to investigate in this study.

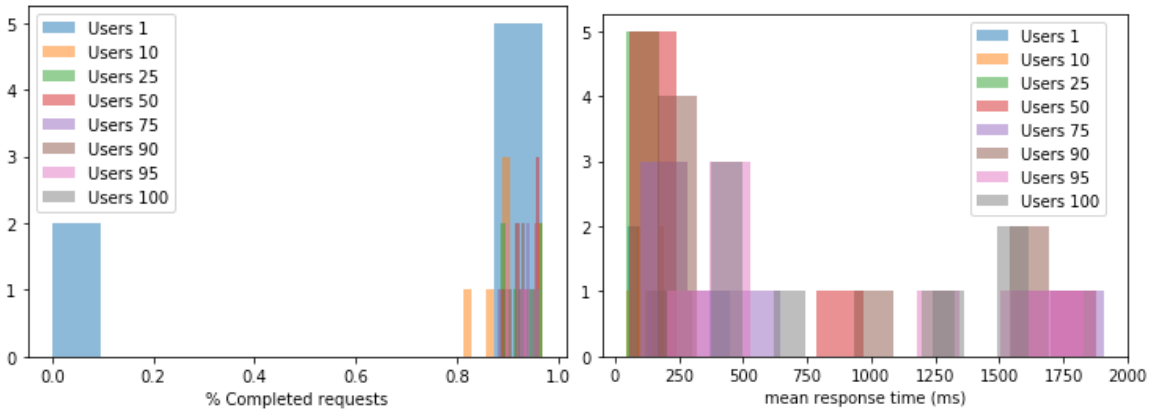
In the era of Big Data, mobile internet, and a huge amount of devices collecting data that need to be processed and disposed to the client queries, performance becomes unresponsive. It is the moment when NoSQL and cache become even more important. We apply the same content on frequently accessed pages in our scenario, 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.

That is when Redis, used along with the default PostgreSQL database applied on Consul, makes the difference. The standardized relational database provides all the benefits of ACID (which the reader can get a deeper understanding of when reading [10]), where Redis enable us to deliver fast read operations. In other words, Redis provides Availability where PostgreSQL provides Consistency.

Finally, we want to mention [1], which presents an extended analysis on Redis

[12]https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a-deployment

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



(a) 4 cores **(b)** 8 cores

Figure 4. Mean response time to 26 GB RAM with ramp users access during 60 seconds.

as fast in-memory storage access for web applications. In their analysis, it is proposed to create a "client-node" connection on the Redis cluster, in which the client connects themselves directly to the correct Redis node. According 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%.

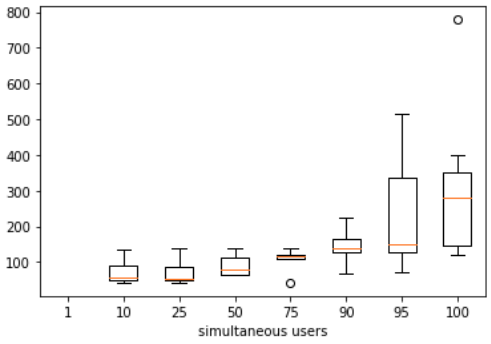
7. Experiments

Now we will focus on Figure 4a, hoping to discover once for all if it is worth advancing caching since we have already seen that the advanced-cache could be dangerous in small environments. Starting from Figure 4a we can notice the same behavior, where there is a small percentage of outliers, and the number of completed requests keeps reaching above 96%; for smaller batches of users, they keep around 98% and 100%- which is not surprising.

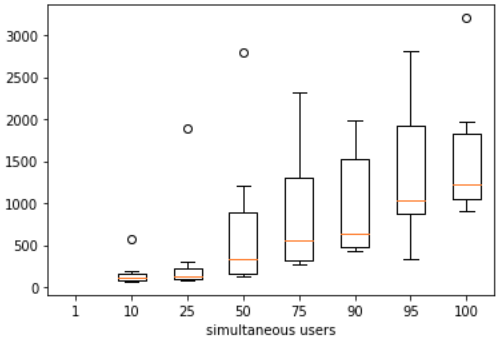
Skipping to mean response time, let us take a look at Figure 4b. Concerning the response time, we can see a small difference. Before, we would see a distribution of requests response times; now, we see a more fixed and smaller range for each user group. Nonetheless, we can notice smaller response time and all requests kept on the 2000ms time range, which shows us that we could not lower this metric with our available hardware.

We can notice that up to 25 simultaneous users can hold a mean response time as low 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 changes from 250ms to 2000ms, or 8 times the initial response time. In other words, the mean response time increases at a faster rate than a linear correlation. On the other hand, we are still under the three-second rule thumbing of Google's SEO recommendations.

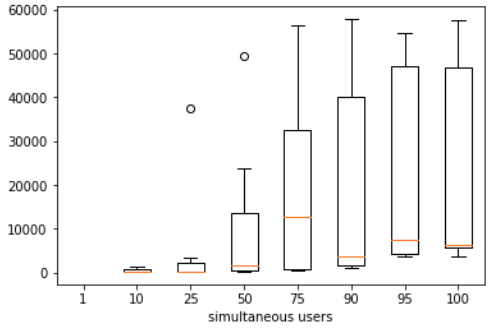
When comparing boxplots (Figure 5) and referring to a percentile of 75% of the requests, we see the same behavior and the same mean response time interval. As the reader might already expect, the same behavior happens when we are talking about 95%



(a) 75% requests



(b) 95% requests



(c) 99% requests

Figure 5. Boxplot distribution of mean response with caching.

of the requests on plots 5b and 99% on the plot in Figure 5c.

By looking at the heatmap in Figure 6, we may see a few interesting correlations, such as for replicas and t_800, which represents the number of request under 800ms. Leading to a combination of process power and memory. Another interesting correlation is between cores and total_request_ok and t_800. If we compare with basic caching, we notice that cores have less impact on response times, but almost the same influence on total requests, as we have already mentioned.

8. Limitations

It is important to clarify to the reader that we are performing all those tests on a single machine. So, we are using the same machine to deploy the loading generator software, Consul application, databases, Redis cluster, and Kubernetes itself. Therefore, we expect to suffer from performance loss due to all those software programs running on the same machine. The results are still valid to extrapolate to a distributed environment, though.

It is important to mention, as well, that we are not suffering from network routing. As we mentioned before, given we are using the same server to do all the simulations, requests made from Gatling to simulate server load will fail only to our server incapacity, not for network issues.

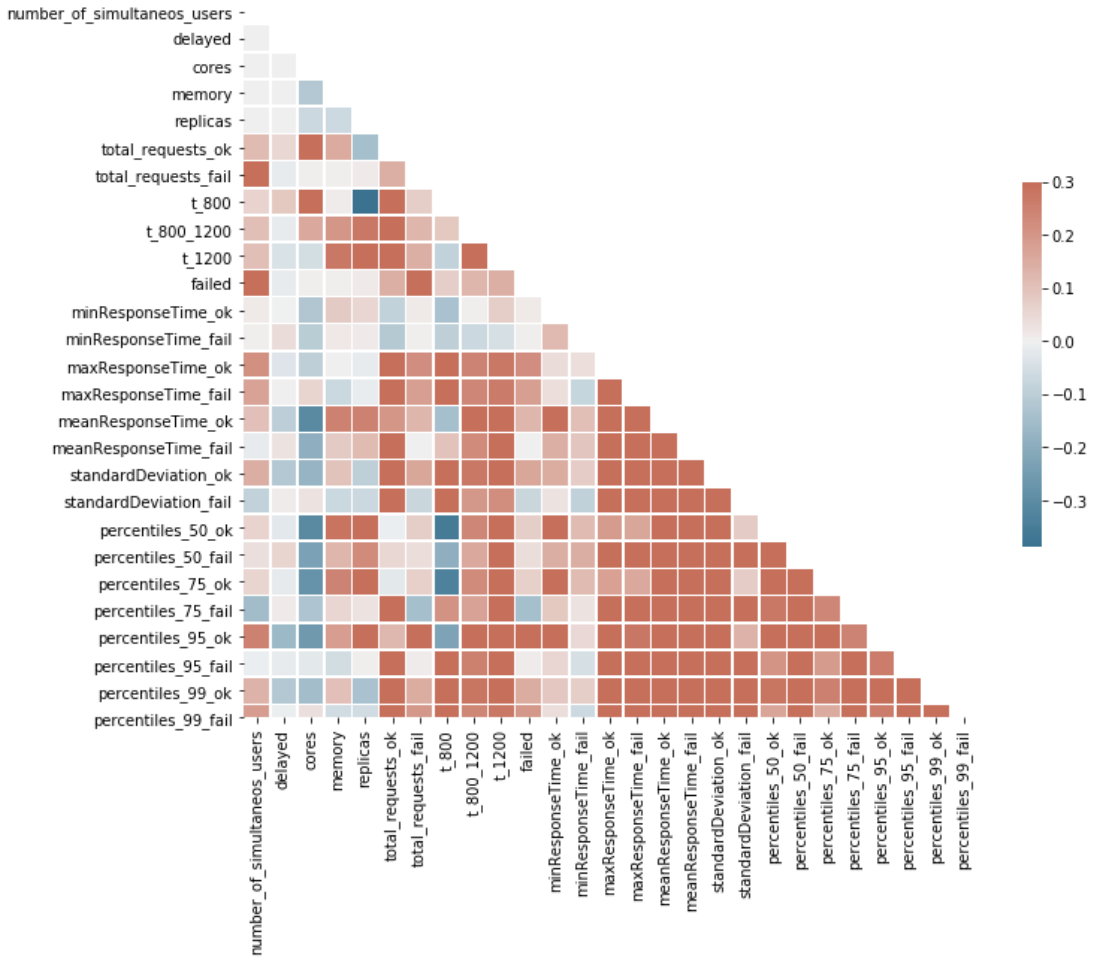


Figure 6. Correlation heatmap.

9. Discussion

By analyzing the results, we have concluded that Consul[14], as a Ruby on Rails application, dramatically relies on the replica set and the memory for performance, with cores playing a smaller role and still needed to increase 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.

We saw that the current implementation was able to handle the requests with just a small margin above our golden standard of 1500ms, with an average mean response time of 2200ms for 100 simultaneous users. Nevertheless, cached pages loaded directly from the file system responded between 500ms and around 2500ms, for 2000 users. It is also important to remember that most of our requests clustered around 1300ms, ranging all users' intervals.

Caching translations, session data, GETs, and HEADs requests allowed us to handle 95% of the requests within our golden standard of 3000ms. In addition to that, in better scenarios, we were able to lower our response time by half, allowing page loads

[14]https://github.com/consul/consul

of 500ms. If we were to mention handled requests, our numbers grew to almost 100%, excluding outliers due to network problems. On higher simulations, with about 3000 or 5000 simultaneous users, the numbers dropped slightly, but we were still able to handle 75% of the requests within a page load time of, at least, 500ms. Caching allows us to deliver 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 analyzed simple caching, just leaving Rails to cache what is considered to be important, and caching translations. Then the approach was 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.

On advanced caching, the benefits are not worth the complexity increase. If you are managing huge enterprise-level applications with thousands of simultaneous user accesses, and a team to maintain your application, you will benefit from advanced caching. Our analysis has shown 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 web applications, as stated in [5].

10. Conclusion

We can improve our performance by improving our requests rates but, if you are low on resources or manage enterprise-level applications that do 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 you will already see improvements on your application. The other scenario is managing an application with 5000, 10000, or even more simultaneous users, with a heavy load on databases servers. In this scenario, you must use advance caching to relieve as many queries to the SQL databases, like PostgreSQL, as possible, as querying databases is significantly slower than accessing caching databases like Redis.

Think about the replica set; it is better to leave Kubernetes manage the replica set; you should leave the threshold based on CPU and memory for when the container reaches 75% of CPU usage, or 75% of memory usage 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 can prove automatic cluster management and that can add or remove machines to your cluster, allowing Kubernetes to increase its power even more or decrease application's costs as needed.

Finally, it is also important to understand how you will build your Kubernetes' architecture. For example, our experiments have shown us that one PostgreSQL database instance could handle well 5000 requests simultaneously, so, probably, you would want to use 2 or more instances on your Rails application to connect to the same PostgreSQL database. Another interesting finding is how you will build your Redis cluster and your Rails replicas connecting to the same cluster. You should not build a Kubernetes cluster

with 1-1-1 instances, 1 rails replicas for 1 PostgreSQL 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. As Redis is using LRU strategy[15], if you leave small amounts of memory available for caching, you will experience a huge amount of page faults, due to constantly swapping caching — which will leave it in a worse performance than it was without caching.

11. References

[1] Shanshan Chen et al. *Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis*. 2016 IEEE Trustcom/BigDataSE/I SPA, 2013.

[2] Arun Chinnachamy. *Instant Redis Optimization How to*. Packt Publishing Ltd, 2013.

[3] Blue Corona. "How fast should a website load?" In: (2019).

[4] EnqingTang and Yushun Fan. *Performance Comparison between Five NoSQL Databases*. IEEE Xplore, 2003.

[5] Jhonny Mertz and Ingrid Nunes. *A Qualitative Study of Application-level Caching*. 2020.

[6] Martin Nordén. *To what extent does Ruby on Rails affect performance in applications*. Linkopings Universite — Institutionen for datavertenska, 2015.

[7] Kim Persson. *Optimizing Ruby on Rails for performance and scalability*. KTH Royal Institute of Technology, 2016.

[8] Matthew Halpern; Yuhao Zhu; Vijay Janapa Reddi. *Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction*. 978-1-4673-9211-2/16/$31.00 c 2016 IEEE, 2016.

[9] Gillwald Alison Stork Christoph Calandro Enrico. *Internet going mobile: internet access and use in 11 African countries*. 19th Biennial Conference of the International Telecommunications Society, 2012.

[10] Osvaldo Kotaro Takai, Isabel Cristina Italiano, and João Eduardo Ferreira. *Introdução a Banco de Dados*. EdUSP, 2005.

[11] Philip Walton. "User-centric performance metrics". In: (2019).

[15] https://redis.io/topics/lru-cache