TU Dublin, Tallaght Campus

MSc DevOps

Comparing Serverless Container as a Service Solutions in Public Cloud Service Providers

Vinícius Moura Barros

Department of Computing

Supervised by

Gary Clynch
Department of Computing

8 December 2020

Declaration

I hereby certify that the material, which I now submit for assessment on the programmes of study leading to the award of Master of Science, is entirely my own work and has not been taken from the work of others except to the extent that such work has been cited and acknowledged within the text of my own work. No portion of the work contained in this thesis has been submitted in support of an application for another degree or qualification to this or any other institution.

Vinícius Moura Barros
8 December 2020

Acknowledgements

I would like to acknowledge and express my enormous gratitude to my wife, whose support was essential to keep me well while completing this thesis, and whose passion to keep evolving motivates me every single day.

Thanks to my family that, even from far away, kept believing and supporting me. Thanks to my friends, who shared great knowledge with me during my journey. And thanks to my four cats, who were able to make me smile during difficult times, even when walking on my keyboard.

A special thanks to my supervisor, Gary Clynch, who listened to my progress every week and kept me motivated for the next one, always promptly giving me feedback and suggesting areas to improve. At last, thanks to my lecturer John Burns, who helped me clarify the initial ideas for this thesis when it was only a proposal.

List of Figures

List of Tables

Contents

Abstract

Containerised applications are becoming the new standard to wrap and deploy applications due to flexibility of runtime and capability to be executed in any environment that supports containers. Container as a Service (CaaS) is becoming more popular as organisations can deploy their containers in Cloud Service Providers (CSPs) without the burden of having to provision or manage the underlying infrastructure where these containers will run. Serverless CaaS also offers the flexibility of automatically scaling the infrastructure and billing schema that fit applications with dynamic demands, where customers are only billed for the time resources are utilised.

The comparison described in this research takes into consideration the three major CSPs (to the moment), however, is generic enough so it can be further extended to other CSPs. It covers details of how a containerised application was architected and developed alongside an automated way of provisioning and assessing the performance of different serverless CaaS solutions from different aspects such as application performance and internal CaaS processing times based on different sets of configurations of CPU and memory. Results about which solution is cheaper depending on the demand are also present, as well as a study about whether these CaaS solutions can really be considered serverless.

The research has identified that understanding well an application is of great importance when deciding which serverless CaaS solution to use. This research also found that some CaaS show better performance than others when running with similar resources and also that some applications can even run free of charge in a serverless CaaS depending on the usage.

Keywords: Serverless, CaaS, Container, Cloud, CSP, Performance, Comparison

1 Introduction

Cloud service providers (CSP) have introduced benefits and flexibility never seen before, such as the ability to provision infrastructure in different parts of the world in no time and only pay for the time infrastructure is used. This is reflected on the increasing number of organisations moving or already developing cloud-native solutions (Jambunathan and Yoganathan 2018).

The competition between public CSPs, either to attract new customers or keep the existing ones, is leading to the development of better solutions which are also easier to use. This is leveraging the amount of managed services available, reducing the overhead work and responsibility to operate and maintain applications running in cloud environments.

Serverless architectures are the ones where the infrastructure is provisioned and managed by a third party (the CSP in this case) whereas customers are responsible only for the development of applications and for their running configurations in the CSP, being usually supported by three pillars: high-availability and scalability, granular billing and event-driven services (Van Eyk et al. 2017).

Containers have been identified as the new application development paradigm by several sources (Jambunathan and Yoganathan 2018, Flexera 2020, *Cloud Run: Container to production in seconds | Google Cloud* 2020), as they offer an efficient way of encapsulating applications and their dependencies combined with the portability that allows deployments to different places that support containers. FaaS (Function as a Service) is still one of the most relevant serverless services available, however, Container as a Service (CaaS) started to become more popular due to the flexibility provided by containers in addition to the benefits of not having to manage a cluster of virtual machines required to host and run containerised applications.

The objective of this research is to compare serverless CaaS solutions provided by the major CSPs to the date (Amazon Web Services, Microsoft Azure and Google Cloud Platform) in matters of a simple application performance, CPU performance, internal CaaS processing, pricing and features available. Are these solutions truly serverless? Which solution has the best performance for a CPU intense task? Do the units of CPU provide similar results in different providers? Are there features that make one solution better than the other? These are the questions this research has studied and found answers for.

Following this chapter, in Chapter 2, the existing literature is going to be reviewed and more details about how the interest for serverless architectures and containerised applications has grown. From the increase in the number of academic studies being published in recent years to the growth in the utilisation of serverless and containers by companies taking advantages of quick and easy implementations provided by CSPs.

Chapter 3 is where the research methodology is presented. Early in the chapter, a list of questions to be answered and hypothesis for each one of them will be stated. After the questions, how the research was idealised and planned will be explained as well as what metrics were collected so relevant information could be extracted. It is also in Chapter 3 that it is described how the plan was actually executed. Details will be given about how the components were chosen and organised, from the development of an application to be tested, going through the development of an automated way of testing the services in the CSPs, up to the collection of meaningful data that could be eventually analysed.

In Chapter 4, based on the criteria and data obtained from Chapter 3 and on the literature review from Chapter 2, the study will present the comparisons between the serverless CaaS solutions. Also in this chapter, discussions about collected data will be made alongside conclusions based on the findings, including definitions about which service is the best or worst depending on the topic analysed. Finally, in chapter 5, conclusions about the studied serverless CaaS are going to be presented, alongside possible continuations of this study with topics this research was not able to cover, should academic students or companies wish to extend or increase the scope of this research in the future.

2 Literature Review

2.1 Introduction

Recent works related to serverless and containers are going to be covered in this literature review. The focus on these topics has increased as demonstrated by the number of available studies that has grown in recent years. However, are these are relatively new topics, there are still some gaps in the literature, which was one of the motivations for this research.

2.2 Serverless

Having a clear definition of serverless is needed especially as many publications refer to it interchangeably as FaaS (Function as a Service), which nowadays may not mean only that. More often Cloud Service Providers (CSPs) are making serverless services available for customers.

Van Eyk et al. (2017) defined serverless architectures as the ones being supported by 3 pillars:

- High availability and scalability provided by CSP with almost no operational management needed by the client
- Granular billing based on the utilisation of resources
- Services are event-driven

Therefore, FaaS, such as AWS Lambda, can be considered a serverless service for computing purposes.

Comparing to other models of services in the cloud, serverless may bring the attention and encourage companies to adopt it as complexity and maintenance are reduced since part of the responsibility is offloaded to the CSP, as pointed by Lynn et al. (2017).

Some benefits are worth mention when using a serverless architecture:

- Possibility to downscale to zero resources (in some cases)

- Pay only for utilised resources and for the time they were utilised
- Services can be event-driven
- Fast time to market
- Reduced operations costs
- Key enabler for microservices applications (as mentioned by Jambunathan and Yoganathan 2018)
- CSPs are utilising an infrastructure that could be otherwise idle (as mentioned by Goli et al. 2020)

2.3 Organisations, Cloud and Containers

More and more organisations are either moving to the cloud, feeling obligated to move or even developing cloud-native applications from the beginning, according to Jambunathan and Yoganathan (2018). This has been encouraged as virtualised resources tend to be cheaper, especially when owning and maintaining these infrastructures can be expensive.

The growth in the usage of the cloud continues to be demonstrated in the survey published by Flexera (2020). They were able to identify that 93% of the companies are using more than one cloud (either private or public), and that siloed applications on a single cloud are the most used architect. The most used public cloud service providers are Amazon Web Services (AWS), followed by Microsoft Azure and Google Cloud Platform (GCP). Also as part of the report, it was highlighted that 65% of companies are using Docker for containers, and there was a noticeable growth in the number of organisations utilising or planning to use CaaS from public CSPs when compared to last year, 2019. At the same time as the utilisation of containers for the development of solution grows, the biggest three challenges companies face are the lack of expertise, followed by the difficult to move applications to containers and security concerns.

During the AWS Virtual Containers Day 2020 event (*AWS VIRTUAL CONTAINERS DAY* 2020), the researchers asked if AWS had any evidence that CaaS were becoming more popular and, based on the answer, it was clear that AWS is also aware of this growing demand since they stated that "We are seeing a high level of adoption of customers using Fargate to run their containers serverlessly, as well as customers building internal PaaS's for their own developers to utilise without having to worry about the underlying infrastructure that powers the cluster".

Goli et al. (2020) described in detail how a migration from a monolithic application to a serverless CaaS structure can be done. In their analysis, they identified challenges that organisations may face when moving to serverless, such as resource limitations imposed by CSP. Their study also demonstrates the importance of a well-designed and architected solution to achieve a cost-efficient and scalable application in a serverless way, especially when the billing model relies on utilised resources for a period of time. This study also reinforces the idea that serverless applications may not suit every type of application.

2.4 Containers

Containers have been identified as the new application development paradigm by several sources (Jambunathan and Yoganathan 2018, Flexera 2020, *Cloud Run: Container to production in seconds | Google Cloud* 2020).

One of the biggest advantages of using containers is that containers are a portable and reliable way of encapsulating and running applications in different places, such as developers machines, testing or production environments, in any computer, in a private or public cloud provider that supports containers. Jambunathan and Yoganathan (2018) reinforced this idea by stating that containers are lightweight, portable, highly secured scalable and language-neutral. Containers also combine well with microservices, which allows large solutions to be broken down into smaller and independent components or services.

With the increase in the usage of containers, managing multiple applications running in different containers becomes a challenge. Configuring the network, the communication between every single container, high availability, auto-scaling and even the deployment would be harder as solutions grow. Not only the management of the containers is complex, but managing these containers running on distributed virtual or real machines (also known as nodes) is not an easy task. Then, the need for a proper orchestration of containers becomes the next challenge. In this project, the researches are going to study and compare the solutions available from CSP where the customers do not need to worry about this complex management.

2.4.1 CaaS vs FaaS

In opposition to FaaS, containers are not limited to specific runtimes or languages, as well as not having hard limits related to execution time. Applications in containers have all required dependencies and libraries encapsulated together, therefore they can run in any environment capable of running containers. This is a considerable benefit for organisations trying to avoid CSP vendor lock-in, or for organisations that require that part of their infrastructure runs on private clouds or on-premises.

Considering that the way that FaaS applications run is entirely up to the CSP, debugging and logging applications vary depending on the vendor as well. Containerised applications should be easier for developers, as the execution of containers is similar in every environment it runs, including in developers personal computers.

2.4.2 Containers as a Service (CaaS) and Serverless CaaS

Container as a Service is a cloud service model that allows clients of CSPs to configure and run containers or cluster of containers in a way that the provisioning, maintenance and security of the infrastructure and software underneath the application are entirely under the responsibility of the CSP. Inside this model, the service this research is going to focus on is the serverless CaaS.

The Serverless CaaS architecture goes one step further by removing the necessity of creating or maintaining a cluster of virtual machines to run containers and customers are only responsible for the development and configuration of the container itself. This facilitation provided by serverless CaaS helps to reduce the biggest challenges, pointed by Flexera (2020), that companies face when considering using containers.

In Figure 2, it is possible to observe the number of articles available when searching

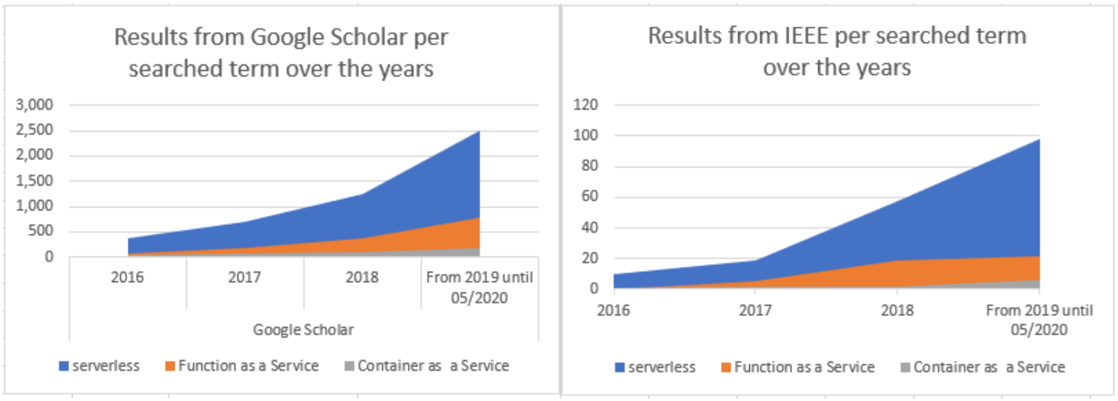
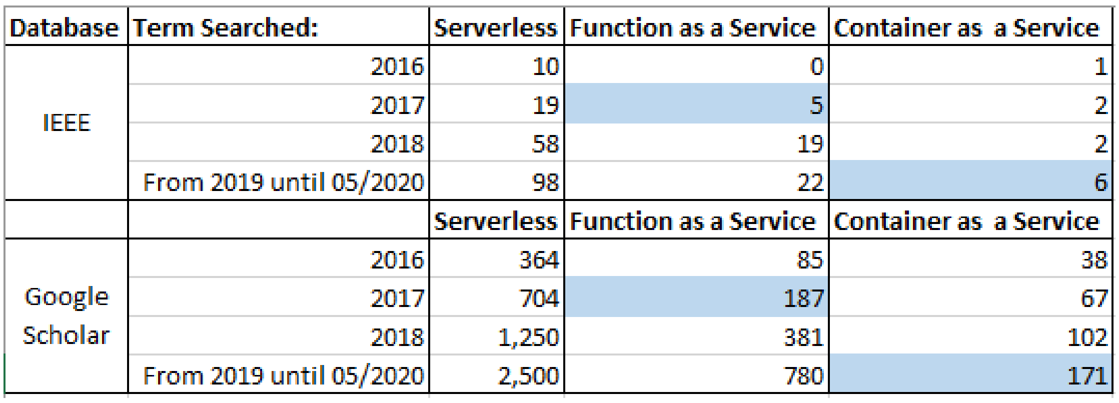


Figure 1: Visualisation of the increase of studies about CaaS

for Serverless, FaaS and CaaS in IEEE and Google Scholar databases per year since 2016. The same data is plotted in Figure 1 for a clear view of how serverless solutions are becoming more relevant and studied over the years as the number of studies is still growing.



Database	Term Searched:	Serverless	Function as a Service	Container as a Service
IEEE	2016	10	0	1
	2017	19	5	2
	2018	58	19	2
	From 2019 until 05/2020	98	22	6
		Serverless	Function as a Service	Container as a Service
Google Scholar	2016	364	85	38
	2017	704	187	67
	2018	1,250	381	102
	From 2019 until 05/2020	2,500	780	171

Figure 2: Comparing number of articles available about serverless, FaaS and CaaS

The cells highlighted in blue in Figure 2 show that in matters of studies available, CaaS has almost the same amount of data available as FaaS had in 2017, almost 3 years ago. This helps us understand why the great majority of studies about serverless performance are related to FaaS.

Given the fact that serverless CaaS may be under consideration for many companies, especially the ones already using microservices or containers, understanding better what each one of the major CSPs offers, how they work, their pricing, configurations, limitations and performance are topics that could be relevant to the community, and no pre-existing academic study covering these items was found by the time this research was idealised and developed.

2.4.3 Serverless CaaS in major CSPs

Amazon AWS Fargate was firstly announced and made available to the public in November of 2017 by *Introducing AWS Fargate* (2017). It allows customers to run containers either in EKS (for Kubernetes) or in ECS (Elastic Container Service), the latter being the one to be investigated in the research given the possibility to use the solution in a serverless manner.

AWS ensures customers only pay for resources required to run the containers without the burden of managing scaling, patching, securing or managing the servers. They also guarantee that CPU, memory, storage and network are isolated and not shared with other containers as in *AWS Fargate* (2003).

Microsoft Azure Container Instances (ACI) was in preview in February 2018 and has been publicly available since April of 2018 (*Azure updates* 2020). Microsoft makes it clear on ACI web page (*Azure Container Instances documentation - serverless containers, on demand* 2020) that the service should be simple to use by stating that "it's just your application, in a container, running in the cloud". It is also highlighted that managing the infrastructure to run the containers is not required and that deploying an application can be done with a single command.

GCP (Google Cloud Platform) Cloud Run was in Alpha in August 2018, in Beta in April 2019 but it is not clear when it was officially made available in their release notes *Cloud Run (fully managed) release notes | Cloud Run Documentation* (2020). However, it has been generally available since November 2019, being the last of the three solutions to be released. In their solution website *Cloud Run: Container to production in seconds | Google Cloud* (2020), Google states the idea that containers have become a standard to deploy applications and that customers using it will only pay for the time the application is running, also guaranteeing that the service is fully managed and that the users "can sleep well" without having to worry about it.

3 Research Methodology

3.1 Introduction

In this chapter, each one of the questions that this research proposes to find answers to are going to be detailed. For each one of them, information about the plan to test, to collect data and to display results are going to be presented alongside the initial hypothesis about possible outcomes. It is also in this section that the description of how a containerised application was developed, the justification for the chosen technologies and methods and any possible relevant information that explains and justifies the approach taken to help to fill the knowledge gap identified in Chapter 2.

AWS ECS Fargate, Azure Container Instances and GCP Cloud Run are the similar solutions available in the major public cloud service providers when this proposal was initially written. These three serverless CaaS solutions will be the ones utilised during this study. However, the experimentation to be described in this chapter was designed thinking about a possible future expansion of the research, therefore, it should be reproducible in similar products from other CSPs. The researchers will script the provision of necessary infrastructure as well as the execution of the tests that will be utilised to collect the data, aiming to make the produced artefacts and the results publicly available once the research is finished.

3.2 Research questions, hypotheses and methodologies

The research will be mainly quantitative and from primary sources. It will include the development of an application, execution of experiments, collection of data, analysis of collected data and presentation of findings in a concise and illustrative manner. The only qualitative question will be 3.2.1, which will also rely on secondary resources.

3.2.1 Are the serverless CaaS solutions tested truly serverless?

In the solutions' websites (*AWS Fargate* 2003, *Azure Container Instances documentation - serverless containers, on demand* 2020 and *Cloud Run: Container to production in seconds | Google Cloud* 2020) the products are advertised as serverless solutions. However, the study aims to confirm whether each one of them covers what was stated by Van Eyk et al. (2017).

The hypothesis is that the answer will be yes, however, some CSPs may have containers running at all times, even when not required.

To confirm the hypothesis, a research will be done to investigate in depth the availability, scalability, billing and events capable of triggering and running serverless CaaS. This study aims to identify whether clients are charged for idle or non-utilised resources. As a result, a confirmation is going to be stated declaring whether each CaaS subject can be considered a serverless solution based on the three pillars stated by Van Eyk et al. (ibid.). As the study demands a deep investigation about the serverless CaaS solutions, the study also aims to highlight similarities and differences between the solutions.

3.2.2 Which serverless CaaS solution has the best performance?

Are there any differences between CSPs for an almost empty application, such as a "hello world"? What about an application that demands a high CPU utilisation? Does the time an application is executed affect its response time?

To answer these questions, alongside the study, an application will be developed and made available with two different endpoints, one which is almost empty containing only a "hello world" output, which will be called Hello World Endpoint and the other containing a high CPU task, by calculating factorials for given numbers, which will be called High CPU Endpoint. This application will be uploaded to a container registry.

This experiment will be performed by following the steps:

1. Deploy the application to each one of studied CSP

2. For each one of the possible variations of configuration (CPU and memory):
   (a) Configure the application in each CSP
   (b) Trigger/Execute the application several times. (Run in different hours of the day and during several days. All CSP solutions will be tested at the same time or as close as possible).
   (c) Collect metrics

The main metric to be collected in this test is the application execution time, which should be included in the application to be developed as part of the study.

The hypothesis for this question is that for the almost empty application, the results will be similar in all CSPs. For the one that requires CPU, GCP might have the best performance, as Kubernetes was originally designed by Google. However, as AWS Fargate is the most mature tool, being in the market since 2017, AWS may present the best results.

3.2.3 Which serverless CaaS solution is the cheapest?

The objective of this question is to identify whether a solution is more recommended than another when pricing is the most important aspect to be taken into consideration. The research also wants to identify if there are free tiers available for each one of the studied solutions, and how much each one of them costs according to a specific demand.

To calculate this, the billing information provided in official documentation will be used to plot charts with hypothetical scenarios with different sets of variables, such as execution time, number of executions, number of CPU and memory required.

The study aims to compare the pricing while ignoring any possible performance differences provided by the services and consider that containers will perform exactly the same in each of one of the providers. However, as measurements from the application performance are also going to be collected in another studied question, the study is also going to identify how much it costs to run the tests and how much would it cost to run similar workloads based on the performance from the serverless CaaS solutions. Comparisons will be made taking into account scenarios with and without free tiers.

Results obtained for this question could be later combined with question 3.2.2 to identify what service has the best cost/benefit ratio.

3.2.4 Which serverless CaaS solution to recommend based on the application utilisation?

Conclusions obtained for this question will be exclusively related to the results obtained from the whole study when analysing our application with two endpoints (Hello World and High CPU) running in the serverless CaaS solution as well as some hypothetical scenarios.

Based on the results obtained from Questions 3.2.1, 3.2.2 and 3.2.3, this question aims to compare and define which CSP would fit best to deploy applications with functionalities similar to the endpoints, taking into consideration price, performance and unique functionalities.

The hypothesis is that an application similar to the Hello World endpoint will fit well in any CSP and the cheapest one would be the recommended one. For an application similar to the High CPU endpoint, the price will only be used to define the best solution if performance results are equivalent between the solutions.

The results of this investigation will be presented alongside each one of the answered Questions 3.2.1, 3.2.2 and 3.2.3.

3.2.5 Plan to find answers

For the quantitative questions, a generic plan was designed to guide the study during the implementation of the tests. An overview of how the study aims to perform the tests is illustrated in Figure 3.

The plan starts from the Setup layer, and the first component is the containerised application. This application will have two different endpoints, one that simply outputs a "Hello World" text, and another that demands the utilisation of CPU. This containerised application would then be stored in a Container Registry, so

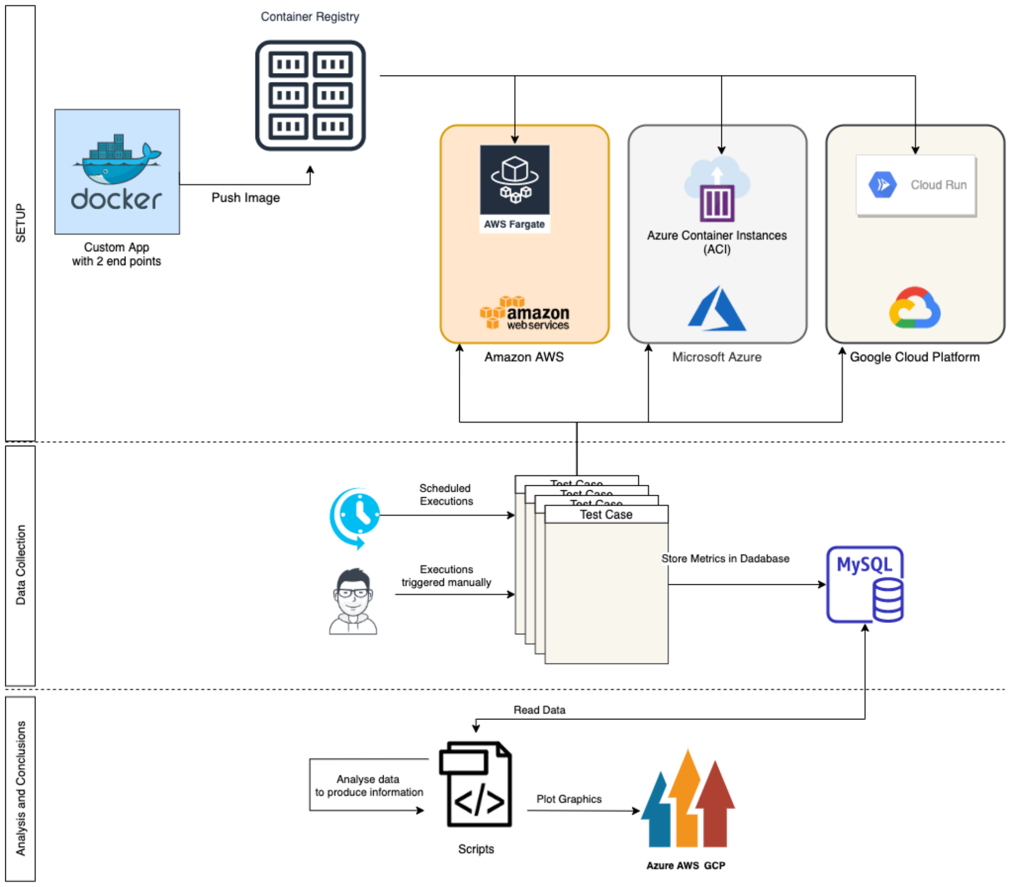


Figure 3: Overview of proposed strategy in three vertical layers. First the setup, second the execution of tests to collect data, and third to do analysis and make conclusions.

that it can be pulled and utilised in each of one the CSPs CaaS to be assessed (AWS ECS Fargate, Azure Container Instances and GCP Cloud Run).

In the Data Collection layer, once the containerised application is running in the serverless CaaS solution in each CSP, test cases can be manually or programmatically executed to assess the performance. These tests are going to generate metrics, which are going to be stored in a centralised database.

The last part is the Analysis and Conclusions layer. Once collected metrics are available in the database, algorithms can be developed to filter data, extract meaningful information and plot some graphics to help the study to find standards and trends used to state conclusions.

3.3 The Containerised Application

To be able to assess and compare the performance between the CSPs, it was defined that a containerised application should be created and deployed in each one of the providers. Initially, the plan was to develop two different applications, however, having a single application with two endpoints offers an easier way to manage the experiment without compromising the results. A visual representation of the components of the application can be seen in Figure 4.

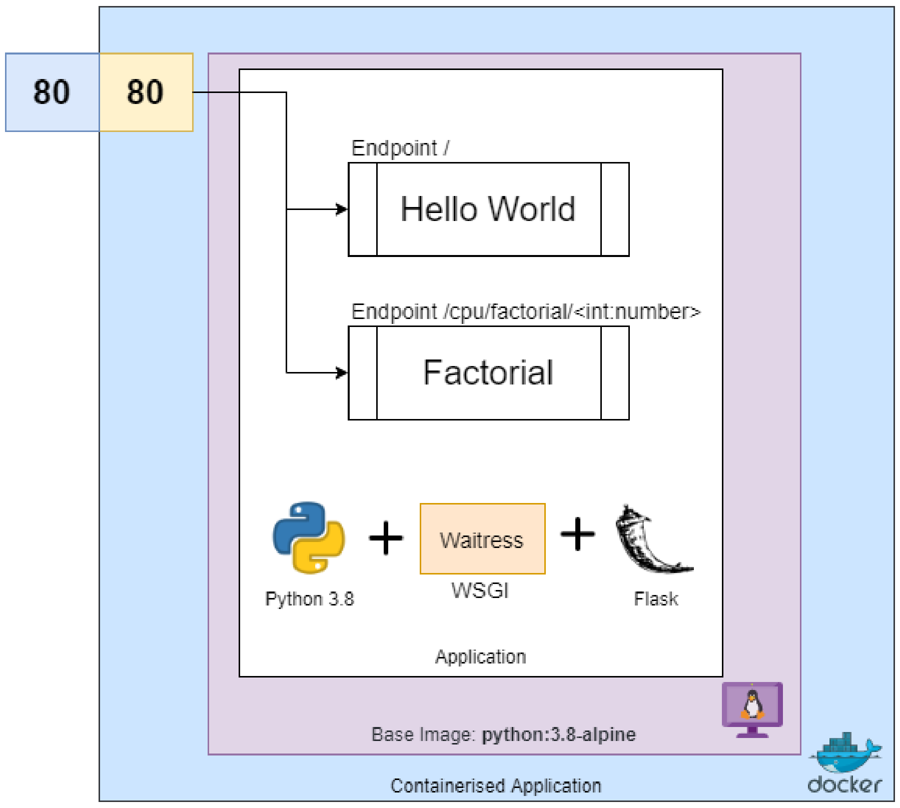
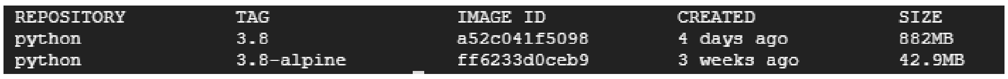

80
80
Endpoint /
Hello World
Endpoint /cpu/factorial/<int:number>
Factorial
Python 3.8
+
Waitress
WSGI
+
Flask
Application
Base Image: python:3.8-alpine
Containerised Application
docker

Figure 4: Visualisation of the containerised application and its technologies

The application was containerised using Docker and was based on an official Python Docker image that uses the Alpine Linux. The Alpine distribution was chosen instead of the default one (Debian distribution), as the Alpine distribution is lightweight (at least 20 times smaller than the default as shown in Figure 5) but still contains the necessary base libraries to run the Python application (Inc 2020). This means that the final image would also be smaller and downloading it would also be faster, directly reducing the deployment time without compromising the final result.



```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
python              3.8                 a52c041f5098        4 days ago          882MB
python              3.8-alpine          ff6233d0ceb9        3 weeks ago         42.9MB
```

Figure 5: Comparison between sizes of Python default Docker image and Alpine image

Python 3.8 was the language chosen to run the containerised application. Other languages could be used, but Python was chosen as it is still one of the most popular technologies (Stackoverflow 2020) also offering flexibility to run for different purposes.

The application is going to be exposed as a web application so that it is easy to interact with it by performing HTTP requests. The application itself was developed using Python (version 3.8) and the web microframework Flask (version 1.1.2), being served by Waitress (version 1.4.4), a production WSGI (Web Server Gateway Interface) recommended by Pallets (2020).

Similarly to the programming language, other frameworks or even only native code could be used. However, Flask was chosen as it is lightweight and easy to use when implementing a web application. The items in Listing 1 are the external dependencies (following Python dependencies format) that the application relies on.

```
Flask==1.1.2
waitress==1.4.4
```

Listing 1: Python external dependencies required for the containerised application.

This application exposes two endpoints via port 80. Any port could be utilised, but as it is a web application, port 80 was chosen as it is commonly used for this purpose.

By confirming that, by default, nor Flask nor Waitress cache any of the requests, this study also guarantees that results obtained from the application are not cached on the server-side, avoiding the utilisation of wrong measures from previously computed results.

As the application will have the exact same code, the same libraries and dependencies, it is expected that the application is going to behave similarly in any environment where it is deployed.

Once the application was developed, it was containerised by the creation and execution of a Dockerfile. The Dockerfile describes a list of commands to be executed for the creation of a container image. The contents of the Dockerfile can be found in Listing 2.

```
1 FROM python:3.8-alpine
2 COPY ./app /app
3 WORKDIR /app
4 RUN pip install -r requirements.txt
5 CMD [ "python", "app.py" ]
```

Listing 2: Contents of Dockerfile utilised to create docker image for the containerised application.

The first line specifies that the image to be created will be based on the existing Python 3.8 using Alpine Linux image. Then, the application source-code folder is copied into a folder (/app) inside the image. The next step, in line 3, defines that the start working directory for the new image will be the application directory. The following step is to install all the external dependencies the application relies on (Flask and Waitress) into the image. The list of dependencies is inside the file requirements.txt. The very last command specifies that once a container based on this new image starts, it will execute the application file (app.py) using Python.

Once built, the application image was hosted in Docker Hub, a free to use and public container registry. The application developed in this study can be found in the URL address `https://hub.docker.com/r/viniciusbarros/msc-performance-web-app`.

3.3.1 Running the application

Once containerised, the application can run in any environment that supports Docker Containers. Before running it, its image needs to be download from the registry. It can be done by performing the command specified in Listing 3.

```
docker pull viniciusbarros/msc-performance-web-app:latest
```

Listing 3: Docker command to download (or pull) the application image from Docker Hub registry.

To have it running directly in a machine with Docker the command described in Listing 4 needs to be executed.

```
docker run -p 80:80 msc-performance-web-app:latest
```

Listing 4: Docker command to create and run a container based on the image downloaded from Docker Hub registry.

Note that the parameter "p" means the port mapping from the host machine to the containerised application. The first number before the colon punctuation (80) can be replaced by any port available in the host machine.

Once the application is running, it can be accessed by performing HTTP requests to http://localhost:80.

3.3.2 Hello World Endpoint

This endpoint can be accessed by performing a GET HTTP request to the root (/) of the application. It returns a JSON (JavaScript Object Notation) text with a "Hello World" message, and how long the application took (in seconds) to attribute the hello world string into a variable, as exemplified in Listing 5.

```
{
  "data": "Hello World MSC - Container as a Service Comparison",
  "duration": 0.0000112
}
```

Listing 5: Example of output for Hello World endpoint.

3.3.3 High CPU Endpoint

The second endpoint available in the application can be accessed by performing a GET HTTP request to the endpoint "/cpu/factorial/<int:number>", where the <number> is an integer parameter. As the name of the endpoint suggests, it calculates and outputs the factorial for a given number.

As the calculation of factorial depends mainly on the utilisation of CPU, the goal of this part of the application is to check if similar inputs have similar responses in different CSPs. An example of the output can be seen in Listing 6 for the input "10" in the number parameter.

```
{
  "data": "Factorial of 10 is: 3628800",
  "duration": 0.0000224,
  "end": 1601135044.6502888,
  "start": 1601135044.6502664
}
```

Listing 6: Example of output for High CPU (factorial) endpoint.

3.4 Methodology to assess the application performance

Details about how the research intends to assess the performance of the application are going to be detailed in this section. Firstly, the performance aspects to be analysed are going to be detailed. Then, a description of how the study was set up and automated for the execution of tests, justifying chosen applications and methodologies. Finally, more information will be presented in regards to the metrics to be collected, how they are going to be used and what plots will be programmatically generated.

3.4.1 Defining performance from the application point of view

The performance from the application (or back-end) point of view takes into consideration how long the containerised application running in a serverless CaaS takes to process a request.

This metric is calculated and outputted by both endpoints in the application, Hello World and High CPU. To measure this value, right before the application starts computing, the current time is captured (start). Right after the request is processed, the current time (end) is captured once again. The performance measurement is calculated based on the difference between end and start times, which tells us how long the application took to perform a specific task. It is important to highlight that this time only takes into consideration the time needed for the application to compute the result. Therefore, network or other times before and after the processing are not considered.

When comparing the serverless CaaS solutions, the CSP that has the fastest time will be defined as the one with the best performance from the application point of view.

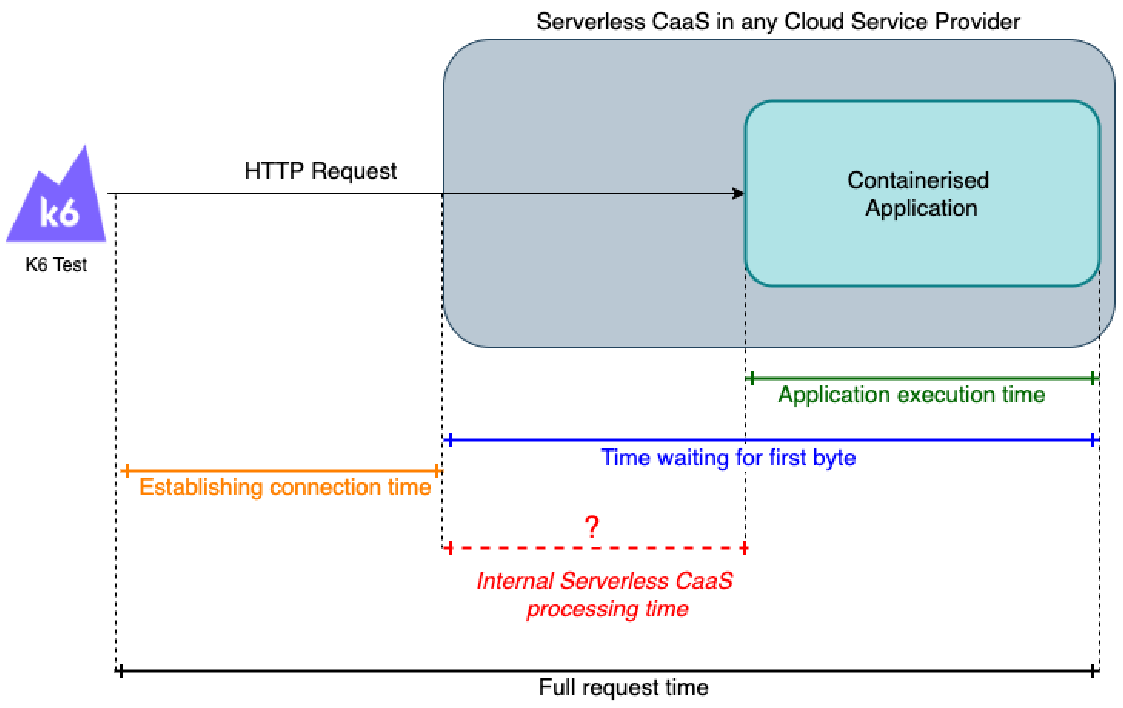
To exemplify, let us hypothetically consider that the application running with 1 CPU and 2GB of memory took 2.2 seconds to calculate the factorial of 60000 in CSP A, and took 2.6 seconds to run the exact same scenario in CSP B. In this example, CPS A has the best performance from the application point of view as it was 0.4 seconds faster than CSP B.

3.4.2 Defining performance for the internal CaaS processing time

Given the fact that the study is capable of to assessing the application performance based on the processing metrics the application is outputting, and considering that, with K6, the waiting time for the first byte of a request is also available, it is possible to estimate how much of the processing time is effectively being utilised for internal processing of the tested serverless CaaS. With this calculation, it is possible to measure how long each serverless CaaS takes to receive, forward and return a request processed by the containerised application. This is going to measure the performance of the part of the infrastructure that is under the CSP responsibility. This time the research aims to calculate is the one represented in colour red in Figure 6, which illustrates a simplified way of measuring the processing times in an HTTP request.

Considering the metrics collected from K6 in combination with the custom metric generated by the application, the time that is important in this section can be estimated by subtracting the application execution time (app_execution_time) from the time waiting for a response after the connection is established, also known as "time to first byte" (http_req_waiting time).

Even though the application execution time (calculated and outputted by the application) only takes into consideration a part of the application, it is expected that this is the part of the application that consumes the majority of the processing time. Having this clarified, it is possible to measure the difference and have a primitive way of knowing which embed serverless CaaS overhead processing is the fastest.



Serverless CaaS in any Cloud Service Provider
k6
K6 Test
HTTP Request
Containerised Application
Application execution time
Time waiting for first byte
Establishing connection time
?
Internal Serverless CaaS processing time
Full request time

Figure 6: Illustration of a simplified way of measuring processing times. The dashed line in colour red is the processing time to be calculated to check which serverless CaaS has the best performance, for the infrastructure part that is under the CSP responsibility. The green line is the processing time collected from the application. The blue line is the waiting for first byte metric already available from k6.

The serverless CaaS that has the smallest Serverless CaaS internal processing time will be defined as the one that has the best internal processing performance.

3.4.3 The setup

To be able to run the tests as many times as possible, and considering that one of the goals of this study is to ensure that tests can be reproducible by the community, most of the setup was automated and created as code. Figure 7 illustrates the components to be described in this section as well as how they are connected.

Terraform was the tool utilised for provision and destruction of the infrastructure necessary in AWS, Microsoft Azure and Google Cloud Platform. The facts that it is open source and free to use (HashiCorp 2020) were not the only reasons considered for this choice. Terraform is also cloud-agnostic, which allows us to use the same coding pattern to configure, deploy and destroy environments without the need to deal with proprietary languages or CLIs (Command Line Interfaces) provided by each CSP. Furthermore, Terraform has compatibility with major and smaller CSPs, which could allow a further expansion of the scope of this study, should more serverless CaaS solutions are made available.

A Terraform template was created for each one of the CSP, alongside with a configuration file, which allows us to deploy the application using different sets of configurations, including different amounts of CPU, memory and geographic region to be utilised. As each serverless CaaS in each CSP has its own peculiarities, each Terraform file created for each provider is different. In AWS, for instance, before defining the task that will run the containerised application it is also required to create a cluster and a service to expose the task.

The Terraform templates created for AWS ECS Fargate, Azure Container Instances and GCP Cloud Run can be found respectively in Appendices S, V, R, U, T, W.

K6 was the tool chosen to automatically test the containerised application and extract performance metrics. It is also open source and free to use (AB 2020a). With K6, it is possible to create reproducible test cases as code, using JavaScript programming language, and scenarios that can be executed as many times as needed.

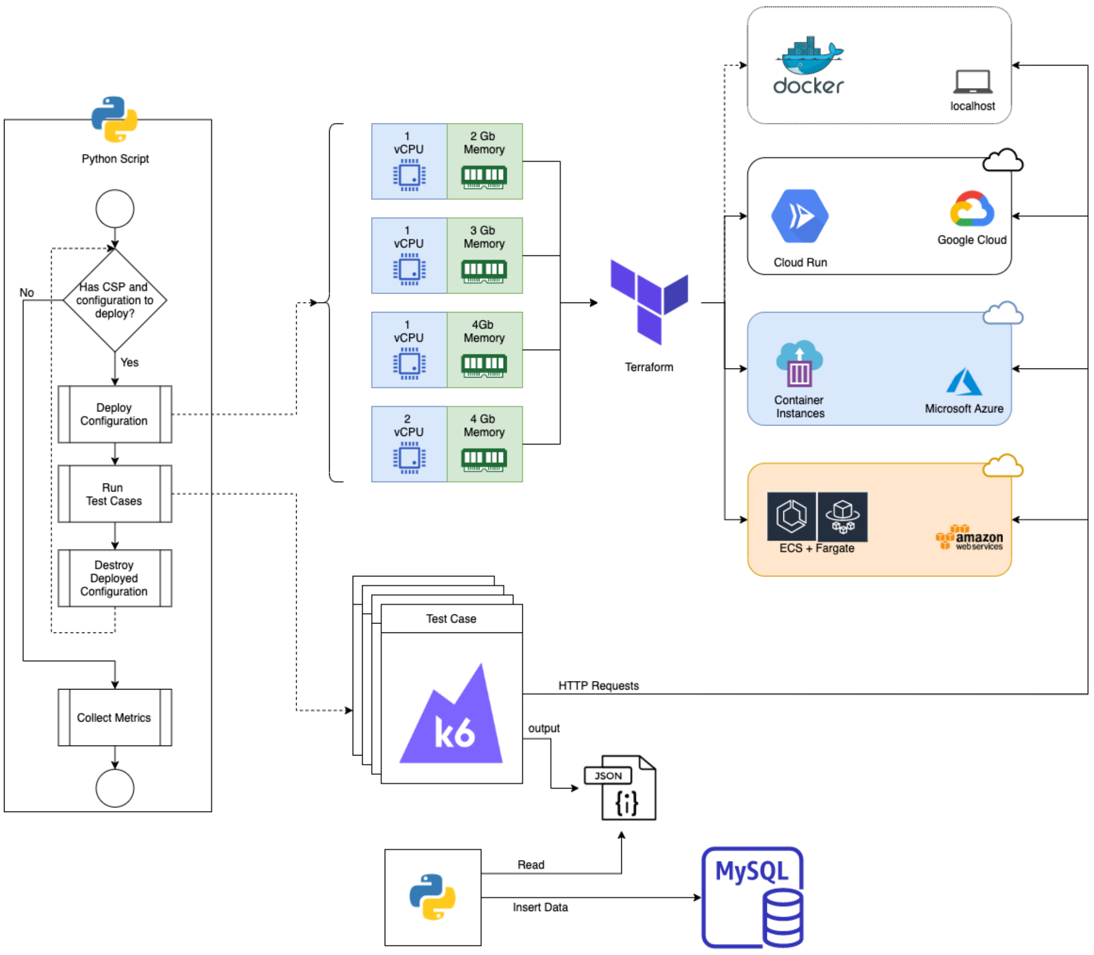


Figure 7: Visualisation of the setup created to perform tests and collect metrics

To measure the performance of the Hello World endpoint, a K6 test case was created to simulate a single user (1 VU or virtual user) performing 10 requests one after another, to the endpoint. The code for this test can be found in Appendix F.

For the High CPU endpoint, another K6 test case was created, also using 1 VU and performing 10 requests, one after another. The goal is not to stress test the application, but to measure its performance when all resources are available for the application to run. The code for this test can be found in Appendix E.

The orchestration to provision, perform the tests and collect metrics was scripted and developed using Python 3.8. The source code for the Python script responsible for the orchestration and execution of the tests can be found in Appendix M.

MySQL was selected to be the database for storing the metrics in a way it could be analysed, queried and filtered. Any database could be utilised for this purpose, however, MySQL was chosen as it is open source with a free to use version and also because of the possibility of reading data from it directly from Grafana, which is going to be utilised for the initial analysis. To insert data into the database, another Python script was developed and this script source code can be found in Appendix L.

3.4.4 Running the tests

This section will contain details about how the tests are executed, as illustrated in Figure 7. It is important to highlight that Docker was included as one of the tested subjects, as metrics from executions against it are going to be used as a baseline for some comparisons.

The journey starts by executing the Python script. This script holds information about the CSPs and also has each one of the configurations to be deployed to the providers, as detailed in Table 10. Then, for each one of the possible configurations, the script will:

1. Deploy the containerised application into the serverless CaaS for each CSP, using Terraform, specifying one of the possible configuration combinations (Table 10);
2. Collect the generated public URL that allows us to interact with the application;
3. One after another, trigger the K6 tests to assess the application using the endpoints:
    (a) Hello World endpoint
    (b) High CPU endpoint calculating factorial of 10
    (c) High CPU endpoint calculating factorial of 1000

    (d) High CPU endpoint calculating factorial of 10000
    (e) High CPU endpoint calculating factorial of 32000
    (f) High CPU endpoint calculating factorial of 43000
    (g) High CPU endpoint calculating factorial of 50000
    (h) High CPU endpoint calculating factorial of 60000

4. Store metrics obtained from the tests in a JSON file for each test case;
5. Destroy the deployed infrastructure using Terraform

Once the tests are executed, JSON files with metrics for each test cases are generated. The second Python script (Appendix L) is used to read each one of the files, normalising the data and saving it into the MySQL database.

To speed up the testing process and to ensure the tests are executed almost at the same time in all CSPs, the script was optimised to accept a CSP as a parameter, allowing us to execute tests in parallel instead of in a roll. Figure 8 demonstrates the concept utilised to run the same script in parallel and how metrics were collected.

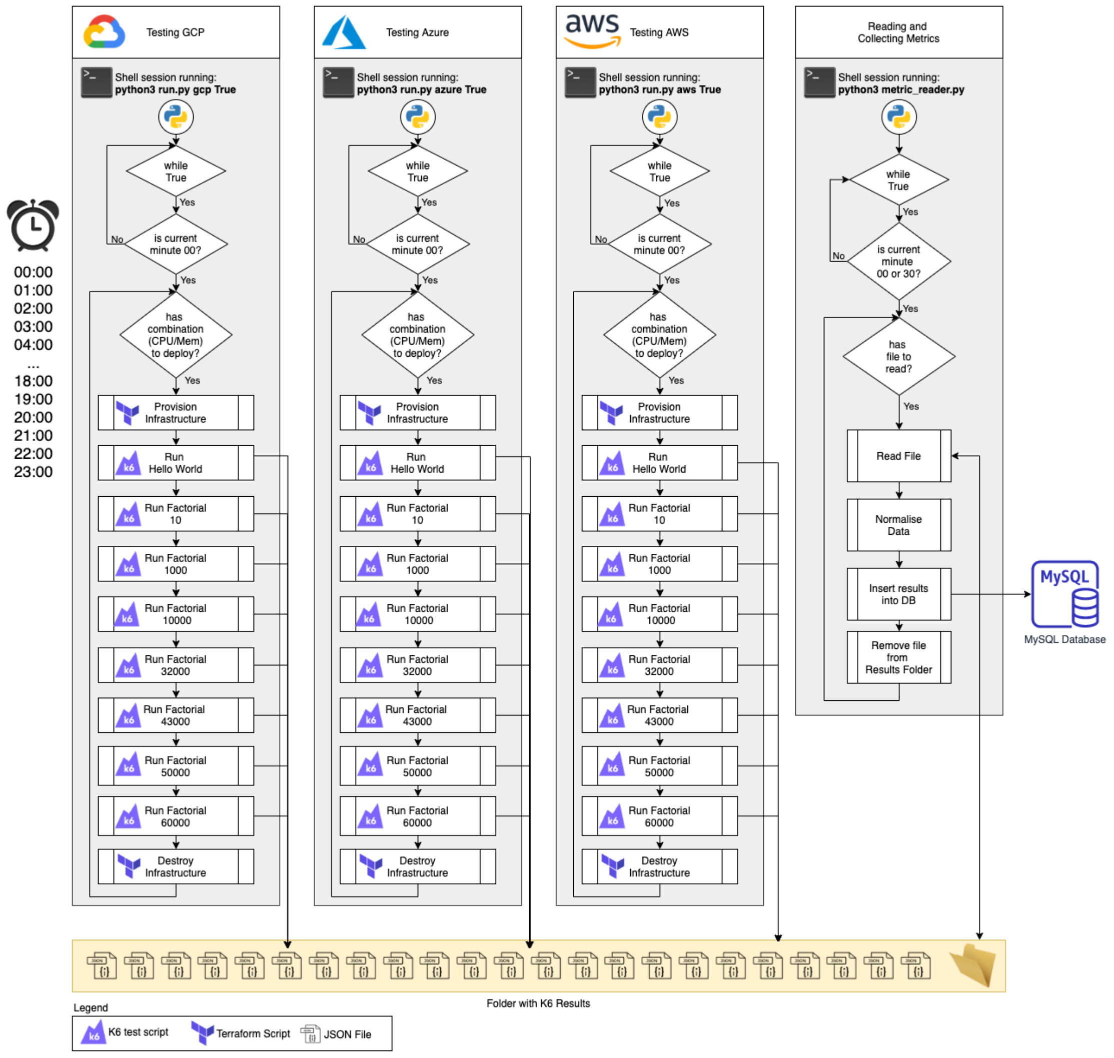


Figure 8: Explanation of parallel tests execution and collection of metrics. The first (from the left to the right) 3 grey boxes represent a shell session running automation to deploy the serverless CaaS and run K6 tests against each CSP and outputting JSON files into a shared folder. The last grey box represents the script responsible for reading the JSON files, normalising the data and finally save it in a MySQL database. The clock and hours on the left side, symbolise that the scripts will be constantly running but performing tasks only at specific times.

3.4.5 Metrics collected

Table 1 holds information about the variants for each one of the metrics the automated tests are collecting.

In addition to the default metrics generated by K6 (described in Table 3), the automation is also collecting some extra data from the test cases and from the data that is outputted by the containerised application (Table 2).

Metric Variant	**Suffix**	**Description**
Average	avg	Average among collected values
Maximum	max	Greatest value among collected values
Minimum	min	Smallest value among collected values
90th %	p90	90% of the requests were within this value
95th %	p95	95% of the requests were within this value
Count %	count	Total of a certain collected value
Rate %	rate	Amount of data/time utilised for a given metric

Table 1: Possible variants present in some of the collected metrics.

Metric	**Variants**	**Description**
datetime		Date and time when the test was executed
csp		Cloud Service Provider tested
type		Type of test (hello-world or factorial)
cpu		Amount of CPU's utilised in the containerised application tested
memory		Amount of memory (in GB) utilised in the the containerised application tested
app_execution_time	avg, max, min, p90, p95	Execution time from the application point of view

Table 2: Custom metrics and its variations collected for each one of the performed tests.

Metric	**Variants**	**Description**
data_received	count, rate	Amount of data received during test
data_sent	count, rate	Amount of data sent during test
http_req_blocked	avg, max, min, p90, p95	Time spent waiting for a free TCP connection before performing the request
http_req_connecting	avg, max, min, p90, p95	Time spent connecting with remote host
http_req_duration	avg, max, min, p90, p95	Total time spent during the execution
http_req_receiving	avg, max, min, p90, p95	Time spent receiving data from remote host
http_req_sending	avg, max, min, p90, p95	Time spent sending data to remote host
http_req_tls_handshaking	avg, max, min, p90, p95	Time spent during TLS handshaking with remote host
http_req_waiting	avg, max, min, p90, p95	Time waiting for an http response
http_reqs	count, rate	Number of HTTP requests performed
iteration_duration	avg, max, min, p90, p95	Duration of a full request performed
iterations	count, rate	Number of times requests were performed
vus	max, min, value	Number of virtual users utilised
vus_max	max, min, value	Maximum possible number of virtual users

Table 3: K6 default metrics and its variations collected for each one of the performed tests (AB 2020b).

In summary, performance tests were executed between 02/10/2020 and 26/10/2020. Test cases were executed 208 times against a local machine to find the baseline for comparison. 9007 times against AWS, 8168 against Azure and 9490 against GCP. Initially, the tests were manually triggered, but to achieve consistency and aiming to have enough data so average times could be reliable, the tests were executed programmatically, at every hour of the day, from 15/10/2020 until 26/10/2020. The distribution of the execution of performance tests can be seen in Figure 9.

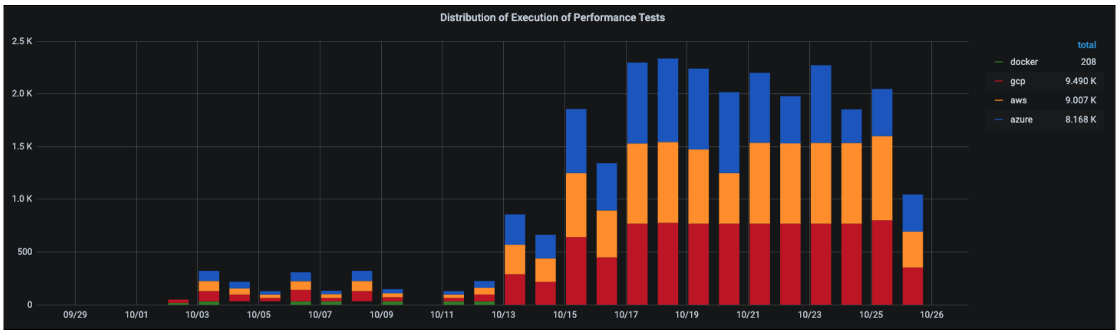


Figure 9: Distribution of performance tests over time, from 02/10/2020 to 26/10/2020. Each bar represents a day. Each colour in each bar represents the number of executed tests in a specific platform. Green parts are executions performed against local environment running Docker, red parts represent tests run in GCP Cloud Run, Orange parts represent tests run in AWS ECS Fargate and blue parts represent tests executed in Azure Container Instances.

Despite the effort to make sure that all test cases were successfully executed at all times, unfortunately, tests were interrupted a few times for different reasons, including the machine running the tests being turned off unexpectedly, K6 test failing to perform the tests for unknown reasons and CSP not making resources available as the automation script was expecting. As a result, it is possible to observe that the number of executed cases were not exactly the same for all CSPs, however, it helped the researches identifying parts of the script that needed improvements to make it more reliable. At last, it is important to highlight that test cases that completely failed for the reasons mentioned above were not collected or registered in the database.

3.5 Defining how Graphics will be plotted

To ensure researchers can start analysing the data since it starts being collected, and given the fact that the collected data is going to be stored in a MySQL database, the group decided to start analysing the data utilising Grafana. Grafana is a web-based application with an open-source and free to use version (*Grafana* 2020) that allows users to read data from different sources, including MySQL, and visualise metrics in a variety of types of charts. By being able to easily plot and visualise some data with Grafana, some initial visual trends may be spotted and used to lead more in-depth investigations.

For more complex plots, the group decided to script the creation of graphs as much as possible, so that they can be generated or re-utilised should more data is collected in the future.

As the containerised application is also written in Python, it was decided to script the generation of graphs utilising Python 3.8 combined with the library Matplotlib (https://matplotlib.org) that allows us to create custom charts. The group also decided to expose this tool to plot charts on-demand as a web application, using Flask and Waitress as well. The code for generating these charts is also hosted in the GitHub repository alongside the application. A list of charts to be plotted programmatically can be visualised in Tables 4, 5 and 6.

Type	**Endpoint**	**Data**	**Combination**	**Total**
Performance	Hello World	min, max, avg, baseline	A,B,C,D	4
Performance	Factorial 10	min, max, avg, baseline	A,B,C,D	4
Performance	Factorial 1000	min, max, avg, baseline	A,B,C,D	4
Performance	Factorial 10000	min, max, avg, baseline	A,B,C,D	4
Performance	Factorial 32000	min, max, avg, baseline	A,B,C,D	4
Performance	Factorial 43000	min, max, avg, baseline	A,B,C,D	4
Performance	Factorial 50000	min, max, avg, baseline	A,B,C,D	4
Performance	Factorial 60000	min, max, avg, baseline	A,B,C,D	4
Performance	Factorial 10	avg only	A,B,C,D	4
Performance	Factorial 1000	avg only	A,B,C,D	4

Performance	Factorial 10000	avg only	A,B,C,D	4
Performance	Factorial 32000	avg only	A,B,C,D	4
Performance	Factorial 43000	avg only	A,B,C,D	4
Performance	Factorial 50000	avg only	A,B,C,D	4
Performance	Factorial 60000	avg only	A,B,C,D	4

Table 4: List of 60 performance charts that are going to be scripted so they can be plotted on demand. Combination column related to Table 10.

Type	**Free Tier**	**Duration/Scenario**	**Combination**	**Total**
Pricing per CSP	No	1 hour/day/1 day	A	1
Pricing per CSP	No	1 hour/day/7 days	A	1
Pricing per CSP	No	24 hours/day/7 days	A	1
Pricing per CSP	No	24 hours/day/365 days	A	1
Pricing per CSP	Yes	2.5 hours/day/20 days	A	1
Pricing per CSP	Yes	5 hours/day/20 days	A	1
Crossing Prices	Yes	Hours running in a month	A	1
Crossing Prices	Yes and No	Factorial of 60000 with real performance metrics	A	1

Table 5: List of 8 pricing charts that are going to be scripted so they can be plotted on demand. The free tier column means that the free tier was taken into consideration in the graph data. Combination column related to Table 10.

Comparison	**Type**	**Total**
Weekend vs Weekdays	Application Execution Time (App. point of view)	1
Weekend vs Weekdays	Internal CaaS Processing Time	1

Table 6: List of 2 pie charts that are going to be scripted so they can be plotted on demand. These pie charts will display whether executions on weekends are faster or slower when compared against weekdays executions.

4 Findings

4.1 Introduction

This section is going to cover details about how the study answered each one of the questions proposed in chapter 3. Before answering the questions, a better understanding of each one of the serverless CaaS solutions will be presented. Then, the analysis for each one of the questions is going to be presented followed by the outcomes of the study and conclusion.

4.2 Understanding the studied CSP solutions

4.2.1 AWS ECS Fargate

It is important to highlight that the study is only analysing Fargate from the perspective of a serverless solution, where Fargate will be used to run the containers in an infrastructure fully managed by the AWS. Before running a container in Fargate, first, users are required to set up a Cluster, a Service, a Task Definition and finally the Container Definition. One or more containers can run in a Task Definition, which can be exposed as part of a Service included in a Cluster. A visualisation of this structure can be found in Figure 10.

The cluster is where it is possible to define the VPC (Virtual Private Cloud) and subnets where the services will be deployed.

The Service is what exposes a task and its containers so they can be utilised as a web application, for instance. In the service configuration, it is possible to configure the number of desired tasks to be running, starting from 1, and load balancing if needed.

The load balancing is achieved by setting up the provisioning of an AWS Application Load Balancer, which is another paid service provided by AWS, which can be configured to automatically forward requests to the service. It is important to highlight that this extra AWS service is not going to be utilised in this study.

In the Task definition, it is possible to set up maximum limits for CPU and RAM that will be available for the containers to run inside it. It is also there that all

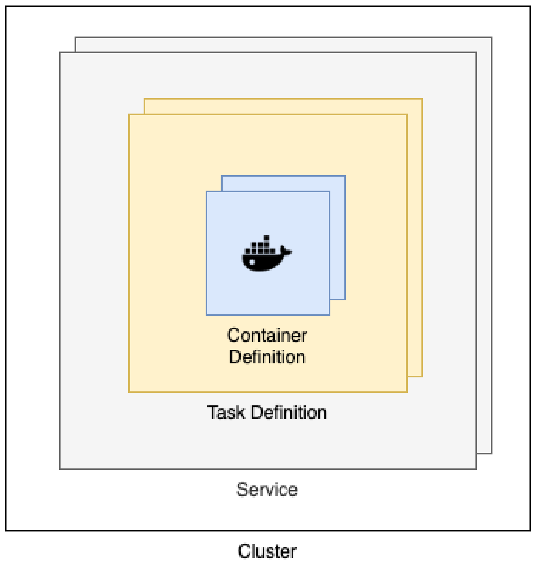


Figure 10: AWS Fargate - Cluster, Service, Task and Container Definitions' structure

networking configuration and the IAM (Identity and Access Management) roles are defined for the containers inside this task.

The amount of memory that can be allocated to a task depends directly on the number of vCPUs chosen, as shown in Table 7

vCPU	**Memory (GB)**
0.25	0.5, 1, 2
0.5	between 1 and 4 (integer and inclusive)
1	between 2 and 8 (integer and inclusive)
2	between 4 and 16 (integer and inclusive)
4	between 8 and 30 (integer and inclusive)

Table 7: Possible combinations of CPU and Memory to be used in an AWS ECS Fargate Task Definition.

The Container Definition consists of the definition of the container image to be used (from a registry), how many units of CPU (1024 per core) will be reserved and how much Memory (soft or hard limits) will be allocated for the container, considering this amount is available in the Task definition as well.

When compared to the other two solutions, AWS ECS Fargate is the one that provides the most granular level of configuration and customisation.

By default, ECS Fargate does not provide an URL that can be used to access the container that is exposed by the public accessible service. However, it is possible to associate a domain name to point to the service using another AWS Service, Route 53.

The other way to obtain an URL to be able to interact with the containerised application, which was the way utilised in this research, consists in searching in AWS Network Interface for the Public DNS name automatically generated and associated with the containerised application as in Listing 7.

```
1 {
2     "NetworkInterfaces": [
3         {
4             "Association": {
5                 "IpOwnerId": "amazon",
6                 "PublicDnsName":
                   ↪  "ec2-18-232-131-211.compute-1.amazonaws.com",
7                 "PublicIp": "18.232.131.211"
```

Listing 7: Example of partial response obtained when querying for network interface to identify PublicDnsName (line 6) for containerised application running in AWS ECS Fargate.

4.2.2 GCP Cloud Run

To get a containerised application running in Cloud Run, a new service needs to be created by informing which image should the new container use, the amount of CPU and memory, and the port the created container uses to receive requests.

When compared to the other two solutions, Cloud Run is the only one that does not allow you to use Docker Hub as container registry, therefore, to use a custom image, the image needs to be uploaded to a GCP Container Registry.

The combinations of CPU and memory available for Cloud Run can be checked in Table 8.

vCPU	Memory (MB)
1	between 128 and 4096 (inclusive)
2	between 128 and 4096 (inclusive)

Table 8: Possible combinations of CPU and Memory to be used in a Cloud Run Service.

A few extra options available include the number of concurrent requests per container (default to 80) and automatic auto-scaling from 0 to 1000 instances, with the latter being able to be increased by submitting a request to GCP.

Once set up, Cloud Run gives you an URL that can be used to perform HTTP requests and interact with the service. When comparing to the other serverless CaaS, this is the only solution that offers, by default, a secure HTTPS endpoint for interaction with the containerised application.

4.2.3 Azure Container Instances

Configuring a container in Azure Container Instances is similar to configuring one in GCP Cloud Run. Customers can specify the container image to be used, the amount of CPU and Memory. The combinations of CPU and Memory can be checked in Table 9.

vCPU	Memory (GB)
1	between 0.5 and 16 (integer or decimals)
2	between 0.5 and 16 (integer or decimals)
3	between 0.5 and 16 (integer or decimals)
4	between 0.5 and 16 (integer or decimals)

Table 9: Possible combinations of CPU and Memory to be used in a Container Instances container.

An interesting feature that is available in Azure CI allows customers to give their project a custom DNS (Domain Name Service) prefix. This URL can later be utilised to interact with the application by performing HTTP requests, following the pattern:

http://{YOUR_CUSTOM_NAME}.{HOSTING_REGION}.azurecontainer.io.

4.2.4 Overlapping Configurations

After analysing each one of the possible configurations in each one of the CSPs, a list of overlapping configurations supported in all 3 CSPs was grouped in Table 10. These are the different configurations that are going to be used in this study.

Combination	**vCPU**	**Memory (GB)**
A	1	2
B	1	3
C	1	4
D	2	4

Table 10: Possible combinations of CPU and memory that can be used in all studied CSPs (AWS ECS Fargate, Azure Container Instances and GCP Cloud Run).

4.3 Are the solutions tested truly serverless?

This section of the research is going to cover a discussion about whether each one of the solutions can be considered serverless according to the definition proposed by Van Eyk et al. (2017):

- Pillar 1: High availability and scalability provided by CSP with almost no operational management needed by the client
- Pillar 2: Granular billing based on the utilisation of resources
- Pillar 3: Services are event-driven

For that reason, the topics being checked are high-availability, scalability, billing and triggering events for each one of the studied subjects. In matters of high-availability, the study is only taking into account the high-availability of running containers, as the infrastructure is already of the responsibility of the CSP.

4.3.1 AWS ECS Fargate

In matters of High-Availability, AWS by default will dispose of faulty containers and launch new ones. They also allow customers to set custom health-checks so containers can be periodically checked and replaced if needed.

About scalability, unless a load balancer is configured to work together with ECS Fargate, only a single container instance will be made available. If a load balancer is set up, then the service can be configured to scale in or out depending on CPU or Memory metrics, or based on custom events that can be configured by customers (AWS 2020c). Unfortunately, at the moment this study was written, AWS does not allow applications to be down-scaled to 0 containers.

This gives power to customers to configure auto-scaling according to their application, however, it also demands a deep understanding of the application to get efficient scalability.

Considering the findings, it is possible to consider that AWS ECS Fargate meets the criteria defined in Pillar 1, but only when an extra component, the Load Balancer, is added to the solution.

In relation to billing, AWS charges per second (with a minimum of 1 minute) for CPU and Memory utilised, including the time needed to pull the container image up to the point the task is terminated. Therefore, the criteria defined in Pillar 2 is also met.

When it comes to triggering events, if a container is deployed in a task and exposed by a service, unless manually stopped, a task will continue running indefinitely. AWS, however, provides ways of start, run or stop the execution of the tasks based on a scheduled event (cron-like) or based on events triggered by other AWS resources, (AWS 2020b). Finally, In relation to Pillar 3, it is possible to consider that AWS ECS Fargate also meets the criteria depending on how the application is utilised.

4.3.2 GCP Cloud Run

For Pillar 1 (high-availability), GCP Cloud Run is the only solution among the three studied serverless CaaS that provides by default the possibility of down-scaling to zero when not in use and also offers the possibility to scale as much as needed depending on the demand of traffic accessing the application.

In relation to Pillar 2, GCP bills based on the number of requests and also on the time that resources (CPU and memory) were utilised.

In regards to Pillar 3, by default GCP Cloud Run exposes an HTTPS domain that can be utilised to trigger and interact with the containerised application. Alongside the domain, the containerised application can also be triggered based on a list of more than 60 events available from GCP as explained in *Cloud Run: Container to production in seconds | Google Cloud* (2020).

4.3.3 Azure Container Instances

Even though Azure CI guarantees high availability by disposing and replacing faulty containers automatically, is the only solution that at the moment does not offer a way to scale depending on demand, nor to define the number of instances to run. On Container Instances page, Microsoft even recommends the usage of AKS (Azure Kubernetes Service) if an application demands auto-scaling (*Azure Container Instances documentation - serverless containers, on demand* 2020). In relation to Pillar 1, at the moment this study was done, Azure CI as a standalone serverless CaaS partially meets the criteria, unless combined with AKS.

For Pillar 2, Azure CI charges for the resources (CPU and memory) utilised based on the amount of time utilised by the application.

Azure CI generates a fully qualified domain name that can be used to interact with the containerised application. Azure also offers ways to schedule the execution of containers as well as options to trigger based on events from other Azure services (ibid.).

4.3.4 Conclusion

When considering the pillars defined by Van Eyk et al. (2017), which characterises an application as Serverless, it is possible to conclude that GCP Cloud Run is the only serverless CaaS that meets all criteria as a standalone solution.

AWS ECS Fargate provides a way of automatically scaling an application based on traffic, but only when combined with another AWS resource (Application Load Balancer) that needs to be in constant use.

Azure Container Instances still does not offer a way of automatically scaling serverless containers, unless CI is used as part of another Azure service (AKS).

Nevertheless, the three studied subjects brand themselves as serverless solutions and, undoubtedly, allow their customers to run containerised applications without the need of manage or configure any part of the required underlying infrastructure. They also offer high-available solutions with self-healing and billing models that only charge based on the usage.

As these solutions are relatively new, when compared to FaaS, it is also expected that they get better with time offering more flexibility, possible configurations and even lighter chargers.

4.4 Which solution has the best performance?

The results and findings for the application performance data that was collected, following the directives described in Section 3.4, are going to be presented. It will cover the three studied CSPs serverless CaaS solutions using the four possible combinations of configurations identified to be present in all of them, as in Table 10.

4.4.1 Baseline for comparison

Before start comparing the CSPs, it was decided to run the exact same tests in a local environment using Docker. As the Docker CLI allows us to limit CPU and memory allocated for a containerised application, this research considers that results obtained for local executions should follow similar trends when compared with containerised applications running in the cloud. Therefore, having these local Docker executions will provide a baseline for the comparisons. Data was extracted from over 208 tests executed in a local environment running Docker. Details about this baseline such as minimum, average and maximum application execution times for the baseline can be found in Table 11.

Combination	Type	Min (ms)	Avg (ms)	Max (ms)
A	Factorial of 10	0.0048	0.0281	0.5882

A	Factorial of 1000	0.3803	0.5626	1.2603
A	Factorial of 10000	38.0433	43.4379	66.9341
A	Factorial of 32000	459.0015	540.6431	936.6241
A	Factorial of 43000	864.2752	1061.3826	1851.6071
A	Factorial of 50000	1203.205	1601.3173	2417.2188
A	Factorial of 60000	2245.9392	2705.1585	3271.6621
A	Hello World	0.0005	0.0086	0.0193
B	Factorial of 10	0.0088	0.018	0.057
B	Factorial of 1000	0.3719	0.5359	0.9241
B	Factorial of 10000	39.3248	41.8591	46.793
B	Factorial of 32000	482.6093	537.4373	711.3054
B	Factorial of 43000	901.4103	1050.4259	1433.524
B	Factorial of 50000	1241.3037	1548.6138	1977.8218
B	Factorial of 60000	2269.1555	2712.317	3552.4744
B	Hello World	0.0067	0.0106	0.0176
C	Factorial of 10	0.0069	0.0168	0.0508
C	Factorial of 1000	0.3893	0.553	0.8979
C	Factorial of 10000	39.4616	41.6895	47.3628
C	Factorial of 32000	478.236	536.5177	774.9054
C	Factorial of 43000	872.4742	1025.7688	1291.4561
C	Factorial of 50000	1266.8352	1562.7032	2567.4065
C	Factorial of 60000	2350.2976	2681.7906	3281.5498
C	Hello World	0.0072	0.0122	0.1264
D	Factorial of 10	0.0069	0.0179	0.0377
D	Factorial of 1000	0.3731	0.5911	0.9615
D	Factorial of 10000	39.6836	41.6221	51.5628
D	Factorial of 32000	478.3809	522.7921	585.8946
D	Factorial of 43000	904.5238	1033.3186	1234.9231
D	Factorial of 50000	1236.1243	1451.7842	1609.5278
D	Factorial of 60000	2293.5076	2606.8713	3179.7356
D	Hello World	0.0069	0.0114	0.0429

Table 11: Minimum, average and maximum application execution times in milliseconds (ms) for local execution utilising Docker to serve as baseline for further comparisons against serverless CaaS solutions.

4.4.2 The Hello World endpoint performance - From the application point of view

As expected in the first hypothesis defined in Section 3.2.2, the Hello World endpoint has a very similar performance in all tested serverless CaaS, with an average execution time of less than 0.02 milliseconds in all tested combinations of CPU and Memory, as shown in Figures 11, 12, 13 and 14.

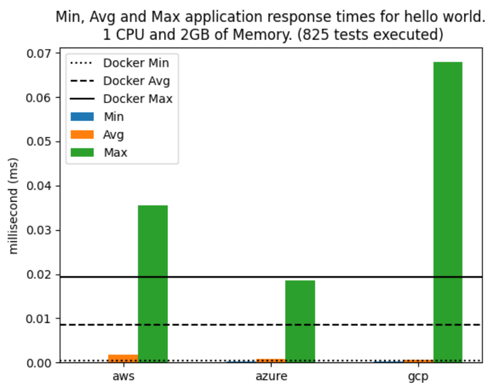


Figure 11: Hello World endpoint performance for 1 CPU and 2GB of Memory, compared with the baseline from local executions using Docker

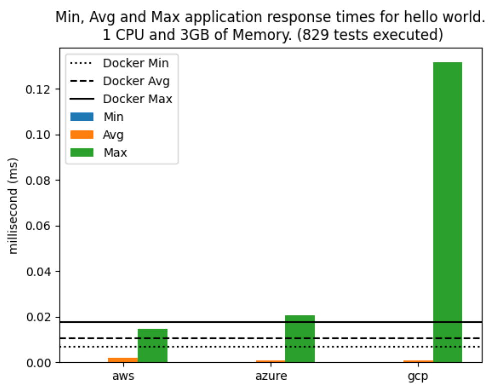


Figure 12: Hello World endpoint performance for 1 CPU and 3GB of Memory, compared with the baseline from local executions using Docker

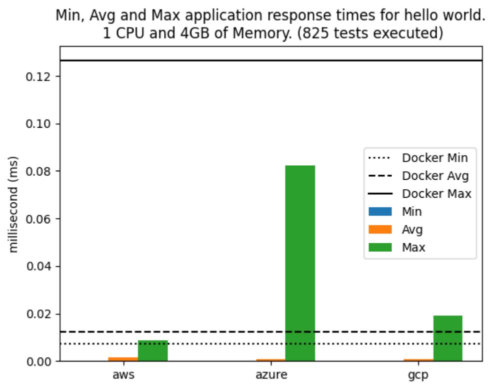


Figure 13: Hello World endpoint performance for 1 CPU and 4GB of Memory, compared with the baseline from local executions using Docker

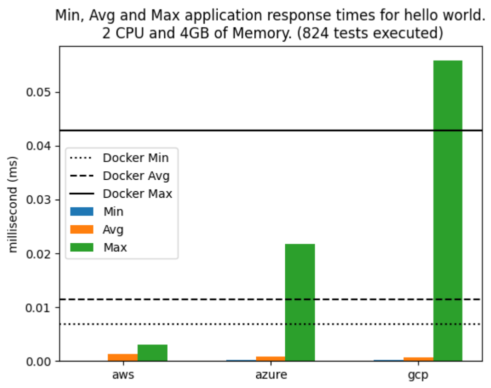


Figure 14: Hello World endpoint performance for 2 CPU and 4GB of Memory, compared with the baseline from local executions using Docker

The study found that the average execution time in all CSPs was always below the average time calculated in a local machine (the dashed line in the figures).

Another interesting point observed is that when measuring the serverless CaaS that had the longest execution time, GCP Cloud Run was the worst in 3 out of the 4 combinations. It is shown in Figure 12, that the maximum time it took in one of the tests, was at least 12 times slower than an average response time. However, when considering only the average times it is possible to see, in Figure 15 plotted in Grafana, that GCP Cloud Run had the best average performance among the three tested subjects, followed by Azure CI and AWS ECS Fargate, which was at least 2 times slower than GCP in all combinations)

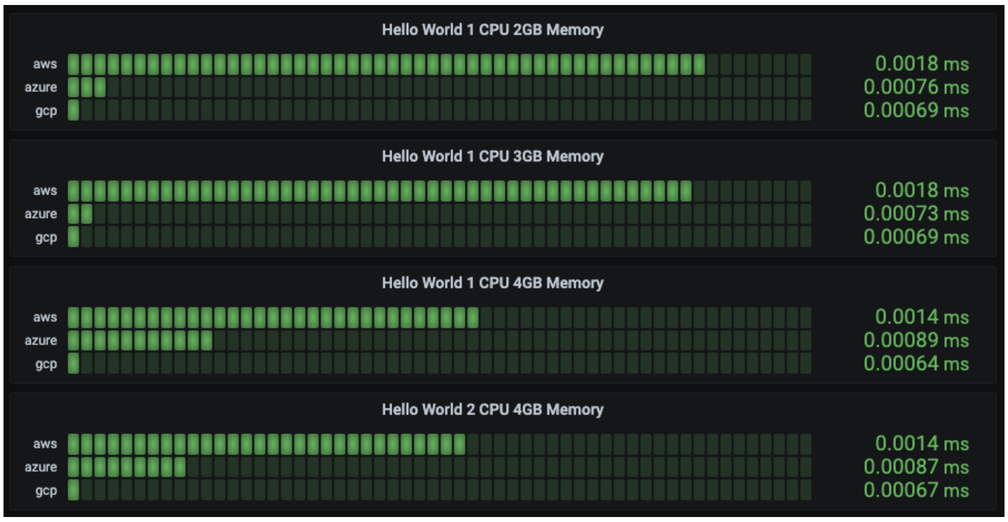


Figure 15: Average performance of Hello World endpoint in all possible combinations.

Finally, the last conclusion the study can take is that the performance of the Hello World endpoint was not substantially impacted directly by the different combinations of CPU and Memory.

4.4.3 The High CPU (factorial) endpoint performance - From the application point of view

In this section, the study is going to present the findings after the analysis and comparison of the collected data for the endpoint that demands a high CPU utilisation, taking into consideration the performance from the application point of view as defined in Section 3.4.1.

After analysing the data, the study concluded that the collected data for the calculation of factorial of 10 and factorial of 1000 was, at least at first, unreliable to make clear statements. No visible trends were spotted when compared with the rest of the factorials in other combinations. For the other factorials, similar trends were identified and the study is going to explore them in this section.

AWS ECS Fargate was the only serverless CaaS solution that kept its average response time below the average response time from the local execution baseline (using Docker) in all test cases and in all combinations of CPU and Memory, as exemplified in Figure 16, which shows application execution times for combination A (Table 10). Plots with all results for the other three combinations (B, C and D) can be found in Appendices H, I, J

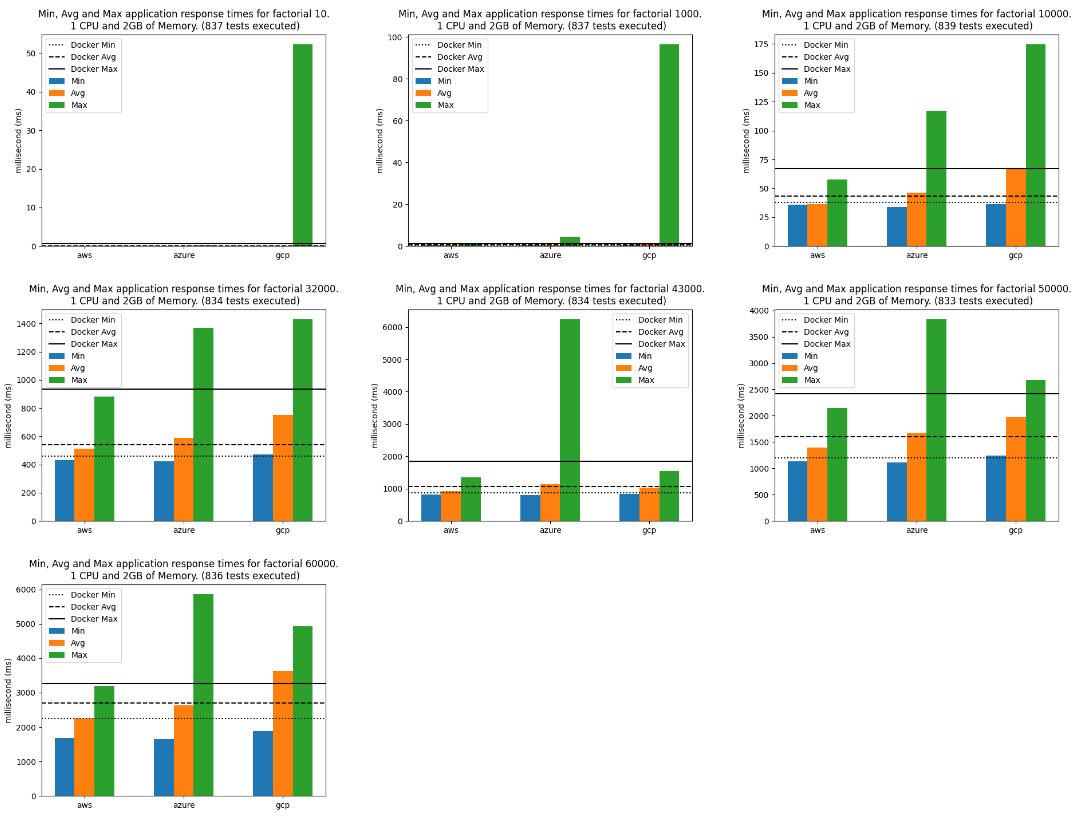


Figure 16: Minimum, average and maximum values when calculating factorials for 1 CPU with 2GB of Memory in each CSP.

Figures 17, 18, 19 and 20 were plotted for the comparison of the average times of the factorial calculation for the number that demands more CPU processing (60000). It is possible to observe that GCP Cloud Run was the only solution that had its execution times above the Docker baseline, and this trend was mostly present when calculating other factorials. AWS ECS Fargate, on the other hand, had the best performance with the fastest average application execution time, close followed by Azure CI. The metrics collected show us that when calculating the factorial of 60000, AWS ECS Fargate is at least 38% faster than GCP Cloud Run and at least 12% faster than Azure CI. Azure CI itself is at least 27% faster than GCP Cloud Run.

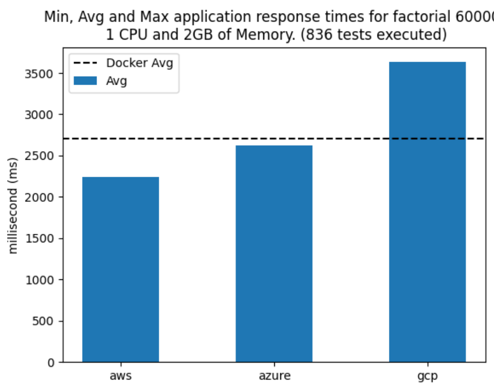


Figure 17: Average application execution times for of High CPU end-point calculating factorial of 60000, when configured for combination A (1 CPU and 2GB of Memory) compared with the baseline from local executions using Docker.

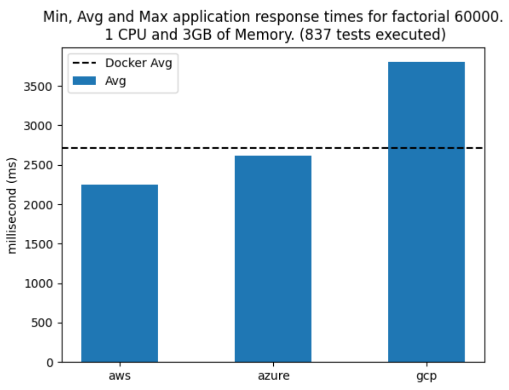


Figure 18: Average application execution times for of High CPU end-point calculating factorial of 60000, when configured for combination B (1 CPU and 3GB of Memory) compared with the baseline from local executions using Docker.

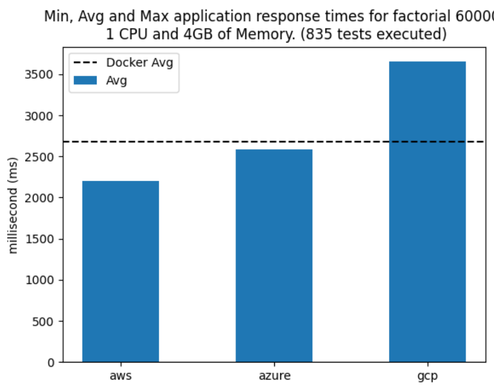


Figure 19: Average application execution times for of High CPU end-point calculating factorial of 60000, when configured for combination C (1 CPU and 4GB of Memory) compared with the baseline from local executions using Docker.

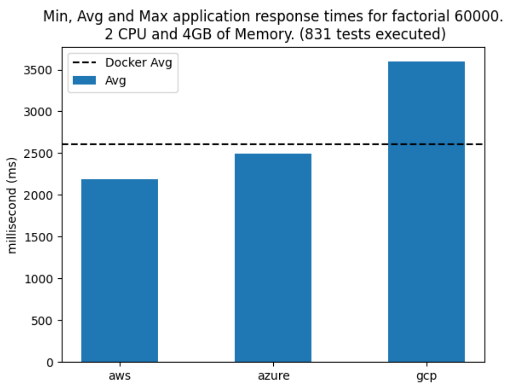


Figure 20: Average application execution times for of High CPU end-point calculating factorial of 60000, when configured for combination D (2 CPU and 4GB of Memory) compared with the baseline from local executions using Docker.

In 4 out of 5 reliable scenarios tested (factorial of 10000, 32000, 43000, 50000 and 60000) the study was able to confirm that AWS ECS Fargate had the best performance, followed by Azure Container Instances with GCP Cloud Run being the worst. However, it is worth to point out that the result from the calculation of factorial of 43,000 was the only scenario that did not follow the same performance trend. This different trend can be observed in Figures 21, 22, 23 and 24. In these figures AWS ECS Fargate had still the best performance, however, GCP Cloud Run performed better than Azure CI. Azure CI was also the only one that had its average execution time above the average baseline from executions performed in a local environment running Docker. This was also the only scenario that had GCP Cloud Run with an average application execution time below the baseline.

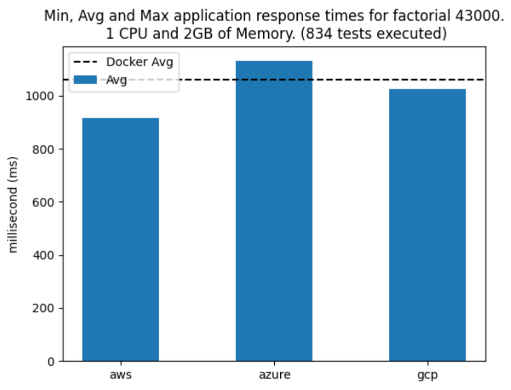


Figure 21: Average application execution times for of High CPU end-point calculating factorial of 43000, when configured for combination A (1 CPU and 2GB of Memory) compared with the baseline from local executions using Docker.

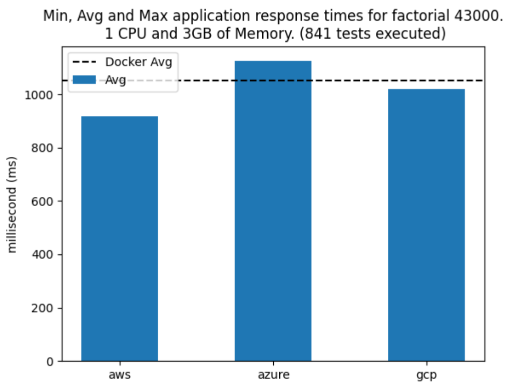


Figure 22: Average application execution times for of High CPU end-point calculating factorial of 43000, when configured for combination B (1 CPU and 3GB of Memory) compared with the baseline from local executions using Docker.

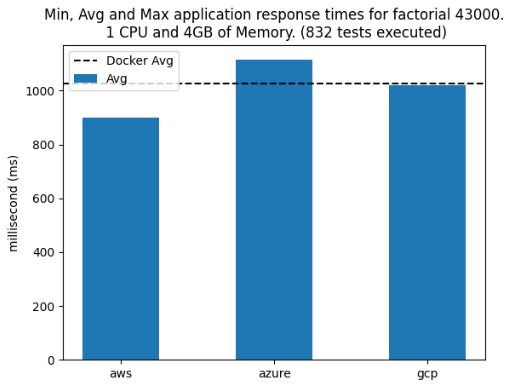


Figure 23: Average application execution times for of High CPU end-point calculating factorial of 43000, when configured for combination C (1 CPU and 4GB of Memory) compared with the baseline from local executions using Docker.

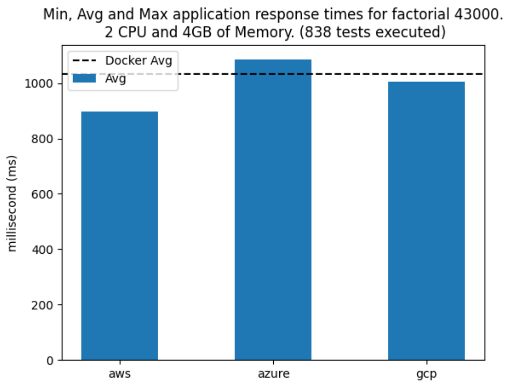


Figure 24: Average application execution times for of High CPU end-point calculating factorial of 43000, when configured for combination D (2 CPU and 4GB of Memory) compared with the baseline from local executions using Docker.

When taking into account the minimum, average and maximum execution times, AWS had the most reliable results, showing less variation between these 3 measures. The ratios between maximum and minimum, maximum and average and average and minimum times can be found in Appendix N.

Table 12 shows that AWS ECS Fargate not only has the smallest numbers for minimum and maximum variation rates between the maximum and average application execution times but also has the smallest difference between these minimum and maximum variances.

When comparing the ratio between the average execution time and minimum execution time, in Table 13 it is possible to confirm that AWS ECS Fargate was again the one showing the smallest values for minimum and maximum ratios, however, when comparing the difference between these ratios, for this scenario, Azure CI showed less variance closely followed by AWS Fargate.

The worst maximum divided by minimum ratio happened in Azure CI when calculating the factorial of 43000 using combination A (1 CPU with 2GB of memory). In this case, the maximum application execution time was 7.78 times slower than

CaaS	min(max AET /avg AET) ratio	max(max AET/avg AET) ratio	Difference
AWS ECS Fargate	1.11	1.79	0.68
Azure CI	1.71	5.51	3.8
GCP Cloud Run	1.30	4.01	2.71

Table 12: Minimum, maximum and difference between ratios when comparing ratio from maximum Application Execution Time (AET) divided by average AET. Cells highlighted in grey represent the smallest value per column.

CaaS	min(avg AET /min AET) ratio	max(avg AET/min AET) ratio	Difference
AWS ECS Fargate	1.01	1.33	0.32
Azure CI	1.36	1.65	0.29
GCP Cloud Run	1.15	2.17	1.02

Table 13: Minimum, maximum and difference between ratios when comparing ratio from average Application Execution Time (AET) divided by minimum AET. Cells highlighted in grey represent the smallest value per column.

the minimum application execution time. This maximum application execution time was also 5.51 times slower than the average.

The best maximum divided by minimum ratio happened in AWS ECS Fargate when calculating the factorial of 10000 using combination D (2 CPUs with 4GB of memory). The maximum application execution time was 1.25 times slower than the minimum and 1.23 times slower than the average.

Interestingly, from each studied serverless CaaS, the values from application execution times with less variance between maximum and average, and between average and minimum were all from tests running in combination D. The ones with worst variance per CaaS were not from a unique combination, however, in 66% of the times they were using combination C (1 CPU and 4GB of memory).

4.4.4 The internal CaaS processing time

To measure the internal processing time, the same metrics collected from the performance tests were utilised, however, taking only in consideration the times for the hello world endpoint. Given the fact that the Hello World endpoint is executed very fast (less than 0.02 ms), this will allow us to estimate how much of the processing time happens outside of the application.

Serverless CaaS	Combination	AET (ms)	TTFB (ms)	CaaS Processing Time (ms)	Samples
Docker Baseline	1cpu/4mem	0.0122	3.98	3.96	6
Docker Baseline	2cpu/4mem	0.0114	4.53	4.52	6
Docker Baseline	1cpu/3mem	0.0106	4.57	4.56	6
Docker Baseline	1cpu/2mem	0.0086	4.92	4.91	8
Azure CI	1cpu/4mem	0.0009	93.63	93.63	253
Azure CI	2cpu/4mem	0.0009	95.27	95.27	252
Azure CI	1cpu/2mem	0.0008	95.79	95.79	255
Azure CI	1cpu/3mem	0.0007	98.47	98.47	258
AWS ECS Fargate	1cpu/3mem	0.0018	128.14	128.14	275
AWS ECS Fargate	1cpu/4mem	0.0014	128.66	128.65	276
AWS ECS Fargate	1cpu/2mem	0.0018	128.85	128.85	275
AWS ECS Fargate	2cpu/4mem	0.0014	131.19	131.19	271
GCP Cloud Run	2cpu/4mem	0.0007	161.64	161.64	301
GCP Cloud Run	1cpu/4mem	0.0006	165.61	165.61	296
GCP Cloud Run	1cpu/3mem	0.0007	165.73	165.73	296
GCP Cloud Run	1cpu/2mem	0.0007	194.65	194.65	295

Table 14: Internal CaaS processing time collected from the execution of tests against Hello World endpoint. Average of Application Execution Time (AET) and the average of Time to First Byte (TTFB or Average HTTP Request Waiting Time) were used based on the number of informed samples. The CaaS Processing Time is calculated based on the difference between TTFB and AET, as explained in Figure 6. The table is sorted based on fastest CaaS Processing Time.

The data for this comparison was extracted and organised in Table 14 and local executions (using Docker) were also added as the baseline for the comparison.

Table 14 is sorted based on the CaaS Internal Processing Time, with fastest times on the top. The first conclusion to be made is that Azure CI has the fastest time, with internal processing times lower than 100ms, followed by AWS ECS Fargate with processing times up to 131ms. GCP Cloud Run was the worst, with processing times up to 195ms, which is almost twice the time of Azure CI.

Another interesting aspect to observed is that with the exception of AWS ECS Fargate, it appears that all other studied subjects, including the local Docker baseline, had faster internal processing times when more resources (CPU and memory) were available.

It is also possible to conclude that in all tested cases (all Factorials and Hello World) in all possible combinations (Table 10), Azure CI was the one with the fastest internal processing times. AWS ECS Fargate was in the second place in

56.25% of the scenarios, close followed by GCP Cloud Run which was faster than Fargate only in 43.75% of the scenarios.

A table with all the comparisons, including the internal processing times for the factorial endpoint can be found in Appendix Q.

All the results considered, the study found that Azure CI was the best performing in this category, followed by AWS ECS Fargate and GCP Cloud Run.

4.4.5 Differences between weekends and weekdays

Considering that the automation developed was executed during all days of the week, a comparison was made to check whether there were differences of performance between weekdays (Monday to Friday inclusive) and weekends (Saturday and Sundays). The initial hypothesis was that weekends executions could be faster, as there are businesses that only operate during weekdays, therefore more resources would be available in the CSPs. Comparison between the average application response times and the average internal CaaS processing time were made for each of one the test scenarios the study is analysing in every combination of CPU and Memory (Table 10).

Tables with more details about each comparison between weekends and weekdays can be found in Appendices P and O.

When comparing the application execution time, it is possible to observe that weekends were 54.2% of the times faster than weekdays as in Figure 25. However, when comparing the internal CaaS processing time, it is noticeable that the initial hypothesis was wrong as 84.4% of the cases were slower during the weekends as shown in Figure 26.

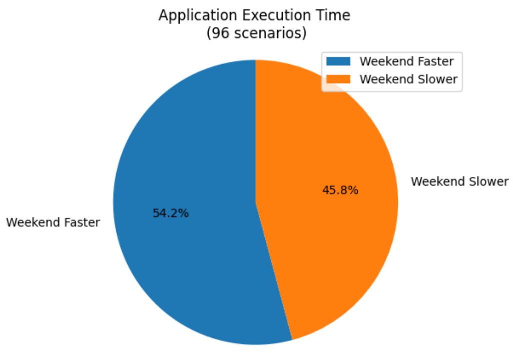


Figure 25: Comparison showing that 54.2% of tests and combinations (96 in total between Hello World and Factorial), were faster in weekends when compared to same same tests and combinations in weekdays.

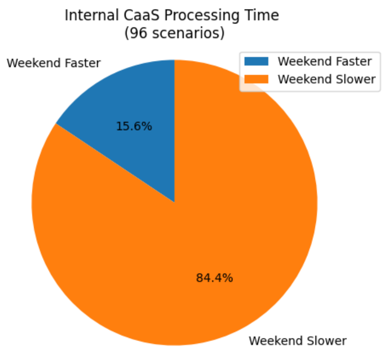


Figure 26: Comparison showing that majority (84.4%) of tests and combinations (96 in total between Hello World and Factorial) were slower in weekends when compared to same same tests and combinations in weekdays.

The results presented in Table 15 are for the comparison between the internal CaaS processing times. It has only the results from tests performed against the Hello World endpoint, which runs fast and gives us a more accurate internal processing time. The internal processing time is calculated based on the difference between the average TTFB (Time To First Byte or request waiting time) and the average application processing times. For this particular comparison, all the tests in all configurations showed slower internal CaaS processing times during weekends. It is important to highlight that, in some cases, the difference was almost zero, with less than 1 millisecond.

CaaS	**Combination**	**Difference (ms)**	**Weekend**	**Samples**
AWS ECS Fargate	A	4.13	Slower	275
AWS ECS Fargate	B	3.08	Slower	275
AWS ECS Fargate	C	0.84	Slower	276
AWS ECS Fargate	D	3.89	Slower	271
Azure CI	A	0.72	Slower	255
Azure CI	B	2.72	Slower	258
Azure CI	C	2.20	Slower	253

Azure CI	D	2.58	Slower	252
GCP Cloud Run	A	3.40	Slower	295
GCP Cloud Run	B	23.21	Slower	296
GCP Cloud Run	C	11.09	Slower	296
GCP Cloud Run	D	12.21	Slower	301

Table 15: Comparing Internal CaaS Processing Times (ICPT) from executions against Hello World endpoint, showing whether weekend executions were faster than weekdays executions. The difference column was calculated based on subtraction of average weekends ICPT from average weekdays ICPT. A negative difference means that weekend had faster and the positive difference means that weekdays had faster average ICPT.

4.4.6 Conclusion

No single serverless CaaS solution had the best performance in every single investigated aspect.

Among the studied subjects, AWS has the serverless CaaS solution with the best performance from an application point of view and for an application that demands a high usage of CPU. When considering the time necessary to process a task that demands CPU processing, AWS ECS Fargate had the fastest response times among the three studied subjects, for all the reliable Factorial test (ignoring 10 and 1000) in every tested combination of CPU and Memory.

When comparing average response times, Azure Container Instances was almost as good as AWS ECS Fargate. However, when taking into consideration worst-case scenarios, Azure CI had one of the worst results by presenting a maximum application execution time 5.51 times slower than an average execution, in the results of one of the test cases.

When comparing an almost empty application, using the Hello World endpoint, GCP had the worst maximum application execution times. However, when considering the average execution times, GCP had the best performance among the three, sometimes being at least twice faster as AWS ECS Fargate.

By calculating the internal processing time for the studied serverless CaaS solutions, the study was able to identify that at the moment, Azure CI is the one with the best performance in this aspect, being faster than the other two serverless CaaS in 100% of the tested cases in all tested combinations of CPU and memory.

Finally, the study found that for an almost empty application, such as the Hello World endpoint, GCP Cloud Run can be considered as the best performing serverless Container as a Service solution, followed by Azure CI and AWS ECS Fargate, considering that worst-case scenarios are acceptable and not so frequent. For an application that demands a high CPU utilisation, AWS ECS Fargate was the best performing serverless CaaS solution, close followed by Azure Container Instances, leaving the third place to GCP Cloud Run.

4.5 Which solution is the cheapest?

In this section, the research is going to determine which solution is the cheapest following the plans defined in the research question in section 3.2.3.

As the billing of the products depends on the region where the resources are deployed, for this study, all containerised applications will be deployed in regions located in Virginia, in the United States of America. The name of the regions vary depending on the provider and they are respectively: us-east-1, East US and us-east-4 for AWS, Azure and GCP.

It is important to highlight that the prices displayed in this section were the prices available when the research was developed and that the currency utilised in this study is the United States Dollar (symbol $).

4.5.1 Price to run AWS ECS Fargate

AWS charges for containers from the moment it starts pulling a container image from the image registry until the task running containers is terminated. The charges happen per second however with a minimum of 1 minute of utilisation (AWS 2020a). The pricing for running serverless containers in AWS Fargate is in Table 16.

Resource	**Price**
vCPU	$0.0000112444/second
Memory (GB)	$0.0000012347/second

Table 16: On-demand prices per resource and utilisation time for resources that may be needed when running AWS ECS Fargate serverless containers in us-east-4 region.

Besides the pricing on demand, AWS also offers some alternatives that could be cheaper depending on how a given application run.

A "Spot Pricing" offers the same amount of resources at a price per second 2.6 times cheaper. In this setup, AWS hosts a containerised application using resources that would be otherwise idle, but with the downside that AWS could interrupt and stop the application if resources are required by AWS.

The other alternative is using a "Compute Savings Plan". In this plan, the price could be up to 50% cheaper as long as the customer knows that a certain amount of computing usage (in dollars per hour) will be used in a long term. For that, a contract from one up to three years needs to be purchased.

Besides CPU and memory, AWS also charges you for other resources that may be required for an application to run, such as network traffic, a dedicated load balancer or even storing logs in CloudWatch (another AWS service).

4.5.2 Price to run Azure Container Instances

Azure Container Instances are billed per second for the number of vCPU and GBs of memory allocated for a container group of one or more containers (Microsoft 2020). If the container group uses Windows containers, an extra charge is applied (also per second) for the Windows software. All the values can be found in Table 17. The billing starts from the moment the container image is being downloaded until it is stopped.

A container group can have up to 4 vCPUs and each CPU can be combined with up to 7 GBs of memory.

Resource	Price
vCPU	$0.0000113/second
Memory (GB)	$0.0000013/second
Windows Container	$0.0000120/second
Dynamic IP Address	$0.0040000/hour
Static IP Address	$0.0036000/hour

Table 17: Prices per resource and utilisation time for resources that may be needed when running Azure Container Instances in the East US region.

During the study, a preview of pricing for GPU resources inside containers was also available. However, as it has not been officially released, this information was not included in the pricing table.

4.5.3 Price to run GCP Cloud Run

Differently than Azure Container Instances and AWS ECS Fargate, GCP Cloud Run charges for the each, rounded up, 100 milliseconds instead of seconds. The values can be found in Table 18 and all pricing information in this section was extracted from Google (2020).

Resource	Price
vCPU	$0.0000240/second
Idle vCPU	$0.0000025/second
Memory (GB)	$0.0000025/second
Idle Memory (GB)	$0.0000025/second
Requests	$0.4000000/million

Table 18: Prices per resource and utilisation time/frequency for resources that may be needed when running GCP Cloud Run in us-east4 region. These prices ignore the monthly free tier available.

The billing for idle resources happens when the resources are not being utilised and only when the minimum number of containers running is set to be greater than zero. This charge is to keep the container warm.

GCP has an interesting free tier that is renewed every month. Every month, the utilisation of the resources listed in Table 19 is free of charge.

Resource	**Free Tier**
vCPU	180,000 seconds/month
Memory (GB)	360,000 seconds/month
Requests	2 million

Table 19: Monthly free tier provided by GCP in relation to Cloud Run.

There are some differences in GCP when compared to the other studied subjects as, at the moment, it is the only one capable of down-scaling to zero containers.

In relation to billing, if an application is set to downscale to zero, GCP only charges for the duration of requests, not billing for the time nor for the resources utilised when pulling the container from the registry and destroying the application afterwards. Also, if an application is capable of handling multiple requests simultaneously, the billable time (for CPU and Memory) are not duplicated if they are within the same time frame. A visual explanation can be found in Figure 27, which was based on the information provided by Google (2020).

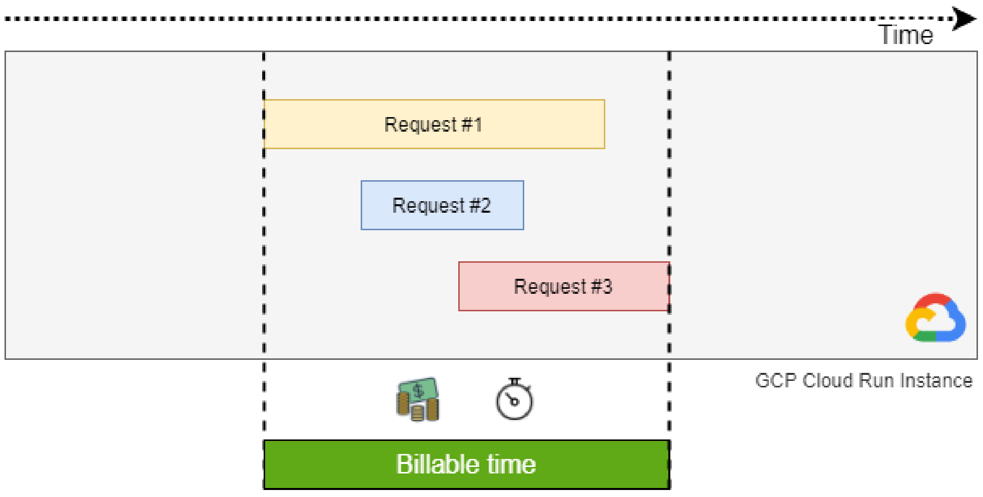


Figure 27: Billable time for GCP Container Run when minimum number of running instances is zero (down-scaling to zero enabled).

4.5.4 How to define which solution is the cheapest?

To answer this question a few approaches can be considered, however, the scope of the analysis done in this study will be limited. In this study, any other possible charges that may incur to run the containerised application in each CSP will be ignored, therefore, charges such as the price for a public IP or charges in relation to data transferred are not going to be considered.

Given the fact that GCP Cloud Run is the only solution (at the time this research was written) capable of automatically down-scaling to zero instances, the study is also considering that there is no idle time and that the application is in constant use.

Having the scope limited, the study is only going to measure charges for CPU and memory. As the price is always relative to the number of CPU and memory allocated, the combination utilised does not have an impact on our plots. Therefore, all the examples and plots in this section are going to use the combination A (1 CPU and 2 GB of Memory) from Table 10.

The main goal of this section is to identify which solution is the cheapest when ignoring free tiers, when considering free tier and when considering the performance measured from the application endpoint that demands a high CPU utilisation. Finally, the real costs for running the performance tests which were executed in Section 4.4 will also be presented.

4.5.5 Which solution is the cheapest, when ignoring free tiers?

When just considering the price per second, Figure 28 was plotted considering a hypothetical application running uninterruptedly for 24 hours during 1 year. It is possible to observe that AWS is the cheapest, costing \$432.40, close followed by Azure, which would cost \$438.35. On the other hand, it is clear that GCP is the most expensive one, being slightly over twice more expensive than AWS or Azure, with a price of \$914.54.

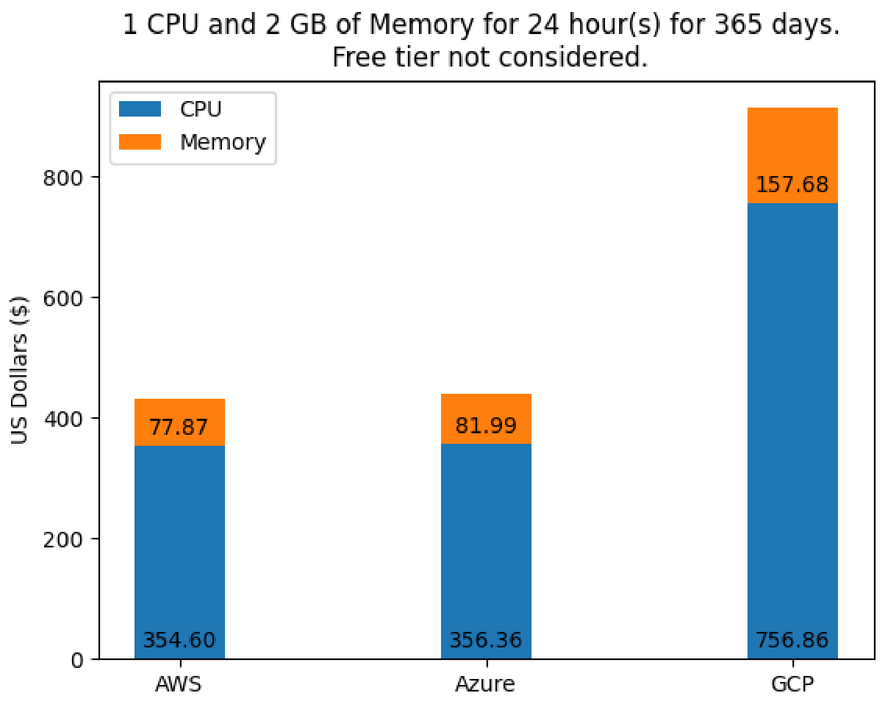


Figure 28: Pricing of running one container with 1 CPU and 2 GB of Memory for 24 hours a day during 1 year without considering possible free tiers. AWS: \$432.47, Azure: \$438.35 and GCP: \$914.54.

4.5.6 Which solution is the cheapest when considering free tiers?

As shown in Table 19, GCP offers a generous monthly free tier of 50 hours per CPU and 100 hours per GB of Memory. In configuration A (1 CPU and 2 GB of Memory), this translates to 50 free hours of container execution per month. It means free \$4.77 credits per month or \$5.57 if the value for the 2 million free requests is also considered.

Therefore, it is possible to conclude that if the containerised application uses 50 hours or less per month, GCP is clearly the cheapest option. To simulate these 50 hours, let us consider a hypothetical scenario where an application needs to run for 2.5 hours a day during working days in a month that has 20 working days. The price for running this application is plotted in Figure 29. Even if this value is not considered to be expressively high, in the long term it could become significantly important.

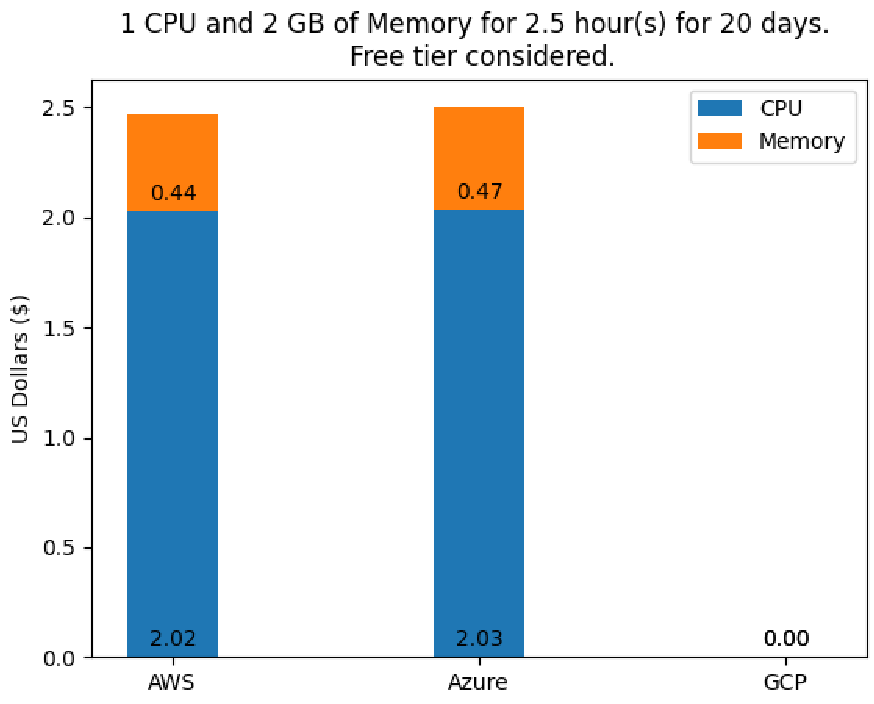


Figure 29: Pricing of running one container with 1 CPU and 2 GB of Memory for 2.5 hours a day during 1 year.

Having defined that GCP Cloud Run is the best option for an application running for 50 hours or less, the next study goal was to identify if there was a point where all CSPs would have almost the same price when the free tier is considered. By plotting the prices over execution time in a month, the study found that an application running for around 96 hours costs almost the same (around $4.80) in all CSPs, as shown in Figure 30. Therefore, another conclusion that can be made is that when considering the free tier, GCP Cloud Run is still the cheapest option for a containerised application if it runs for up to 96 hours per month. Beyond 96 hours per month, AWS ECS Fargate becomes the cheapest serverless CaaS solution.

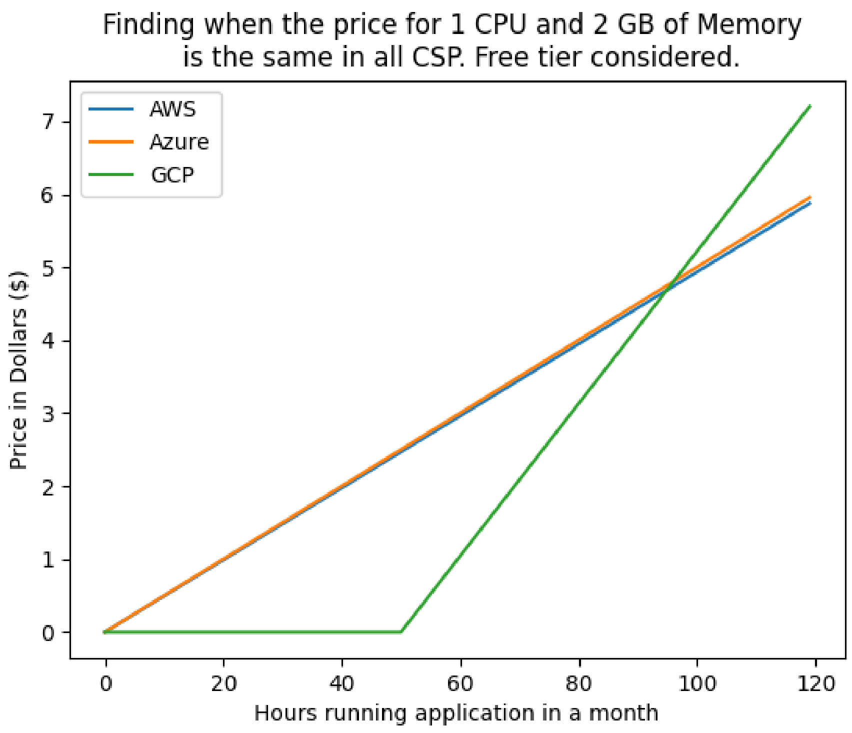


Figure 30: Finding when the price for containers with 1 CPU and 2 GB of Memory is the same in all CSP, considering free tier. Around 96 hours of usage is when the 3 providers have similar prices.

4.5.7 Which solution is cheapest when running an application that demands CPU?

To find the answer to this question, the average application processing time data previously calculated was considered when calculating the factorial of the number 60000. They are 2.244 seconds for AWS, 2.628 for Azure and 3.632 for GCP.

With this information, it was possible to plot how much would it cost to execute the same amount of requests in each one of the CSPs. This calculation is illustrated in Figure 31.

When ignoring the free tier, GCP is not only the most expensive (per second) but also the slowest to process an application that demands a high CPU utilisation. Which makes it the most expensive among the studied CSPs. AWS and Azure present similar prices and execution times, however, the gap between them is more

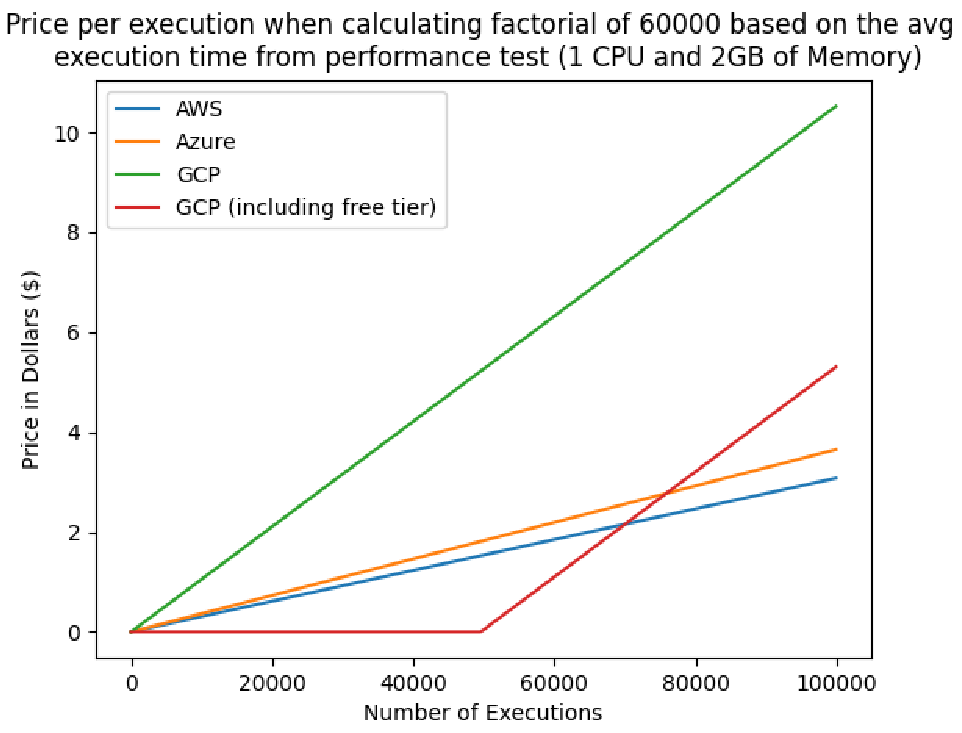


Figure 31: Price per execution when calculating factorial of 60000 based on the average execution time collected in the Performance Tests, using combination A (1 CPU and 2GB of Memory).

noticeable as the number of executions grow, given the fact that AWS is not only cheaper than Azure but also slightly faster.

Nevertheless, despite being the slowest, when GCP free tier is taken into consideration, GCP Cloud Run could still be defined as the cheapest, and even free, depending on the number of executions.

4.5.8 Which solution was the cheapest when running the performance tests?

The results displayed in this section reflect how much it actually cost, per CSP, to run all performance tests described in Section 4.4. The Free tier available is being considered in GCP and values for taxes are also included. The final values for running over 26,000 performance test cases between the three CSPs can be found in Table 20.

Cloud Service Provider	Total Billing Price
AWS	$6.81
Azure	$6.85
GCP	$1.11

Table 20: Real costs in US Dollars to run the performance tests described in Section 4.4.

For all the tests executed in this whole study, the cheapest billing was from GCP. However, that is the case especially because the free tier limits were not exceeded (Table 19). Azure was the most expensive, closely followed by AWS. The detailed billing for each CSP can be found in Appendices B, A, D.

4.5.9 Conclusion

Based on the study performed in this chapter, the main conclusion to be made is that knowing the performance of the containerised application, how many times and for how long the application needs to run is crucial when deciding which CaaS serverless solution is the best to run and host a given application.

When comparing only the prices, completely ignoring free tiers, AWS ECS Fargate is clearly the best choice closely followed by Azure Container Instances. Fargate was not only the fastest when running the High CPU endpoint, but also has the cheapest billing per CPU and memory per second among the three studied subjects.

AWS ECS Fargate also offers cheaper plans if clients commit to run an application for years or if they accept the risks of running applications using "spot instances". Considering that an application will always be running with constant demand, when pricing is the most important aspect the study recommendation for serverless CaaS, from best to worst, is AWS ECS Fargate, Azure Container Instances and GCP Cloud Run.

As some applications may have idle times, AWS and Azure are in disadvantage when compared to GCP. GCP Cloud Run is the only serverless CaaS that, at the moment, provides automatic down-scaling to zero containers and also the only one that offers a cheaper price when the application is idle and is set to have at least one container running. GCP is also the only one that offers a generous free tier of 50 hours per month, considering a container with 1 CPU and 2GB of memory receiving up to 2 million requests.

When considering GCP free tier, GCP may be the best option for companies willing to try running their applications for free, or even for companies that demand less than 50 hours a month for an application that can run with 1 CPU and 2GB of memory.

The final answer for this topic is that the cheapest option depends on a deep understanding of the utilisation and on the performance of the application. AWS and GCP are the best, in their own ways and for different reasons. Azure Container Instances has a compatible price when compared to AWS, but it is not cheaper than AWS nor has a different approach as GCP does. Therefore, when considering only pricing and use cases, the study could not find reasons to recommend Azure CI over the other serverless CaaS.

5 Conclusions

This research compared existing serverless Container as a Service solutions that are available by the current major CSPs (Amazon AWS, Microsoft Azure and Google Cloud Platform).

The first conclusion obtained is that knowing in depth a containerised application and its demands is crucial to choose which serverless CaaS solution should be used to host the application. Only when this knowledge is present it is possible to achieve the best performance available at a cheaper price, given that no single CaaS can be defined as the best in every single studied aspect in this research.

Despite being the youngest of the solutions, GCP Cloud Run was the only CaaS that fully met, without the need of other cloud services, the three pillars defined by Van Eyk et al. (2017) that characterises a service as serverless: high-availability and scalability provided by CSP, granular billing and event-driven architecture.

For an almost empty application, all serverless CaaS performed well and could be recommended, however when considering only the average application execution times, GCP Cloud Run was the best performing solution.

For an application that demands a high CPU utilisation, the study found that AWS ECS Fargate was the best performing solution, with average times faster than the other two and always below the baseline in all tested scenarios. AWS ECS Fargate is the oldest solution and has also demonstrated to be mature and reliable, showing less variance between minimum, average and maximum execution times.

In matters of internal CaaS processing time, in the majority of the cases when more CPU and memory were available, the solutions had higher performance. Azure was the best performing solution among the three studied subjects, followed by AWS and GCP.

When comparing the performance from the application point of view, it is possible to conclude that weekends executions were faster in only 54.2% of the scenarios when compared to weekdays. On the other hand, when comparing the internal CaaS processing time, the study found that the majority (84.4%) of the scenarios were slower in the weekends.

In relation to billing, when taking into consideration price per second, AWS is the cheapest of the solutions, even offering cheaper prices if clients are willing to take some risks or committing for long term plans. However, GCP Cloud Run is the only solution that offers a monthly free tier that allows customers to run applications with 1 CPU and 2GB of memory for free for up to 50 hours a month. GCP is also the only solution, at the moment, capable down-scaling to zero containers when they are not in use.

When ignoring free tiers, considering price and application performance, the study found that for an application that demands high CPU utilisation, AWS ECS Fargate has the best cost/benefit ratio as it was not only the fastest but also the cheapest.

All things considered, the CaaS solutions are relatively new when compared to more consolidated approaches such as FaaS. Therefore, it is expected that these services get better, with more customisation and possible configurations, since the adoption and popularity of CaaS have only increased in recent years, demonstrating a great growth potential.

5.1 Limitations

The results presented in this research are limited to the metrics extracted during the execution of 26665 test cases distributed between 02/10/2020 and 26/10/2020. Considering that at any time CSPs could change how their managed infrastructure and services work, results obtained from this research may not be the same if the same tests are executed in the future.

Another limitation of this study is related to the internal CaaS processing time. As this information is not currently available from the CSPs, the method utilised in this research (subtracting application execution time from the time to the first byte) does not take into consideration the time needed by the operating system and web server running inside the container.

5.2 Future Works

Despite the efforts to cover as many comparisons related to serverless CaaS, some interesting topics were not covered in this research due to time limitations. Questions that might be answered in the future:

- Which serverless CaaS solution is easier to use?
- Do the usage of any of the serverless CaaS solutions imply on vendor lock-in?
- Which serverless CaaS solution scales better?
- Are there any cold starts?

The researches also encourage that future researches take advantage of the tools created in this project for automatic provision of serverless CaaS, execution of automated tests, data capturing and the containerised application itself. They are publicly available and can be found respectively in `https://github.com/`

viniciusbarros/msc-caas-comparison and https://hub.docker.com/r/viniciusbarros/ msc-performance-web-app.

References

AB, Load Impact (2020a). URL: https://k6.io/docs/ (visited on 07/10/2020).

– (2020b). URL: https://k6.io/docs/using-k6/metrics (visited on 07/10/2020).

AWS (2020a). *AWS Fargate Pricing.* URL: https://aws.amazon.com/fargate/pricing/ (visited on 24/10/2020).

– (2020b). *Scheduling Amazon ECS tasks - Amazon Elastic Container Service.* URL: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/scheduling_tasks.html (visited on 07/09/2020).

– (2020c). *Service Auto Scaling - Amazon Elastic Container Service.* URL: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-auto-scaling.html (visited on 19/09/2020).

AWS Fargate (2003). URL: https://aws.amazon.com/fargate/.

AWS VIRTUAL CONTAINERS DAY (2020). URL: pages.awscloud.com/virtual_containers_day_registration.html.

Azure Container Instances documentation - serverless containers, on demand (2020). URL: https://docs.microsoft.com/en-us/azure/container-instances/ (visited on 07/09/2020).

Azure updates (2020). URL: https://azure.microsoft.com/en-us/updates/?product=container-instances (visited on 07/09/2020).

Cloud Run (fully managed) release notes | Cloud Run Documentation (2020). URL: https://cloud.google.com/run/docs/release-notes (visited on 07/09/2020).

Cloud Run: Container to production in seconds | Google Cloud (2020). URL: https://cloud.google.com/run (visited on 07/09/2020).

Flexera (2020). *2020 State of the Cloud Survey from Flexera.* URL: https://info.flexera.com/SLO-CM-REPORT-State-of-the-Cloud-2020.

Goli, Alireza et al. (2020). 'Migrating from Monolithic to Serverless: A FinTech Case Study'. In: *Companion of the ACM/SPEC International Conference on Performance Engineering*, pp. 20–25.

Google (Oct. 2020). *Pricing Cloud Run Documentation Google Cloud.* URL: https://cloud.google.com/run/pricing (visited on 24/10/2020).

Grafana (2020). URL: https://grafana.com/oss/grafana/ (visited on 14/11/2020).

HashiCorp (2020). URL: https://www.terraform.io/ (visited on 07/10/2020).

Inc, Docker (2020). *python - Docker Hub*. URL: https://hub.docker.com/_/python (visited on 26/10/2020).

Introducing AWS Fargate (Nov. 2017). URL: https://aws.amazon.com/about-aws/whats-new/2017/11/introducing-aws-fargate-a-technology-to-run-containers-without-managing-infrastructure/.

Jambunathan, B. and K. Yoganathan (2018). 'Architecture Decision on using Microservices or Serverless Functions with Containers'. In: *2018 International Conference on Current Trends towards Converging Technologies (ICCTCT)*, pp. 1–7.

Lynn, T. et al. (2017). 'A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms'. In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 162–169.

Microsoft (2020). *Pricing - Container Instances: Microsoft Azure*. URL: https://azure.microsoft.com/en-us/pricing/details/container-instances/ (visited on 24/10/2020).

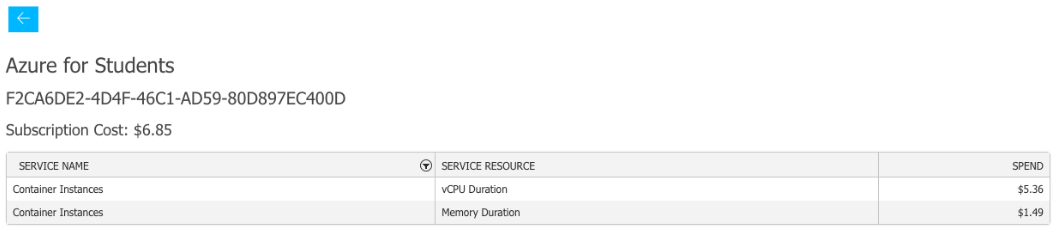
Pallets (2020). *Deploy to Production¶*. URL: https://flask.palletsprojects.com/en/1.1.x/tutorial/deploy/ (visited on 19/09/2020).

Stackoverflow (July 2020). *Stack Overflow Developer Survey 2020*. URL: https://insights.stackoverflow.com/survey/2020 (visited on 24/10/2020).

Van Eyk, Erwin et al. (2017). 'The SPEC cloud group's research vision on FaaS and serverless architectures'. In: *Proceedings of the 2nd International Workshop on Serverless Computing*, pp. 1–4.

6 Appendices

A Azure Billing to Run Performance Tests



Azure for Students

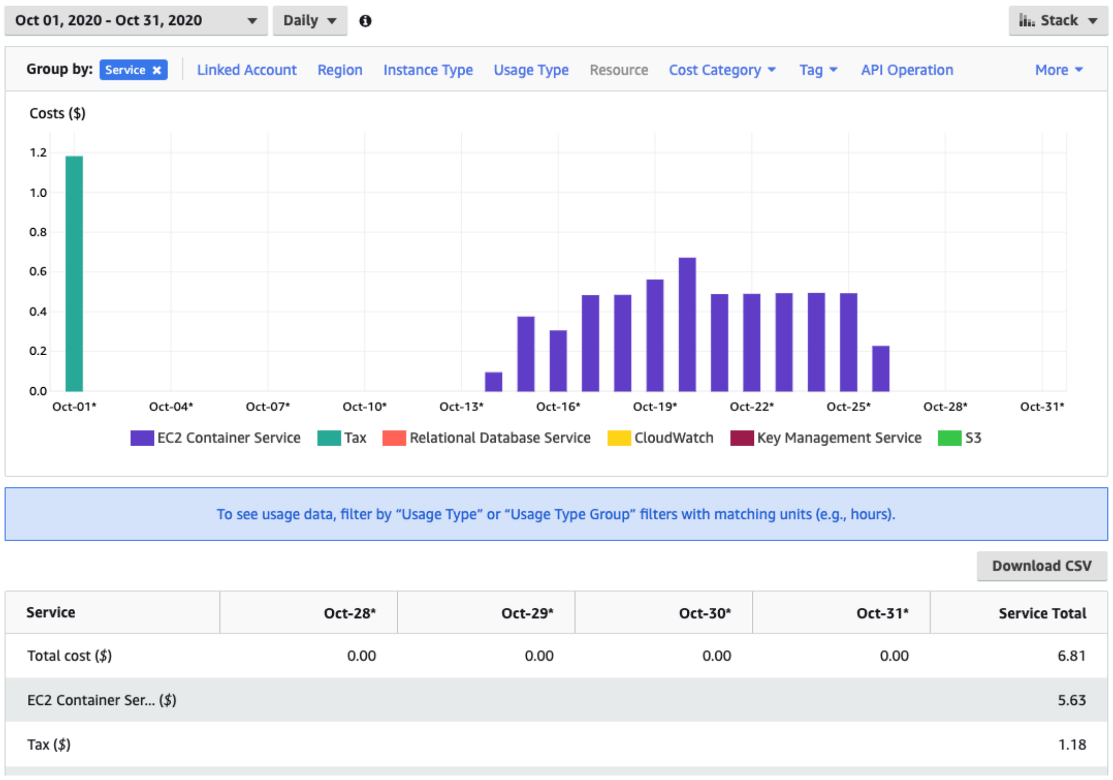
F2CA6DE2-4D4F-46C1-AD59-80D897EC400D

Subscription Cost: $6.85

SERVICE NAME	SERVICE RESOURCE	SPEND
Container Instances	vCPU Duration	$5.36
Container Instances	Memory Duration	$1.49

Figure 32: Azure real costs to run performance tests from section 4.4

B AWS Billing to Run Performance Tests



To see usage data, filter by "Usage Type" or "Usage Type Group" filters with matching units (e.g., hours).

Service	Oct-28*	Oct-29*	Oct-30*	Oct-31*	Service Total
Total cost ($)	0.00	0.00	0.00	0.00	6.81
EC2 Container Ser... ($)					5.63
Tax ($)					1.18

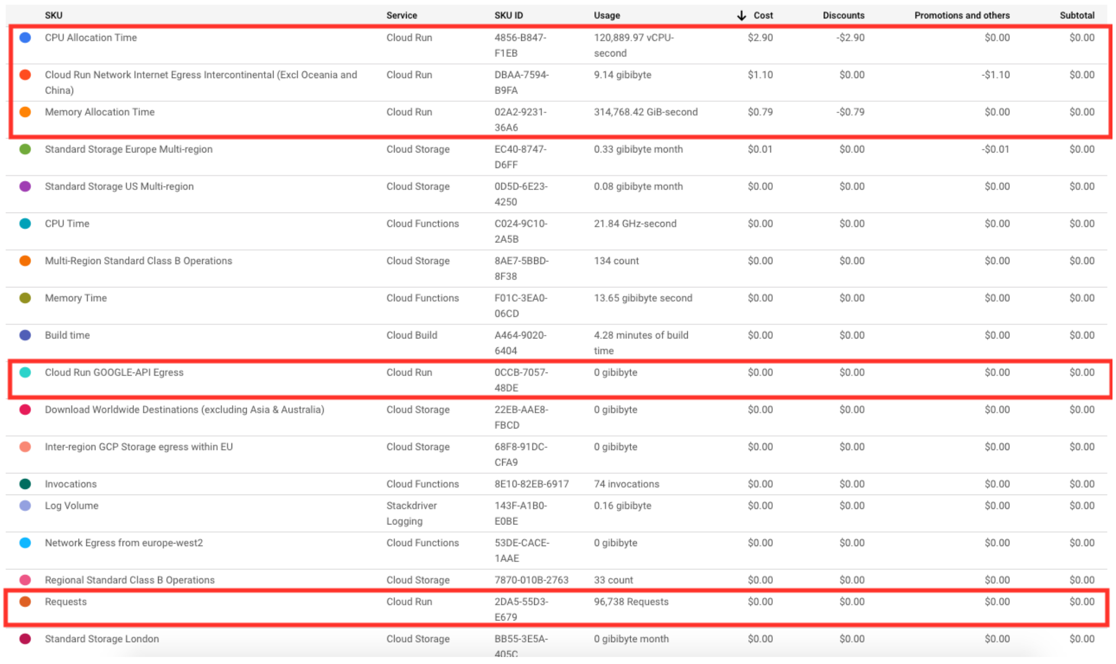
Figure 33: AWS real costs to run performance tests from section 4.4.

C Baseline of metrics from the application execution time of containerised application

Combination	Type	Min (ms)	Avg (ms)	Max (ms)
1cpu/2mem	10!	0.0048	0.0281	0.5882
1cpu/2mem	1000!	0.3803	0.5626	1.2603
1cpu/2mem	10000!	38.0433	43.4379	66.9341
1cpu/2mem	32000!	459.0015	540.6431	936.6241
1cpu/2mem	43000!	864.2752	1061.3826	1851.6071
1cpu/2mem	50000!	1203.205	1601.3173	2417.2188
1cpu/2mem	60000!	2245.9392	2705.1585	3271.6621
1cpu/2mem	hello world	0.0005	0.0086	0.0193
1cpu/3mem	10!	0.0088	0.018	0.057

1cpu/3mem	1000!	0.3719	0.5359	0.9241
1cpu/3mem	10000!	39.3248	41.8591	46.793
1cpu/3mem	32000!	482.6093	537.4373	711.3054
1cpu/3mem	43000!	901.4103	1050.4259	1433.524
1cpu/3mem	50000!	1241.3037	1548.6138	1977.8218
1cpu/3mem	60000!	2269.1555	2712.317	3552.4744
1cpu/3mem	hello world	0.0067	0.0106	0.0176
1cpu/4mem	10!	0.0069	0.0168	0.0508
1cpu/4mem	1000!	0.3893	0.553	0.8979
1cpu/4mem	10000!	39.4616	41.6895	47.3628
1cpu/4mem	32000!	478.236	536.5177	774.9054
1cpu/4mem	43000!	872.4742	1025.7688	1291.4561
1cpu/4mem	50000!	1266.8352	1562.7032	2567.4065
1cpu/4mem	60000!	2350.2976	2681.7906	3281.5498
1cpu/4mem	hello world	0.0072	0.0122	0.1264
2cpu/4mem	10!	0.0069	0.0179	0.0377
2cpu/4mem	1000!	0.3731	0.5911	0.9615
2cpu/4mem	10000!	39.6836	41.6221	51.5628
2cpu/4mem	32000!	478.3809	522.7921	585.8946
2cpu/4mem	43000!	904.5238	1033.3186	1234.9231
2cpu/4mem	50000!	1236.1243	1451.7842	1609.5278
2cpu/4mem	60000!	2293.5076	2606.8713	3179.7356
2cpu/4mem	hello world	0.0069	0.0114	0.0429

D GCP Billing to Run Performance Tests



	SKU	Service	SKU ID	Usage	↓ Cost	Discounts	Promotions and others	Subtotal
●	CPU Allocation Time	Cloud Run	4856-B847-F1EB	120,889.97 vCPU-second	$2.90	-$2.90	$0.00	$0.00
●	Cloud Run Network Internet Egress Intercontinental (Excl Oceania and China)	Cloud Run	DBAA-7594-B9FA	9.14 gibibyte	$1.10	$0.00	-$1.10	$0.00
●	Memory Allocation Time	Cloud Run	02A2-9231-36A6	314,768.42 GiB-second	$0.79	-$0.79	$0.00	$0.00
●	Standard Storage Europe Multi-region	Cloud Storage	EC40-8747-D6FF	0.33 gibibyte month	$0.01	$0.00	-$0.01	$0.00
●	Standard Storage US Multi-region	Cloud Storage	0D5D-6E23-4250	0.08 gibibyte month	$0.00	$0.00	$0.00	$0.00
●	CPU Time	Cloud Functions	C024-9C10-2A5B	21.84 GHz-second	$0.00	$0.00	$0.00	$0.00
●	Multi-Region Standard Class B Operations	Cloud Storage	8AE7-5BBD-8F38	134 count	$0.00	$0.00	$0.00	$0.00
●	Memory Time	Cloud Functions	F01C-3EA0-06CD	13.65 gibibyte second	$0.00	$0.00	$0.00	$0.00
●	Build time	Cloud Build	A464-9020-6404	4.28 minutes of build time	$0.00	$0.00	$0.00	$0.00
●	Cloud Run GOOGLE-API Egress	Cloud Run	0CCB-7057-48DE	0 gibibyte	$0.00	$0.00	$0.00	$0.00
●	Download Worldwide Destinations (excluding Asia & Australia)	Cloud Storage	22EB-AAE8-FBCD	0 gibibyte	$0.00	$0.00	$0.00	$0.00
●	Inter-region GCP Storage egress within EU	Cloud Storage	68F8-91DC-CFA9	0 gibibyte	$0.00	$0.00	$0.00	$0.00
●	Invocations	Cloud Functions	8E10-82EB-6917	74 invocations	$0.00	$0.00	$0.00	$0.00
●	Log Volume	Stackdriver Logging	143F-A1B0-E0BE	0.16 gibibyte	$0.00	$0.00	$0.00	$0.00
●	Network Egress from europe-west2	Cloud Functions	53DE-CACE-1AAE	0 gibibyte	$0.00	$0.00	$0.00	$0.00
●	Regional Standard Class B Operations	Cloud Storage	7870-010B-2763	33 count	$0.00	$0.00	$0.00	$0.00
●	Requests	Cloud Run	2DA5-55D3-E679	96,738 Requests	$0.00	$0.00	$0.00	$0.00
●	Standard Storage London	Cloud Storage	BB55-3E5A-405C	0 gibibyte month	$0.00	$0.00	$0.00	$0.00

Figure 34: GCP real costs to run performance tests from section 4.4. Values are being deducted from the column "Promotions and others" as there were student credits in the GCP account utilised.

E K6 Test Script for High CPU Endpoint

```
import http from 'k6/http';
import { check, group, sleep, fail } from 'k6';
import { Trend } from 'k6/metrics';

let app_execution_time = new Trend('app_execution_time');

export let options = {
  vus: 1,  // 1 user looping for 1 minute
  iterations: 10, //Hardcoded number of iterations
};

export default () => {
  let req = http.get(`${__ENV.DOMAIN}/${__ENV.ENDPOINT}`);

  check(req, {
    'status is 200': r => r.status === 200,
  });

  app_execution_time.add(req.json('duration')*1000);

  sleep(0.1)
}
```

F K6 Test Script for Hello World Endpoint

```
import http from 'k6/http';
import { check, group, sleep, fail } from 'k6';
import { Trend } from 'k6/metrics';

let app_execution_time = new Trend('app_execution_time');

export let options = {
  vus: 1,  // 1 user looping for 1 minute
  iterations: 10, //Hardcoded number of iterations
};

export default () => {
  let req = http.get(`${__ENV.DOMAIN}/${__ENV.ENDPOINT}`);

  check(req, {
    'status is 200': r => r.status === 200,
  });

  check(req, { 'show correct data': (resp) => resp.json('data') ==
  ↪  "Hello World MSC - Container as a Service Comparison" });
  app_execution_time.add(req.json('duration')*1000);

  sleep(0.1)
}
```

G MySQL query to retrieve application execution time data for comparison between weekends and weekdays

```
SELECT
    results.csp,
    results.factorial,
    ROUND(SUM(CASE results.weekend
             WHEN 'Weekend' THEN -results.app_execution_time_avg
             WHEN 'Monday to Friday' THEN
              ↪ results.app_execution_time_avg
           END), 6) AS difference_ms,
    IF(SUM(CASE results.weekend
             WHEN 'Weekend' THEN -results.app_execution_time_avg
             WHEN 'Monday to Friday' THEN
              ↪ results.app_execution_time_avg
           END)  > 0, "Weekend Slower", "Weekend Faster") as
            ↪ resolution,
results.count as samples
FROM (
  SELECT
    csp,
    factorial,
    AVG(app_execution_time_avg) as app_execution_time_avg,
    count(id) as count,
    IF(WEEKDAY(datetime) in (5,6), "Weekend", "Monday to Friday")
     ↪  as weekend
  FROM metrics
  WHERE
    memory = 2 AND
    cpu = 1 AND
    type = 'factorial'
    AND csp != 'docker'

  GROUP BY csp, factorial, weekend
  ORDER BY csp
) as results
GROUP BY csp, factorial
```

H Plots for application execution time in all CSPs for all tests using combination B

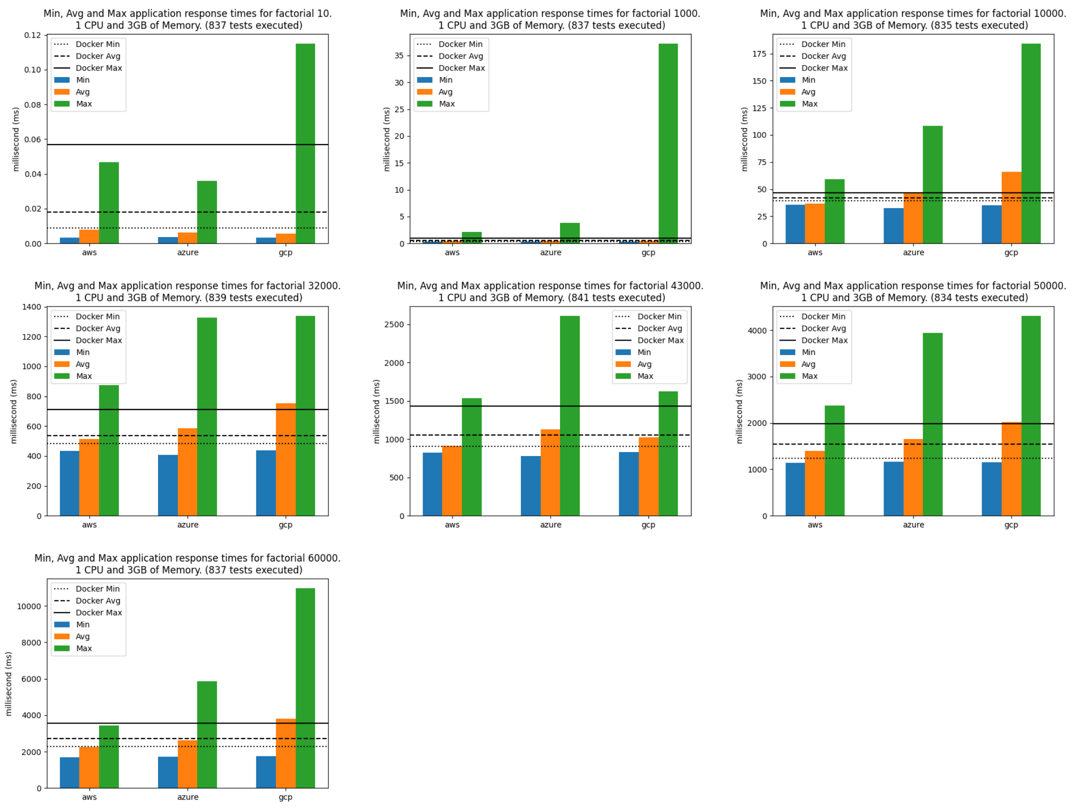


Figure 35: Minimum, average and maximum plots for application execution time in all CSPs for all tests using combination B (1 CPU and 3GB of memory).

I Plots for application execution time in all CSPs for all tests using combination C

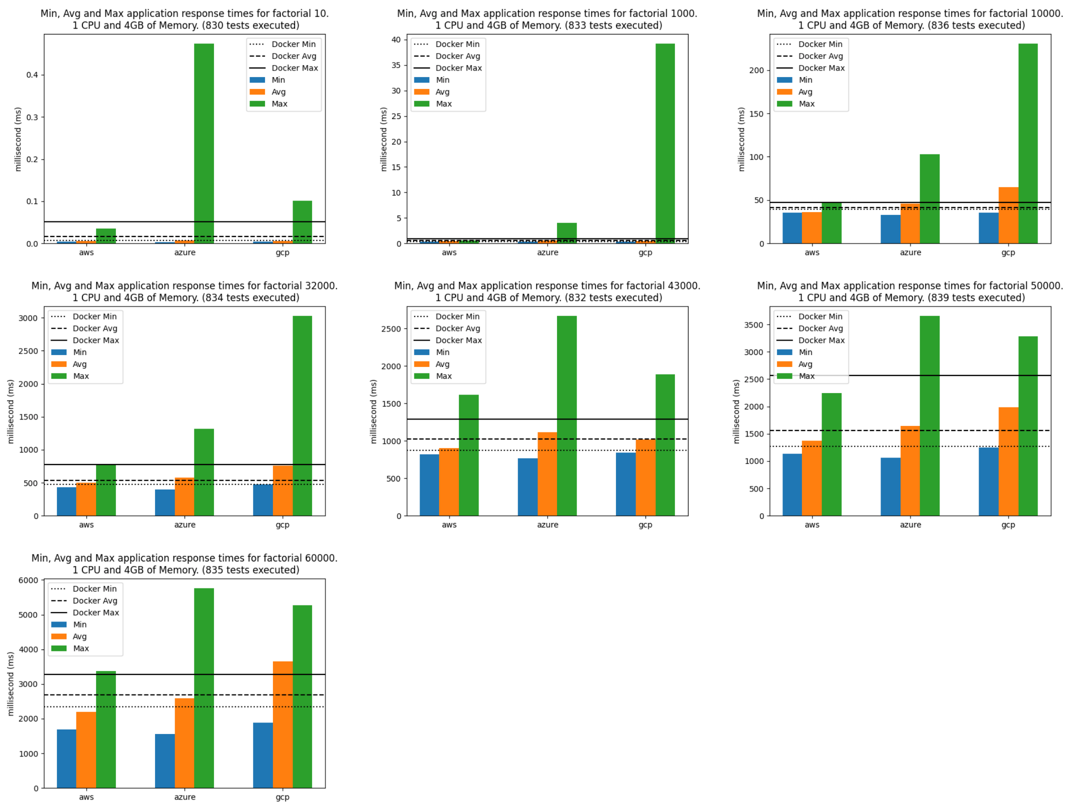


Figure 36: Minimum, average and maximum plots for application execution time in all CSPs for all tests using combination C (1 CPU and 4GB of memory).

J Plots for application execution time in all CSPs for all tests using combination D

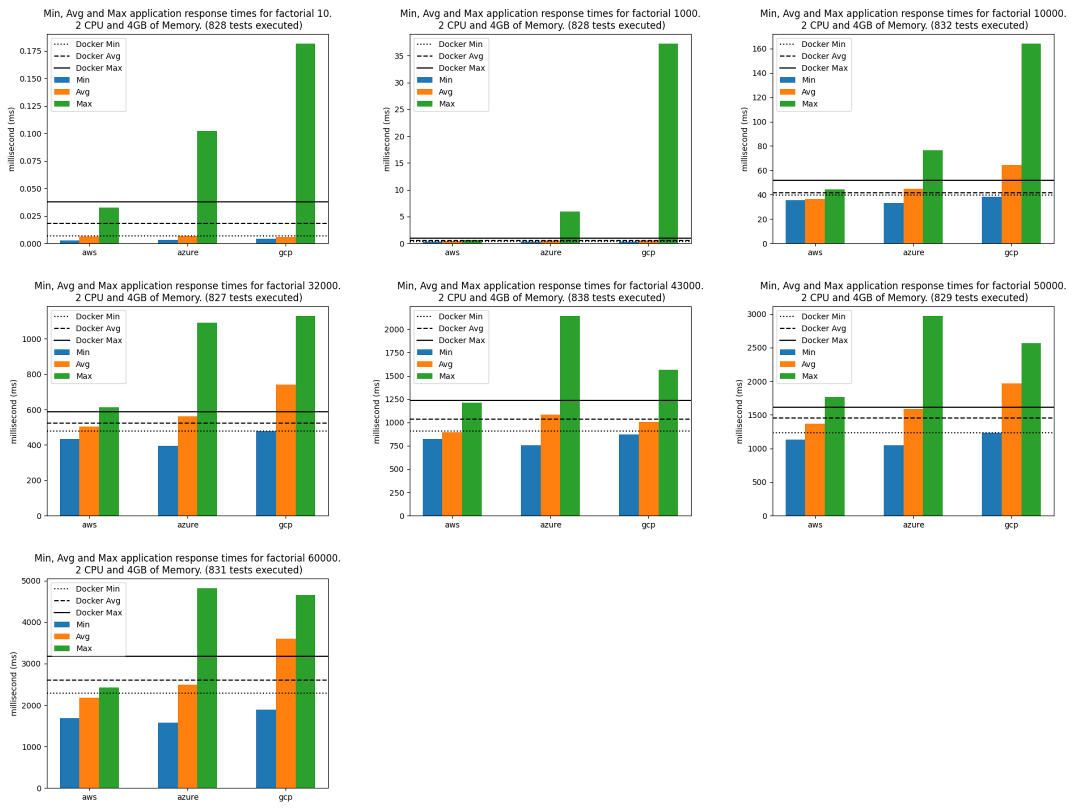


Figure 37: Minimum, average and maximum plots for application execution time in all CSPs for all tests using combination D (2 CPU and 4GB of memory).

K Python Application

```
import time
from flask import Flask
from waitress import serve
from logging.config import dictConfig
dictConfig({
    'version': 1,
    'formatters': {'default': {
        'format': '[%(asctime)s] %(levelname)s in %(module)s:
        ↪  %(message)s',
    }},
```

```python
    'handlers': {'wsgi': {
        'class': 'logging.StreamHandler',
        'stream': 'ext://sys.stdout',
        'formatter': 'default'
    }},
    'root': {
        'level': 'INFO',
        'handlers': ['wsgi']
    }
})

app = Flask(__name__)


@app.route("/")
def hello_world():
    """
    Simple hello work function

    Returns:
        String: hello world string
    """
    app.logger.info("GET /")
    start = time.time()
    data = "Hello World MSC - Container as a Service Comparison"
    diff = float("%.7f" % (time.time() - start))

    return {
        "data": data,
        "duration": diff
    }, 200

```

```python

@app.route("/cpu/factorial/<int:number>")
def calculate_factorial(number):
    """
    Given an integer number, returns its factorial.
    Args:
        number (int): Positive Integer

    Returns:
        String: Text of the factorial
    """
    app.logger.info(f"GET /cpu/factorial/{number}")
    start = time.time()

    if number < 0:
        return "Number needs to be positive", 400
    elif number == 0:
        result = f"Factorial of 0 is: 1"
    else:
        factorial = 1
        for i in range(1, number + 1):
            factorial = factorial*i
        result = f"Factorial of {number} is: {factorial}"

    end = time.time()
    # Precision of 6 decimals
    diff = float("%.7f" % (end - start))
    app.logger.info(f"GET /cpu/factorial/{number} took {diff}
    ↪  seconds")

    return {
        "data": result,
            "start": start,
            "end": end,
            "duration": diff
        }, 200


if __name__ == "__main__":
    # Serving the app using Waitress
    app.logger.info("Starting Webserver")

    serve(app, host='0.0.0.0', port=80, threads=4)
```

L Python Script to Collect Performance Metrics

```python
import os
import logging
from datetime import datetime
import time
import json
import glob
import re
import MySQLdb


logging.basicConfig(
    format='%(asctime)s - %(levelname)s - %(message)s',
    level=logging.DEBUG,
    handlers=[
        logging.FileHandler("execution.log"),
        logging.StreamHandler()
    ]
)
```

```python


class MetricReader:
    FILE_NAME_PATTERN = r"metrics/(?P<datetime>[0-9_]{19})_(?P<type>factorial_(?P<factorial>\d{1,6})|hello_world)_(?P<csp>[a-zA-Z]*)_(?P<cpu>[0-9Mim.]*)_cpu_(?P<memory>[0-9Mim.]*)_memory.json"
    DB_HOST = os.environ.get('DB_HOST')
    DB_USER = os.environ.get('DB_USER')
    DB_PASS = os.environ.get('DB_PASS')
    DB_NAME = os.environ.get('DB_NAME')

    def __init__(self) -> None:
        self.regex_filename = re.compile(self.FILE_NAME_PATTERN)
        self.db = MySQLdb.connect(
            host=self.DB_HOST, db=self.DB_NAME, user=self.DB_USER, passwd=self.DB_PASS)
        self.db_c = self.db.cursor()

    def read_all(self):
        data = []

        files = glob.glob("metrics/*.json")

        max_per_insert = 10
        count = 0

        for _, filename in enumerate(files):
            data_filename = self.extract_data_from_filename(filename)
            data_content = self.extract_data_from_content(filename)
            full_data = {**data_filename, **data_content}
            new = tuple(full_data.values())
            if len(new) == 77:
                count += 1
                data.append(new)
            else:
                print(f"Problem with file {filename}")


            if max_per_insert == count:
                self.save_in_db(data)
                data = []

        if data != []:
            self.save_in_db(data)

        # Moving read files
        for _, filename in enumerate(files):
            os.rename(filename, filename.replace('metrics/',
             ↪  'saved_metrics/'))

    def extract_data_from_filename(self, filename):

        match = self.regex_filename.match(filename)
        data = match.groupdict()
        # fixing some
        data['datetime'] = data['datetime'].replace(
            '_', '-', 2).replace('_', ' ', 1).replace('_', ':')
        data['type'] = data['type'].split('_')[0]
        data['cpu'] = data['cpu'].split('.')[0]
        data['memory'] = data['memory'].split('.')[0]
        # normalizing CPU/Memory values
```

```python
        for item in (('1024', '1'), ('2048', '2'), ('3072', '3'), ('4096', '4'), ('Mi', '')):
            data['cpu'] = data['cpu'].replace(*item)
            data['memory'] = data['memory'].replace(*item)

        return data

    def extract_data_from_content(self, filename):

        try:
            f = open(filename, "r")
            raw = json.loads(f.read())

            data = {}
            for prefix in raw['metrics']:

                for metric, val in raw['metrics'][prefix].items():
                    metric = metric.replace('(', '').replace(')', '')
                    data[f"{prefix}_{metric}"] = val

            return data
        except Exception as e:
            logging.error(f"Failed to read file {filename}")
            logging.error(e)
            raise e

    def save_in_db(self, data):
        self.db_c.executemany(
            """INSERT INTO metrics (datetime, type, factorial,
                ↪  csp, cpu, memory, app_execution_time_avg,
                ↪  app_execution_time_max, app_execution_time_med,
                ↪  app_execution_time_min, app_execution_time_p90,
                ↪  app_execution_time_p95, checks_fails,
                ↪  checks_passes, checks_value, data_received_count,
                ↪  data_received_rate, data_sent_count,
                ↪  data_sent_rate, http_req_blocked_avg,
                ↪  http_req_blocked_max, http_req_blocked_med,
                ↪  http_req_blocked_min, http_req_blocked_p90,
                ↪  http_req_blocked_p95, http_req_connecting_avg,
                ↪  http_req_connecting_max, http_req_connecting_med,
                ↪  http_req_connecting_min, http_req_connecting_p90,
                ↪  http_req_connecting_p95, http_req_duration_avg,
                ↪  http_req_duration_max, http_req_duration_med,
                ↪  http_req_duration_min, http_req_duration_p90,
                ↪  http_req_duration_p95, http_req_receiving_avg,
                ↪  http_req_receiving_max, http_req_receiving_med,
                ↪  http_req_receiving_min, http_req_receiving_p90,
                ↪  http_req_receiving_p95, http_req_sending_avg,
                ↪  http_req_sending_max, http_req_sending_med,
                ↪  http_req_sending_min, http_req_sending_p90,
                ↪  http_req_sending_p95,
                ↪  http_req_tls_handshaking_avg,
                ↪  http_req_tls_handshaking_max,
                ↪  http_req_tls_handshaking_med,
                ↪  http_req_tls_handshaking_min,
                ↪  http_req_tls_handshaking_p90,
                ↪  http_req_tls_handshaking_p95,
                ↪  http_req_waiting_avg, http_req_waiting_max,
                ↪  http_req_waiting_med, http_req_waiting_min,
                ↪  http_req_waiting_p90, http_req_waiting_p95,
                ↪  http_reqs_count, http_reqs_rate,
                ↪  iteration_duration_avg, iteration_duration_max,
                ↪  iteration_duration_med, iteration_duration_min,
                ↪  iteration_duration_p90, iteration_duration_p95,
                ↪  iterations_count, iterations_rate, vus_max,
                ↪  vus_min, vus_value, vus_max_max, vus_max_min,
                ↪  vus_max_value)
```

```
            VALUES (%s, %s, %s, %s, %s, %s, %s,%s, %s, %s, %s, %s, %s, %s,%s, %s, %s, %s, %s, %s, %s,%s, %s, %s, %s, %s, %s, %s,%s, %s, %s, %s, %s, %s, %s,%s, %s, %s, %s, %s, %s, %s,%s, %s, %s, %s, %s,%s, %s, %s, %s, %s, %s,%s, %s, %s, %s, %s, %s, %s,%s, %s, %s, %s, %s, %s, %s,%s, %s, %s, %s, %s, %s, %s )""",
            data
        )
        self.db.commit()


runner = MetricReader()
sleep = 30

while True:
    logging.info(f"Collecting Metrics...")
    runner.read_all()
    sleep_time_remaining = sleep
    while (sleep_time_remaining > 0):
        logging.info(f"Sleeping for {sleep_time_remaining} minutes...")
        time.sleep(60)
        sleep_time_remaining -= 1
```

M Python Script to Run Performance Tests

```
import os
import sys
import logging
from datetime import datetime
import time
import json
```

```python


logging.basicConfig(
    format='%(asctime)s - %(levelname)s - %(message)s',
    level=logging.DEBUG,
    handlers=[
        logging.FileHandler("execution.log"),
        logging.StreamHandler()
    ]
)


class Runner:
    TIME_FORMAT = "%Y_%m_%d_%H_%M_%S"
    CASES = [
        "k6 run -e DOMAIN='{domain}' -e ENDPOINT='' hello-world.js
         ↪  --insecure-skip-tls-verify --address localhost:0
         ↪  --summary-export=metrics/{now}_hello_world_{hosting}_{⌋
         ↪  cpu}_cpu_{memory}_memory.json",
        # 0.00001
        "k6 run -e DOMAIN='{domain}' -e
         ↪  ENDPOINT='cpu/factorial/10' cpu.js
         ↪  --insecure-skip-tls-verify --address localhost:0
         ↪  --summary-export=metrics/{now}_factorial_10_{hosting}_⌋
         ↪  {cpu}_cpu_{memory}_memory.json",
        # 0.0005
        "k6 run -e DOMAIN='{domain}' -e
         ↪  ENDPOINT='cpu/factorial/1000' cpu.js
         ↪  --insecure-skip-tls-verify --address localhost:0
         ↪  --summary-export=metrics/{now}_factorial_1000_{hosting⌋
         ↪  }_{cpu}_cpu_{memory}_memory.json",
        # 0.05
        "k6 run -e DOMAIN='{domain}' -e
         ↪  ENDPOINT='cpu/factorial/10000' cpu.js
         ↪  --insecure-skip-tls-verify --address localhost:0
         ↪  --summary-export=metrics/{now}_factorial_10000_{hostin⌋
         ↪  g}_{cpu}_cpu_{memory}_memory.json",
        # 0.5
        "k6 run -e DOMAIN='{domain}' -e
         ↪  ENDPOINT='cpu/factorial/32000' cpu.js
         ↪  --insecure-skip-tls-verify --address localhost:0
         ↪  --summary-export=metrics/{now}_factorial_32000_{hostin⌋
         ↪  g}_{cpu}_cpu_{memory}_memory.json",
        # 1s
        "k6 run -e DOMAIN='{domain}' -e
         ↪  ENDPOINT='cpu/factorial/43000' cpu.js
         ↪  --insecure-skip-tls-verify --address localhost:0
         ↪  --summary-export=metrics/{now}_factorial_43000_{hostin⌋
         ↪  g}_{cpu}_cpu_{memory}_memory.json",
        # 2s
        "k6 run -e DOMAIN='{domain}' -e
         ↪  ENDPOINT='cpu/factorial/50000' cpu.js
         ↪  --insecure-skip-tls-verify --address localhost:0
         ↪  --summary-export=metrics/{now}_factorial_50000_{hostin⌋
         ↪  g}_{cpu}_cpu_{memory}_memory.json",
        # 3s
        "k6 run -e DOMAIN='{domain}' -e
         ↪  ENDPOINT='cpu/factorial/60000' cpu.js
         ↪  --insecure-skip-tls-verify --address localhost:0
         ↪  --summary-export=metrics/{now}_factorial_60000_{hostin⌋
         ↪  g}_{cpu}_cpu_{memory}_memory.json",
    ]
    SCENARIOS = [
        {
            "hosting": "docker",
            "configs": [
                {
                    "cpu": '1',
                    "memory": '2'
                },
                {
                    "cpu": '1',
                    "memory": '3'
                },
                {
                    "cpu": '1',
                    "memory": '4'
                },
                {
                    "cpu": '2',
                    "memory": '4'
                }
            ],
            "start": "docker run --rm -d -p 8080:80
             ↪  --name=caas-local --cpus={cpu} --memory={memory}g
             ↪  msc-performance-web-app:latest",
            "stop": "docker stop caas-local && docker rm
             ↪  caas-local",
            "domain": "http://localhost:8080"
        },
        {
            "hosting": "gcp",
            "configs": [
                {
                    "cpu": '1.0',
                    "memory": '2048Mi'
                },
                {
                    "cpu": '1.0',
                    "memory": '3072Mi'
                },
                {
                    "cpu": '1.0',
                    "memory": '4096Mi'
                },
                {
                    "cpu": '2.0',
                    "memory": '4096Mi'
                }
            ],
            "start": "cd ../../terraform/gcp && terraform plan
             ↪  -out plan -var 'cpu={cpu}' -var 'memory={memory}'
             ↪  && terraform apply -auto-approve -var 'cpu={cpu}'
             ↪  -var 'memory={memory}'",
            "stop": "cd ../../terraform/gcp && terraform destroy
             ↪  -auto-approve",
            "get_url": 'cd ../../terraform/gcp && terraform output
             ↪  -json service_url'
        },
        {
            "hosting": "aws",
            "configs": [
                {
                    "cpu": '1024',
                    "memory": '2048'
                },
                {
                    "cpu": '1024',
                    "memory": '3072'
                },
                {
                    "cpu": '1024',
                    "memory": '4096'
                },
                {
                    "cpu": '2048',
                    "memory": '4096'
                }
            ],
            "start": "cd ../../terraform/aws && terraform plan
             ↪  -out plan -var 'cpu={cpu}' -var 'memory={memory}'
             ↪  && terraform apply -auto-approve -var 'cpu={cpu}'
             ↪  -var 'memory={memory}'",
            "stop": "cd ../../terraform/aws && terraform destroy
             ↪  -auto-approve",
            "get_networks": 'aws ec2 describe-network-interfaces'
        },
        {
            "hosting": "azure",
            "configs": [
                {
                    "cpu": '1',
                    "memory": '2'
                },
                {
                    "cpu": '1',
                    "memory": '3'
                },
                {
                    "cpu": '1',
                    "memory": '4'
                },
                {
                    "cpu": '2',
                    "memory": '4'
                }
            ],
            "start": "cd ../../terraform/azure && terraform plan
            ↪  -out plan -var 'cpu={cpu}' -var 'memory={memory}'
            ↪  && terraform apply -auto-approve -var 'cpu={cpu}'
            ↪  -var 'memory={memory}'",
            "stop": "cd ../../terraform/azure && terraform destroy
            ↪  -auto-approve",
            "get_url": 'cd ../../terraform/azure && terraform
            ↪  output -json service_url'
        }
    ]

    def __init__(self) -> None:
        self.sleep_time_between_cases = 10

        logging.info(
            f"We have {len(self.SCENARIOS)} scenario(s) to run.
            ↪  {len(self.CASES)} cases each.")

    def execute(self, csp_filter):
        logging.info("--------Starting Script Runner--------")

        for scenario in self.SCENARIOS:
            hosting = scenario.get('hosting')
            # Skipping execution
            if csp_filter not in ['all', hosting]:
                logging.info(f"Skipping execution of {hosting}")
                continue

            if hosting == 'docker':
                self.run_docker(scenario)
            elif hosting == 'gcp':
                self.run_gcp(scenario)
            elif hosting == 'aws':
                self.run_aws(scenario)
            elif hosting == 'azure':
                self.run_azure(scenario)
            else:
                logging.warning(f'Hosting {hosting} is not yet
                 ↪  supported')

        logging.info("--------Ending Script Runner--------")

    def run_command(self, command):
        logging.info(f"Running command: {command}")
        stream = os.popen(command)
        output = stream.read()

        logging.info(f"Output: {output}")
        return output

    def run_docker(self, scenario):
        logging.info(f"Let's run this Docker scenario.")
        logging.info(f"{len(scenario.get('configs',[]))} config(s)
         ↪  to run!")

        for conf in scenario.get('configs', []):
            cpu = conf.get('cpu')
```

```python
            memory = conf.get('memory')

            # Starting Container
            start_cmd = scenario.get('start').format(
                cpu=cpu,
                memory=memory
            )

            logging.info(f"Running docker w/ CPU: {cpu} and Memory: {memory}")
            self.run_command(start_cmd)

            # Running Test Cases
            for case in self.CASES:
                run_cmd = case.format(cpu=cpu, memory=memory, hosting=scenario.get(
                    'hosting'), domain=scenario.get('domain'), now=datetime.now().strftime(self.TIME_FORMAT))
                self.run_command(run_cmd)
                # Sleeping between
                time.sleep(self.sleep_time_between_cases)

            # Stopping Container
            logging.info(f"Stopping Docker w/ CPU: {cpu} and Memory: {memory}")
            self.run_command(scenario.get('stop'))

    def run_gcp(self, scenario):
        logging.info(f"Let's run this GCP scenario.")
        logging.info(f"{len(scenario.get('configs',[]))} config(s) to run!")

        for conf in scenario.get('configs', []):
            cpu = conf.get('cpu')
            memory = conf.get('memory')

            start_cmd = scenario.get('start').format(
                cpu=cpu,
                memory=memory
            )

            # Provisioning infrastructure
            logging.info(
                f"Creating GCP Cloud Run w/ CPU: {cpu} and Memory:
                ↪  {memory}")
            self.run_command(start_cmd)

            # Getting generated dynamic URL
            output = json.loads(self.run_command(scenario.get('get
            ↪  _url')))
            url = output[0].get('url')

            # Running Test Cases
            for case in self.CASES:
                run_cmd = case.format(cpu=cpu, memory=memory,
                ↪  hosting=scenario.get(
                    'hosting'), domain=url, now=datetime.now().str
                    ↪  ftime(self.TIME_FORMAT))
                self.run_command(run_cmd)
                # Sleeping between
                time.sleep(self.sleep_time_between_cases)

            # Stopping Container
            logging.info(
                f"Destroying GCP Cloud Run w/ CPU: {cpu} and
                 ↪  Memory: {memory}")
            self.run_command(scenario.get('stop'))

    def run_aws(self, scenario):
        logging.info(f"Let's run this AWS ECS Fargate scenario.")
        logging.info(f"{len(scenario.get('configs',[]))} config(s)
         ↪  to run!")

        for conf in scenario.get('configs', []):
            cpu = conf.get('cpu')
            memory = conf.get('memory')

            start_cmd = scenario.get('start').format(
                cpu=cpu,
                memory=memory
            )

            # Provisioning infrastructure
            logging.info(
                f"Creating AWS ECS Fargate w/ CPU: {cpu} and
                 ↪  Memory: {memory}")
            self.run_command(start_cmd)

            has_dns = False
            while has_dns != True:
                try:
                    networks = json.loads(self.run_command(
                        scenario.get('get_networks')))
                    url = 'http://' + \
                        networks['NetworkInterfaces'][0]['Associat
                         ↪  ion']['PublicDnsName']
```

```python
                has_dns = True

            except:
                logging.warning(
                    'DNS Still not available... Sleeping for 90 seconds')
                time.sleep(90)

        # Getting instance IP
        # Note: Unfortunatelly  ECS/Terraform dont provide an easy way to get the IP or domain

        # Running Test Cases
        for case in self.CASES:
            run_cmd = case.format(cpu=cpu, memory=memory, hosting=scenario.get(
                'hosting'), domain=url, now=datetime.now().strftime(self.TIME_FORMAT))
            self.run_command(run_cmd)
            # Sleeping between
            time.sleep(self.sleep_time_between_cases)

        # Stopping Container
        logging.info(
            f"Destroying AWS ECS Fargate w/ CPU: {cpu} and Memory: {memory}")
        self.run_command(scenario.get('stop'))

    def run_azure(self, scenario):
        logging.info(f"Let's run this Azure scenario.")
        logging.info(f"{len(scenario.get('configs',[]))} config(s) to run!")

        for conf in scenario.get('configs', []):
        cpu = conf.get('cpu')
        memory = conf.get('memory')

        start_cmd = scenario.get('start').format(
            cpu=cpu,
            memory=memory
        )

        # Provisioning infrastructure
        logging.info(
            f"Creating Azure Container Instance w/ CPU: {cpu}
            ↪   and Memory: {memory}")
        self.run_command(start_cmd)

        # Getting generated dynamic URL
        domain = json.loads(self.run_command(scenario.get('get⌋
        ↪   _url')))
        url = 'http://' + domain

        logging.info(
            'Sleeping for 60 sec waiting for service to be
            ↪   ready')
        time.sleep(60)

        # Running Test Cases
        for case in self.CASES:
            run_cmd = case.format(cpu=cpu, memory=memory,
            ↪   hosting=scenario.get(
                'hosting'), domain=url, now=datetime.now().str⌋
                ↪   ftime(self.TIME_FORMAT))
                self.run_command(run_cmd)
                # Sleeping between
                time.sleep(self.sleep_time_between_cases)

            # Stopping Container
            logging.info(
                f"Destroying Azure Container Instance w/ CPU:
                 ↪  {cpu} and Memory: {memory}")
            self.run_command(scenario.get('stop'))


runner = Runner()
csp_filter = sys.argv[1] if len(sys.argv) > 1 else 'all'
repeat = sys.argv[2] if len(sys.argv) > 2 else False
minutes_to_sleep = 60

if repeat:
while True:

        current_minute = int(time.strftime('%M'))
        while (current_minute != 0):
        logging.info(
            f"Time remaining until next execution
             ↪  {60-current_minute} minute(s)...")
        time.sleep(60)
        current_minute = int(time.strftime('%M'))

        logging.info("Let's do 1 more execution!")
        runner.execute(csp_filter)


else:
logging.info("Running only once.")
runner.execute(csp_filter)
```

N Ratios between min, max and avg in Performance from the application point of view

CSP	combination	factorial	max/min	max/avg	avg/min	count
aws	1cpu/2mem	10000	1.61	1.57	1.02	289
aws	1cpu/2mem	32000	2.04	1.73	1.18	284
aws	1cpu/2mem	43000	1.64	1.47	1.12	284
aws	1cpu/2mem	50000	1.89	1.54	1.22	284
aws	1cpu/2mem	60000	1.90	1.43	1.33	284
aws	1cpu/3mem	10000	1.65	1.62	1.02	284
aws	1cpu/3mem	32000	2.02	1.70	1.19	284
aws	1cpu/3mem	43000	1.87	1.67	1.12	286
aws	1cpu/3mem	50000	2.08	1.70	1.23	283
aws	1cpu/3mem	60000	2.04	1.54	1.33	284
aws	1cpu/4mem	10000	1.32	1.30	1.01	282
aws	1cpu/4mem	32000	1.79	1.54	1.16	282
aws	1cpu/4mem	43000	1.97	1.79	1.10	282
aws	1cpu/4mem	50000	1.98	1.64	1.20	283
aws	1cpu/4mem	60000	2.01	1.53	1.31	283
aws	2cpu/4mem	10000	1.25	1.23	1.01	278
aws	2cpu/4mem	32000	1.41	1.21	1.16	279
aws	2cpu/4mem	43000	1.48	1.35	1.09	280
aws	2cpu/4mem	50000	1.56	1.30	1.20	281
aws	2cpu/4mem	60000	1.45	1.11	1.30	281
azure	1cpu/2mem	10000	3.47	2.52	1.38	255
azure	1cpu/2mem	32000	3.23	2.33	1.39	255
azure	1cpu/2mem	43000	7.78	5.51	1.41	255
azure	1cpu/2mem	50000	3.45	2.30	1.50	254

azure	1cpu/2mem	60000	3.53	2.23	1.58	257
azure	1cpu/3mem	10000	3.31	2.34	1.41	255
azure	1cpu/3mem	32000	3.26	2.26	1.44	257
azure	1cpu/3mem	43000	3.36	2.32	1.45	259
azure	1cpu/3mem	50000	3.38	2.38	1.42	255
azure	1cpu/3mem	60000	3.40	2.24	1.52	257
azure	1cpu/4mem	10000	3.15	2.24	1.40	258
azure	1cpu/4mem	32000	3.32	2.27	1.46	254
azure	1cpu/4mem	43000	3.48	2.39	1.45	254
azure	1cpu/4mem	50000	3.45	2.23	1.54	258
azure	1cpu/4mem	60000	3.68	2.23	1.65	254
azure	2cpu/4mem	10000	2.32	1.71	1.36	257
azure	2cpu/4mem	32000	2.75	1.94	1.42	253
azure	2cpu/4mem	43000	2.83	1.97	1.43	259
azure	2cpu/4mem	50000	2.84	1.87	1.52	253
azure	2cpu/4mem	60000	3.06	1.93	1.58	255
gcp	1cpu/2mem	10000	4.80	2.59	1.85	295
gcp	1cpu/2mem	32000	3.04	1.90	1.60	295
gcp	1cpu/2mem	43000	1.84	1.51	1.22	295
gcp	1cpu/2mem	50000	2.15	1.36	1.58	295
gcp	1cpu/2mem	60000	2.61	1.36	1.92	295
gcp	1cpu/3mem	10000	5.22	2.78	1.88	296
gcp	1cpu/3mem	32000	3.04	1.78	1.71	298
gcp	1cpu/3mem	43000	1.95	1.59	1.23	296
gcp	1cpu/3mem	50000	3.74	2.13	1.75	296
gcp	1cpu/3mem	60000	6.26	2.88	2.17	296
gcp	1cpu/4mem	10000	6.45	3.55	1.81	296
gcp	1cpu/4mem	32000	6.36	4.01	1.59	298
gcp	1cpu/4mem	43000	2.25	1.85	1.21	296
gcp	1cpu/4mem	50000	2.64	1.65	1.60	298
gcp	1cpu/4mem	60000	2.79	1.44	1.93	298

gcp	2cpu/4mem	10000	4.29	2.55	1.68	297
gcp	2cpu/4mem	32000	2.36	1.52	1.55	295
gcp	2cpu/4mem	43000	1.79	1.55	1.15	299
gcp	2cpu/4mem	50000	2.08	1.31	1.59	295
gcp	2cpu/4mem	60000	2.47	1.30	1.91	295

O Results for comparison of application execution time showing whether weekend executions were faster than weekdays.

CaaS	Comb.	Test	Diff (ms)	Weekend	Samples
AWS ECS Fargate	A	10!	-0.000013	Faster	285
AWS ECS Fargate	A	1000!	-0.001564	Faster	287
AWS ECS Fargate	A	10000!	-0.028491	Faster	289
AWS ECS Fargate	A	32000!	-0.463619	Faster	284
AWS ECS Fargate	A	43000!	-4.957809	Faster	284
AWS ECS Fargate	A	50000!	10.51251	Slower	284
AWS ECS Fargate	A	60000!	4.52214	Slower	284
AWS ECS Fargate	A	Hello World	0.000017	Slower	275
AWS ECS Fargate	B	10!	0.000008	Slower	284
AWS ECS Fargate	B	1000!	0.001909	Slower	284
AWS ECS Fargate	B	10000!	0.035425	Slower	284
AWS ECS Fargate	B	32000!	9.365293	Slower	284
AWS ECS Fargate	B	43000!	3.606359	Slower	286
AWS ECS Fargate	B	50000!	12.824283	Slower	283
AWS ECS Fargate	B	60000!	54.637696	Slower	284
AWS ECS Fargate	B	Hello World	0.000011	Slower	275
AWS ECS Fargate	C	10!	0.00002	Slower	280
AWS ECS Fargate	C	1000!	-0.000647	Faster	281
AWS ECS Fargate	C	10000!	-0.021941	Faster	282

AWS ECS Fargate	C	32000!	-2.068172	Faster	282
AWS ECS Fargate	C	43000!	-4.15829	Faster	282
AWS ECS Fargate	C	50000!	-10.744209	Faster	283
AWS ECS Fargate	C	60000!	-31.555358	Faster	283
AWS ECS Fargate	C	Hello World	0.00001	Slower	276
AWS ECS Fargate	D	10!	-0.000134	Faster	276
AWS ECS Fargate	D	1000!	-0.001761	Faster	276
AWS ECS Fargate	D	10000!	0.018731	Slower	278
AWS ECS Fargate	D	32000!	0.092764	Slower	279
AWS ECS Fargate	D	43000!	5.629619	Slower	280
AWS ECS Fargate	D	50000!	-1.676839	Faster	281
AWS ECS Fargate	D	60000!	15.0487	Slower	281
AWS ECS Fargate	D	Hello World	-0.000022	Faster	271
Azure CI	A	10!	0.000135	Slower	255
Azure CI	A	1000!	0.007905	Slower	255
Azure CI	A	10000!	1.157723	Slower	255
Azure CI	A	32000!	15.976781	Slower	255
Azure CI	A	43000!	37.268035	Slower	255
Azure CI	A	50000!	48.74507	Slower	254
Azure CI	A	60000!	52.130799	Slower	257
Azure CI	A	Hello World	0.000068	Slower	255
Azure CI	B	10!	-0.000059	Faster	255
Azure CI	B	1000!	-0.008414	Faster	255
Azure CI	B	10000!	0.235929	Slower	255
Azure CI	B	32000!	2.631455	Slower	257
Azure CI	B	43000!	7.733767	Slower	259
Azure CI	B	50000!	28.47544	Slower	255
Azure CI	B	60000!	19.629036	Slower	257
Azure CI	B	Hello World	0.000057	Slower	258
Azure CI	C	10!	-0.000495	Faster	254
Azure CI	C	1000!	-0.004367	Faster	256

Azure CI	C	10000!	-1.171435	Faster	258
Azure CI	C	32000!	-3.506152	Faster	254
Azure CI	C	43000!	-9.526924	Faster	254
Azure CI	C	50000!	-24.183939	Faster	258
Azure CI	C	60000!	-10.294278	Faster	254
Azure CI	C	Hello World	-0.000081	Faster	253
Azure CI	D	10!	0.000186	Slower	253
Azure CI	D	1000!	-0.007376	Faster	253
Azure CI	D	10000!	-0.624329	Faster	257
Azure CI	D	32000!	-7.414965	Faster	253
Azure CI	D	43000!	-11.846569	Faster	259
Azure CI	D	50000!	-1.175192	Faster	253
Azure CI	D	60000!	8.693936	Slower	255
Azure CI	D	Hello World	0.000039	Slower	252
GCP Cloud Run	A	10!	-0.027061	Faster	297
GCP Cloud Run	A	1000!	-0.069279	Faster	295
GCP Cloud Run	A	10000!	3.779699	Slower	295
GCP Cloud Run	A	32000!	-10.738135	Faster	295
GCP Cloud Run	A	43000!	-4.295703	Faster	295
GCP Cloud Run	A	50000!	-16.681006	Faster	295
GCP Cloud Run	A	60000!	-59.849039	Faster	295
GCP Cloud Run	A	Hello World	-0.000001	Faster	295
GCP Cloud Run	B	10!	0.000236	Slower	298
GCP Cloud Run	B	1000!	0.034708	Slower	298
GCP Cloud Run	B	10000!	3.783559	Slower	296
GCP Cloud Run	B	32000!	9.024799	Slower	298
GCP Cloud Run	B	43000!	1.407438	Slower	296
GCP Cloud Run	B	50000!	50.892956	Slower	296
GCP Cloud Run	B	60000!	112.775989	Slower	296
GCP Cloud Run	B	Hello World	-0.000075	Faster	296
GCP Cloud Run	C	10!	-0.00018	Faster	296

GCP Cloud Run	C	1000!	0.073052	Slower	296
GCP Cloud Run	C	10000!	-0.037183	Faster	296
GCP Cloud Run	C	32000!	-8.816153	Faster	298
GCP Cloud Run	C	43000!	-3.327779	Faster	296
GCP Cloud Run	C	50000!	-10.581793	Faster	298
GCP Cloud Run	C	60000!	-25.751066	Faster	298
GCP Cloud Run	C	Hello World	-0.000004	Faster	296
GCP Cloud Run	D	10!	-0.000416	Faster	299
GCP Cloud Run	D	1000!	0.043396	Slower	299
GCP Cloud Run	D	10000!	-3.381901	Faster	297
GCP Cloud Run	D	32000!	-7.05718	Faster	295
GCP Cloud Run	D	43000!	-7.825849	Faster	299
GCP Cloud Run	D	50000!	-37.155265	Faster	295
GCP Cloud Run	D	60000!	-54.303237	Faster	295
GCP Cloud Run	D	Hello World	-0.000047	Faster	301

Table 23: Difference (Diff) calculated based on difference between average application execution time of weekdays and weekends. A negative difference means that executions were faster during the for that row. Combinations relate to Table 10.

P Results for comparison of internal CaaS processing time showing whether weekend executions were faster than weekdays.

CaaS	**Comb.**	**Test**	**Diff (ms)**	**Weekend**	**Samples**
AWS ECS Fargate	A	10!	2.64	Slower	285
AWS ECS Fargate	A	1000!	13.10	Slower	287
AWS ECS Fargate	A	10000!	-19.55	Faster	289
AWS ECS Fargate	A	32000!	10.83	Slower	284
AWS ECS Fargate	A	43000!	14.64	Slower	284

AWS ECS Fargate	A	50000!	7.45	Slower	284
AWS ECS Fargate	A	60000!	37.19	Slower	284
AWS ECS Fargate	A	Hello World	4.13	Slower	275
AWS ECS Fargate	B	10!	3.69	Slower	284
AWS ECS Fargate	B	1000!	10.77	Slower	284
AWS ECS Fargate	B	10000!	-0.52	Faster	284
AWS ECS Fargate	B	32000!	25.62	Slower	284
AWS ECS Fargate	B	43000!	18.16	Slower	286
AWS ECS Fargate	B	50000!	10.21	Slower	283
AWS ECS Fargate	B	60000!	4.47	Slower	284
AWS ECS Fargate	B	Hello World	3.08	Slower	275
AWS ECS Fargate	C	10!	0.00	Faster	280
AWS ECS Fargate	C	1000!	0.85	Slower	281
AWS ECS Fargate	C	10000!	-15.25	Faster	282
AWS ECS Fargate	C	32000!	17.45	Slower	282
AWS ECS Fargate	C	43000!	22.91	Slower	282
AWS ECS Fargate	C	50000!	15.81	Slower	283
AWS ECS Fargate	C	60000!	17.73	Slower	283
AWS ECS Fargate	C	Hello World	0.84	Slower	276
AWS ECS Fargate	D	10!	0.94	Slower	276
AWS ECS Fargate	D	1000!	1.99	Slower	276
AWS ECS Fargate	D	10000!	-8.93	Faster	278
AWS ECS Fargate	D	32000!	21.35	Slower	279
AWS ECS Fargate	D	43000!	23.31	Slower	280
AWS ECS Fargate	D	50000!	23.02	Slower	281
AWS ECS Fargate	D	60000!	13.53	Slower	281
AWS ECS Fargate	D	Hello World	3.89	Slower	271
Azure CI	A	10!	2.24	Slower	255
Azure CI	A	1000!	2.29	Slower	255
Azure CI	A	10000!	-26.53	Faster	255
Azure CI	A	32000!	7.70	Slower	255

Azure CI	A	43000!	7.52	Slower	255
Azure CI	A	50000!	10.99	Slower	254
Azure CI	A	60000!	-4.26	Faster	257
Azure CI	A	Hello World	0.72	Slower	255
Azure CI	B	10!	7.23	Slower	255
Azure CI	B	1000!	9.59	Slower	255
Azure CI	B	10000!	2.91	Slower	255
Azure CI	B	32000!	61.08	Slower	257
Azure CI	B	43000!	13.63	Slower	259
Azure CI	B	50000!	-9.26	Faster	255
Azure CI	B	60000!	2.03	Slower	257
Azure CI	B	Hello World	2.72	Slower	258
Azure CI	C	10!	-0.06	Faster	254
Azure CI	C	1000!	1.78	Slower	256
Azure CI	C	10000!	-13.29	Faster	258
Azure CI	C	32000!	12.88	Slower	254
Azure CI	C	43000!	11.91	Slower	254
Azure CI	C	50000!	15.09	Slower	258
Azure CI	C	60000!	4.37	Slower	254
Azure CI	C	Hello World	2.20	Slower	253
Azure CI	D	10!	6.00	Slower	253
Azure CI	D	1000!	0.06	Slower	253
Azure CI	D	10000!	3.98	Slower	257
Azure CI	D	32000!	13.86	Slower	253
Azure CI	D	43000!	9.11	Slower	259
Azure CI	D	50000!	13.77	Slower	253
Azure CI	D	60000!	5.24	Slower	255
Azure CI	D	Hello World	2.58	Slower	252
GCP Cloud Run	A	10!	2.60	Slower	297
GCP Cloud Run	A	1000!	0.88	Slower	295
GCP Cloud Run	A	10000!	-5.93	Faster	295

GCP Cloud Run	A	32000!	-14.89	Faster	295
GCP Cloud Run	A	43000!	27.04	Slower	295
GCP Cloud Run	A	50000!	11.46	Slower	295
GCP Cloud Run	A	60000!	43.22	Slower	295
GCP Cloud Run	A	Hello World	3.40	Slower	295
GCP Cloud Run	B	10!	0.66	Slower	298
GCP Cloud Run	B	1000!	0.89	Slower	298
GCP Cloud Run	B	10000!	5.46	Slower	296
GCP Cloud Run	B	32000!	5.99	Slower	298
GCP Cloud Run	B	43000!	-4.94	Faster	296
GCP Cloud Run	B	50000!	11.76	Slower	296
GCP Cloud Run	B	60000!	-5.54	Faster	296
GCP Cloud Run	B	Hello World	23.21	Slower	296
GCP Cloud Run	C	10!	2.34	Slower	296
GCP Cloud Run	C	1000!	1.52	Slower	296
GCP Cloud Run	C	10000!	4.59	Slower	296
GCP Cloud Run	C	32000!	17.18	Slower	298
GCP Cloud Run	C	43000!	27.64	Slower	296
GCP Cloud Run	C	50000!	25.09	Slower	298
GCP Cloud Run	C	60000!	-4114.53	Faster	298
GCP Cloud Run	C	Hello World	11.09	Slower	296
GCP Cloud Run	D	10!	3.15	Slower	299
GCP Cloud Run	D	1000!	4.81	Slower	299
GCP Cloud Run	D	10000!	5.58	Slower	297
GCP Cloud Run	D	32000!	17.41	Slower	295
GCP Cloud Run	D	43000!	34.29	Slower	299
GCP Cloud Run	D	50000!	10.92	Slower	295
GCP Cloud Run	D	60000!	40.37	Slower	295
GCP Cloud Run	D	Hello World	12.21	Slower	301

Table 24: Difference (Diff) calculated based on difference between average internal CaaS processing times of weekdays and weekends. A negative difference means that executions were faster during the for that row. Combinations relate to Table 10.

Q Serverless CaaS Internal Processing times

Serverless CaaS	Combination	Test	AET	TTFB	SCIPT	Samples
Docker Baseline	1cpu/2mem	10!	0.0281	4.56	4.53	8
Azure CI	1cpu/2mem	10!	0.0068	90.3	90.29	255
GCP Cloud Run	1cpu/2mem	10!	0.0234	130.04	130.02	297
AWS ECS Fargate	1cpu/2mem	10!	0.0078	130.35	130.34	285
Docker Baseline	1cpu/2mem	1000!	0.5626	4.55	3.99	8
Azure CI	1cpu/2mem	1000!	0.4947	90.22	89.72	255
GCP Cloud Run	1cpu/2mem	1000!	1.3819	131.77	130.39	295
AWS ECS Fargate	1cpu/2mem	1000!	0.4513	134.89	134.44	287
Docker Baseline	1cpu/2mem	10000!	43.4379	50.52	7.09	8
Azure CI	1cpu/2mem	10000!	46.4861	161.26	114.78	255
AWS ECS Fargate	1cpu/2mem	10000!	36.5181	172.66	136.14	289
GCP Cloud Run	1cpu/2mem	10000!	67.3364	222.96	155.62	295
Docker Baseline	1cpu/2mem	32000!	540.6431	547.8	7.16	8
Azure CI	1cpu/2mem	32000!	588.0645	734.73	146.67	255
AWS ECS Fargate	1cpu/2mem	32000!	511.052	700.65	189.6	284
GCP Cloud Run	1cpu/2mem	32000!	753.8966	993.43	239.53	295
Docker Baseline	1cpu/2mem	43000!	1061.3826	1067.62	6.24	8
Azure CI	1cpu/2mem	43000!	1130.9475	1274.24	143.29	255
AWS ECS Fargate	1cpu/2mem	43000!	917.2225	1095.91	178.68	284
GCP Cloud Run	1cpu/2mem	43000!	1024.066	1263.89	239.82	295
Docker Baseline	1cpu/2mem	50000!	1601.3173	1606.84	5.52	8
Azure CI	1cpu/2mem	50000!	1662.1217	1803.19	141.07	254
AWS ECS Fargate	1cpu/2mem	50000!	1391.9041	1570.86	178.96	284

GCP Cloud Run	1cpu/2mem	50000!	1974.4367	2189.28	214.84	295
Docker Baseline	1cpu/2mem	60000!	2705.1585	2712.38	7.22	8
Azure CI	1cpu/2mem	60000!	2628.1467	2785.13	156.99	257
AWS ECS Fargate	1cpu/2mem	60000!	2244.7776	2432.29	187.51	284
GCP Cloud Run	1cpu/2mem	60000!	3632.5272	3850.68	218.15	295
Docker Baseline	1cpu/2mem	hello world	0.0086	4.92	4.91	8
Azure CI	1cpu/2mem	hello world	0.0008	95.79	95.79	255
AWS ECS Fargate	1cpu/2mem	hello world	0.0018	128.85	128.85	275
GCP Cloud Run	1cpu/2mem	hello world	0.0007	194.65	194.65	295
Docker Baseline	1cpu/3mem	10!	0.018	4.29	4.27	6
Azure CI	1cpu/3mem	10!	0.0062	92	91.99	255
GCP Cloud Run	1cpu/3mem	10!	0.0055	120.57	120.57	298
AWS ECS Fargate	1cpu/3mem	10!	0.0078	128.39	128.38	284
Docker Baseline	1cpu/3mem	1000!	0.5359	4.8	4.27	6
Azure CI	1cpu/3mem	1000!	0.4902	93.17	92.68	255
GCP Cloud Run	1cpu/3mem	1000!	0.4907	125.05	124.56	298
AWS ECS Fargate	1cpu/3mem	1000!	0.4503	131.46	131.01	284
Docker Baseline	1cpu/3mem	10000!	41.8591	48.82	6.96	6
Azure CI	1cpu/3mem	10000!	46.3171	143.79	97.47	255
AWS ECS Fargate	1cpu/3mem	10000!	36.5137	161.56	125.04	284
GCP Cloud Run	1cpu/3mem	10000!	66.2337	200.56	134.32	296
Docker Baseline	1cpu/3mem	32000!	537.4373	543.26	5.82	6
Azure CI	1cpu/3mem	32000!	586.6297	752.23	165.6	257
GCP Cloud Run	1cpu/3mem	32000!	751.9686	933.87	181.9	298
AWS ECS Fargate	1cpu/3mem	32000!	513.7496	717.8	204.05	284
Docker Baseline	1cpu/3mem	43000!	1050.4259	1056.01	5.59	6
Azure CI	1cpu/3mem	43000!	1123.9896	1266.59	142.6	259
AWS ECS Fargate	1cpu/3mem	43000!	915.782	1103.55	187.77	286
GCP Cloud Run	1cpu/3mem	43000!	1020.6504	1214.02	193.37	296
Docker Baseline	1cpu/3mem	50000!	1548.6138	1554.91	6.3	6
Azure CI	1cpu/3mem	50000!	1654.675	1801.68	147.01	255

AWS ECS Fargate	1cpu/3mem	50000!	1393.7481	1575.01	181.27	283
GCP Cloud Run	1cpu/3mem	50000!	2019.2353	2207.04	187.81	296
Docker Baseline	1cpu/3mem	60000!	2712.317	2720.66	8.34	6
Azure CI	1cpu/3mem	60000!	2619.1221	2765.68	146.56	257
AWS ECS Fargate	1cpu/3mem	60000!	2243.3906	2429.24	185.85	284
GCP Cloud Run	1cpu/3mem	60000!	3801.8768	4018.46	216.58	296
Docker Baseline	1cpu/3mem	hello world	0.0106	4.57	4.56	6
Azure CI	1cpu/3mem	hello world	0.0007	98.47	98.47	258
AWS ECS Fargate	1cpu/3mem	hello world	0.0018	128.14	128.14	275
GCP Cloud Run	1cpu/3mem	hello world	0.0007	165.73	165.73	296
Docker Baseline	1cpu/4mem	10!	0.0168	4.07	4.06	6
Azure CI	1cpu/4mem	10!	0.0069	90.1	90.09	254
GCP Cloud Run	1cpu/4mem	10!	0.0056	123.18	123.17	296
AWS ECS Fargate	1cpu/4mem	10!	0.0061	130.61	130.61	280
Docker Baseline	1cpu/4mem	1000!	0.553	4.9	4.34	6
Azure CI	1cpu/4mem	1000!	0.529	90.76	90.23	256
GCP Cloud Run	1cpu/4mem	1000!	0.5006	126.67	126.17	296
AWS ECS Fargate	1cpu/4mem	1000!	0.4159	126.93	126.52	281
Docker Baseline	1cpu/4mem	10000!	41.6895	47.57	5.88	6
Azure CI	1cpu/4mem	10000!	45.9349	153.89	107.95	258
AWS ECS Fargate	1cpu/4mem	10000!	36.2225	169.01	132.78	282
GCP Cloud Run	1cpu/4mem	10000!	64.8765	208	143.13	296
Docker Baseline	1cpu/4mem	32000!	536.5177	542.92	6.41	6
Azure CI	1cpu/4mem	32000!	581.9081	742.42	160.51	254
GCP Cloud Run	1cpu/4mem	32000!	754.5965	955.48	200.88	298
AWS ECS Fargate	1cpu/4mem	32000!	503.6494	706.07	202.42	282
Docker Baseline	1cpu/4mem	43000!	1025.7688	1032.2	6.43	6
Azure CI	1cpu/4mem	43000!	1114.7917	1275.02	160.23	254
AWS ECS Fargate	1cpu/4mem	43000!	900.7542	1101.99	201.24	282
GCP Cloud Run	1cpu/4mem	43000!	1020.816	1232.62	211.81	296
Docker Baseline	1cpu/4mem	50000!	1562.7032	1569.61	6.91	6

Azure CI	1cpu/4mem	50000!	1638.0199	1797.87	159.85	258
GCP Cloud Run	1cpu/4mem	50000!	1990.4462	2185.57	195.12	298
AWS ECS Fargate	1cpu/4mem	50000!	1368.2974	1565.15	196.85	283
Docker Baseline	1cpu/4mem	60000!	2681.7906	2689.4	7.61	6
Azure CI	1cpu/4mem	60000!	2582.8495	2748.74	165.89	254
AWS ECS Fargate	1cpu/4mem	60000!	2197.8231	2395.24	197.41	283
GCP Cloud Run	1cpu/4mem	60000!	3651.4573	6563.23	2911.77	298
Docker Baseline	1cpu/4mem	hello world	0.0122	3.98	3.96	6
Azure CI	1cpu/4mem	hello world	0.0009	93.63	93.63	253
AWS ECS Fargate	1cpu/4mem	hello world	0.0014	128.66	128.65	276
GCP Cloud Run	1cpu/4mem	hello world	0.0006	165.61	165.61	296
Docker Baseline	2cpu/4mem	10!	0.0179	4.22	4.21	6
Azure CI	2cpu/4mem	10!	0.0069	89.66	89.65	253
GCP Cloud Run	2cpu/4mem	10!	0.0058	119.53	119.53	299
AWS ECS Fargate	2cpu/4mem	10!	0.0061	128.95	128.94	276
Docker Baseline	2cpu/4mem	1000!	0.5911	5.19	4.6	6
Azure CI	2cpu/4mem	1000!	0.5221	90.11	89.59	253
GCP Cloud Run	2cpu/4mem	1000!	0.5095	122.87	122.36	299
AWS ECS Fargate	2cpu/4mem	1000!	0.4165	132.41	131.99	276
Docker Baseline	2cpu/4mem	10000!	41.6221	47.24	5.62	6
Azure CI	2cpu/4mem	10000!	44.772	144.26	99.49	257
AWS ECS Fargate	2cpu/4mem	10000!	36.2375	166.86	130.63	278
GCP Cloud Run	2cpu/4mem	10000!	64.3443	200.55	136.2	297
Docker Baseline	2cpu/4mem	32000!	522.7921	528.29	5.5	6
Azure CI	2cpu/4mem	32000!	562.6312	725.7	163.07	253
GCP Cloud Run	2cpu/4mem	32000!	740.2417	916.52	176.28	295
AWS ECS Fargate	2cpu/4mem	32000!	504.1328	710.9	206.77	279
Docker Baseline	2cpu/4mem	43000!	1033.3186	1039.44	6.12	6
Azure CI	2cpu/4mem	43000!	1084.7159	1247.9	163.19	259
GCP Cloud Run	2cpu/4mem	43000!	1006.0047	1196.54	190.53	299
AWS ECS Fargate	2cpu/4mem	43000!	897.3943	1101.98	204.58	280

Docker Baseline	2cpu/4mem	50000!	1451.7842	1457.8	6.01	6
Azure CI	2cpu/4mem	50000!	1591.3697	1756.42	165.05	253
GCP Cloud Run	2cpu/4mem	50000!	1964.6281	2158.5	193.87	295
AWS ECS Fargate	2cpu/4mem	50000!	1365.1754	1568.68	203.51	281
Docker Baseline	2cpu/4mem	60000!	2606.8713	2613.73	6.86	6
Azure CI	2cpu/4mem	60000!	2493.9064	2670.34	176.43	255
AWS ECS Fargate	2cpu/4mem	60000!	2184.5987	2383.88	199.28	281
GCP Cloud Run	2cpu/4mem	60000!	3595.1682	3823.57	228.4	295
Docker Baseline	2cpu/4mem	hello world	0.0114	4.53	4.52	6
Azure CI	2cpu/4mem	hello world	0.0009	95.27	95.27	252
AWS ECS Fargate	2cpu/4mem	hello world	0.0014	131.19	131.19	271
GCP Cloud Run	2cpu/4mem	hello world	0.0007	161.64	161.64	301

Table 25: Internal CaaS processing times. In test column, the number followed by exclamation mark (!) means the calculation of the factorial for that number. AET stands for the average time for the Application Execution Time in milliseconds (ms). TTFB stands for average of Time to First Byte or average HTTP Request Waiting Time, also in milliseconds (ms). SCITP stands for Serverless CaaS Internal Processing Time, which is calculated based on the difference from TTFB and AET. The rows highlighted in grey correspond to the fastest times for that combination and test, ignoring the baseline.

R Terraform File to provision Azure Cloud Instances

```
provider "azurerm" {
  subscription_id = var.subscription
  tenant_id       = var.tenant_id
  features {}
  version = "2.28.0"
}

resource "azurerm_resource_group" "example" {
  name     = var.resource_group
  location = var.location
}

resource "azurerm_container_group" "example" {
  name                = "msc-caas-container"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  ip_address_type     = "public"
  os_type             = "linux"
  dns_name_label      = var.dns

  container {
    name   = "msc-caas-comparison-container"
    image  = var.docker_image
    cpu    = var.cpu
    memory = var.memory
    ports {
      port     = 80
      protocol = "TCP"
    }
  }

  tags = {
    environment = "testing"
  }
}

output "service_url" {
  value       = azurerm_container_group.example.fqdn
  description = "URL to access the app"
}
```

S Terraform File to provision AWS ECS Fargate

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

# Configure the AWS Provider
provider "aws" {
  region     = var.region
  access_key = var.access_key
  secret_key = var.secret_key
  token      = var.token
}



resource "aws_default_vpc" "vpc" {
  tags = {
    Name = "Default VPC"
  }
}

resource "aws_default_subnet" "default_az1" {
  availability_zone = "${var.region}a"
}

resource "aws_security_group" "app_security_group" {
  name   = "security-group-caas"
  vpc_id = aws_default_vpc.vpc.id

  ingress {
    protocol   = "tcp"
    from_port = 80
    to_port    = 80

    cidr_blocks = [
      "0.0.0.0/0",
    ]
  }
  egress {
    from_port    = 0
    to_port      = 65535
    protocol     = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_ecs_cluster" "cluster" {
  name = "msc-caas"
}

resource "aws_ecs_task_definition" "task_def" {
  family                   = "msc-caas-family"
  requires_compatibilities = ["FARGATE"]
  cpu = var.cpu
  memory = var.memory
  network_mode = "awsvpc"
  container_definitions = <<DEFINITION
[{
    "essential": true,
    "image": "viniciusbarros/msc-performance-web-app:latest",
    "name": "msc-caas",
    "portMappings": [{
        "containerPort": 80,
        "hostPort": 80
    }]
}]
DEFINITION
}


resource "aws_ecs_service" "service" {
  name            = "msc-caas"
  cluster         = aws_ecs_cluster.cluster.id
  desired_count = 1

  task_definition = aws_ecs_task_definition.task_def.arn
  launch_type = "FARGATE"
  network_configuration {
      assign_public_ip = true
      security_groups = [aws_security_group.app_security_group.id]
      subnets = [aws_default_subnet.default_az1.id]
  }
#   DAEMON not suported for FARGATE launch types :(
  scheduling_strategy = "REPLICA"
```

T Terraform File to provision GCP Cloud Run

```
terraform {
  required_providers {
    google = {
      source = "hashicorp/google"
    }
  }
}

provider "google" {
  version = "3.39.0"

  credentials = file("key.json")

  project = var.project
  region  = var.region
  zone    = var.zone
}

resource "google_cloud_run_service" "default" {
  name     = "msc-performance-web-app-tf"
  location = var.region

  traffic {
    percent         = 100
    latest_revision = true
  }

  template {
    spec {
      containers {
        image = "gcr.io/vmbarross/msc-performance-web-app"
        ports {
          container_port = 80
        }

        resources {
          limits = {
            cpu    = var.cpu
            memory = var.memory
          }
        }
      }
      container_concurrency = 0
      timeout_seconds = 900
    }
  }
}
# Allowing unauthenticated requests
data "google_iam_policy" "noauth" {
  binding {
    role = "roles/run.invoker"
    members = [
      "allUsers",
    ]
  }
}


resource "google_cloud_run_service_iam_policy" "noauth" {
  location    = google_cloud_run_service.default.location
  project     = google_cloud_run_service.default.project
  service     = google_cloud_run_service.default.name

  policy_data = data.google_iam_policy.noauth.policy_data
}

output "service_url" {
  value       = google_cloud_run_service.default.status
  description = "URL to access the app"
}
```

U Terraform Variables File to provision Azure Cloud Instances

```
variable "subscription" {
  description = "Azure Subscription ID"
  default     = ""
}

variable "tenant_id" {
  description = "Azure Tenant ID"
  default     = ""
}

variable "resource_group" {
  description = "Azure Tenant ID"
  default     = ""
}

variable "dns" {
  description = "The dns to be created"
  default     = "msc-caas"
}

variable "location" {
  description = "The Azure location where all resources in this
↪  example should be created"
  default     = "East US"
}

```

```
variable docker_image {
    description = "Docker image to be deployed"
    default = "viniciusbarros/msc-performance-web-app:latest"
}

variable cpu {
    description = "Amount of CPU to be used"
    default = "1"
}

variable memory  {
    description = "Amount of RAM to be used"
    default = "2"
}
```

V Terraform Variables File to provision AWS ECS Fargate

```
variable region {
    description = "AWS Region to be used"
    default = "us-east-1"
}

variable access_key {
    description = "Access key to grant access to resources"
    default = ""
}

variable secret_key {
    description = "Access secret key to grant access to resources"
    default = ""
}
```

```

variable token {
    description = "token for MFA"
    default = ""
}

variable cpu {
    description = "Amount of CPU to be used"
    default = 512
}

variable memory  {
    description = "Amount of RAM to be used"
    default = 1024
}

variable docker_image {
    description = "Docker image to be deployed"
    default = "viniciusbarros/msc-performance-web-app:latest"
}
```

W Terraform Variables File to provision GCP Cloud Run

```
variable "project" {
  description = "Project Name"
  default     = "vmbarross"
}

variable "region" {
  description = "Region to be used"
  default     = "us-east4"
}

variable "zone" {
  description = "Zone within region to deploy"
  default     = "us-east4-a"
}

variable cpu {
    description = "Amount of CPU to be used - k8s Style"
    default = "1000m"
}

variable memory  {
    description = "Amount of RAM to be used - k8s Style"
    default = "256Mi"
}
```