Instituto Politécnico de Tomar

Escola Superior de Tecnologia de Tomar

Diogo Abreu Ribeiro

An Investigation of Approaches to Mitigating Against Cold Start in Serverless Platforms

Projecto

Orientado por:

Luís Oliveira – IPT

Projecto apresentado ao Instituto Politécnico de Tomar para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Informática – Internet das Coisas

For my grandfather who unfortunately passed away during the writing of this document – I'll be aiming even higher, but if I become half the man you were, I'll be ok.

Acknowledgements

I would like to start by thanking my supervisor Luís Oliveira, for putting up with me in the bad moments and for always working to make the good moments even better. It’s always pleasurable to work with people that are constantly seeking new learnings, and I’m looking forward to doing it again in the future.

I’d like to thank all the colleagues that had this journey with me, especially my good friend Hélder Lopes with whom I shared pretty much all projects during the last 2 years. Being challenged by the brightest minds is what makes this journey interesting, and I wouldn’t have it any other way.

I’d like to thank the polytechnic institute of Tomar, and in particular vice-president Nuno Madeira, for working every day to make this an institution where studying became a very pleasurable but also challenging experience.

I’d like to thank all the teachers that helped me get here, even if I only had one class with them. A special thanks to Renato Panda for providing the best environment I have ever seen inside any classroom while challenging us to keep levelling up.

Last but not least, I’d like to thank all my family and friends who provide me every day with the life that allows me to keep going when things get rough.

I wouldn’t be here without the support from everyone that shared this journey with me, so, to EVERYONE, a big **THANK YOU!**

Abstract

This project presents the investigation and results found in the topic of ways to mitigate against cold start in serverless platforms, AWS (Amazon Web Services) being the focus of the investigation.

The investigation started with the study of the current scientific literature in order to contextualize the problem and present hypothesis to study. From there, test scenarios were developed, resorting to the open source serverless framework, by deploying several functions with different characteristics and gathering results to analyse, regarding cold start times.

The results showed that the cold start problem can and should be optimized from the side of the client, not only for the sake of performance, but also in order to reduce costs.

Keywords: Serverless, AWS, FaaS, Cold Start, Lambda

Table of Contents

Table of Figures

Table of Tables

1 Introduction

1.1 Context and Motivation

Serverless (or FaaS) architectures are a fairly recent approach to cloud native applications and, as such, with all their benefits, new challenges also arise. Some examples of serverless offerings are AWS Lambda, Google Cloud Functions, and Azure Functions. When a serverless function is executed, there isn't always an instance (VM, container, …) ready to execute the code, and, therefore, a new execution environment must be provisioned. The delay between provisioning the new environment and actually starting the execution of the code, is called cold start.

Although for certain applications this delay isn't critical, for others it is unacceptable, therefore, this project is developed with the idea to study what factors affect cold start, and to provide developers with a set of good practices or suggestions that will allow their functions to become more efficient.

1.2 Objectives

The objective of this project is to develop test scenarios where functions with different characteristics will be evaluated in order to determine which factors will be relevant in lengthening or shortening the cold start times. By the end, some good practices should be available for both cost and performance.

1.3 Contributions

This project presents the investigation and its results in a basic manner, that allows the results to be relevant but also gives the opportunity for anyone not in touch with serverless to learn the basis of the architecture.

From the results, a github link is provided, making available for everyone the dataset with the results of the test scenarios, allowing anyone interested to make an analysis and reach their own conclusions, either verifying or maybe even denying the ones presented in this project.

2 Cloud Computing

Cloud platforms have risen in popularity to deliver computing resources that are virtualized and dynamically scalable, as fast as possible [1]. Although there are several definitions of what cloud computing is, based on the NIST definition, we define cloud computing as a model that enables the fast and easy provisioning of a shared pool of computing resources (e.g., networks, servers, storage, applications, and services) from anywhere, no matter what the demand is. This definition ends up summarized in the main characteristics for this model, which are:

- On-demand self-service - A consumer of such services can provision the needed computing resources without the need of any human assistance from the side of the service provider.
- Broad Network Access - These services are available over the network and can be accessed by virtually all devices/clients.
- Resource Pooling - Computing resources owned by the service provider are managed to serve multiple consumers according to general demand. Usually, the consumer has no control over the location of said computing resources, although, there can be the possibility to choose certain characteristics in a higher layer of abstraction (e.g., country where said resources are located).
- Rapid elasticity - The resources provided should be able to scale according to the demand (both manually and automatically), presenting seemingly no limit.
- Measured Service - Resource usage is available for both consumer and provider.

2.1 Cloud Deployment Models

One of the most important parts of a cloud computing deployment comes before the deployment itself, in the planning phase. Institutions must define their objective, in order to choose the correct type of implementation, allowing them to search for the correct set of skills (developers) and hardware (machines) [2]. This does not stop institutions from changing their deployment model as business needs evolve, but it is important to have an objective for the deployment to be successful.

The most common deployment models are:

- Private Cloud - The infrastructure exists to serve a single company. It does not have to be necessarily hosted in the premises of that same company, but it must be isolated in a private network and use software and hardware that are intended to be exclusively used by that company. A private cloud cannot be accessed by the public, making it arguably the most secure deployment model, as only a defined number of users can access the information [3].
- Community Cloud - Almost the same as a private cloud, with one key difference, the set of users. Instead of serving one organization the infrastructure serves multiple organizations with similar interests and requirements [3].
- Public Cloud - As the name indicates, public clouds are available to the public through the resources of cloud providers. These resources are available through a set of services which the providers usually make available at a pay-per-use basis. Although a public cloud provides an easy and ready for use product, where a customer (individual or a company) does not need to worry about maintaining the

infrastructure, it also comes at the cost of having no control over factors like reliability, security, and privacy [3].

- Hybrid Cloud - It's nothing more than a mix of the previous models (private, community, public). The individual models remain as their own but are linked through technologies that allow an institution to draw the best out of each model.

2.2 Cloud Service Models

The cloud service models are the basis that a provider uses to organize its products. It also works as a basis for the consumers, who know what to look for according to their needs. It is how you split the entire architecture of anything IT related, and you sell it (almost) individually. Nowadays, it looks like more and more these frontiers are being broken as everything is starting to turn into a service (figures 1 and 2).

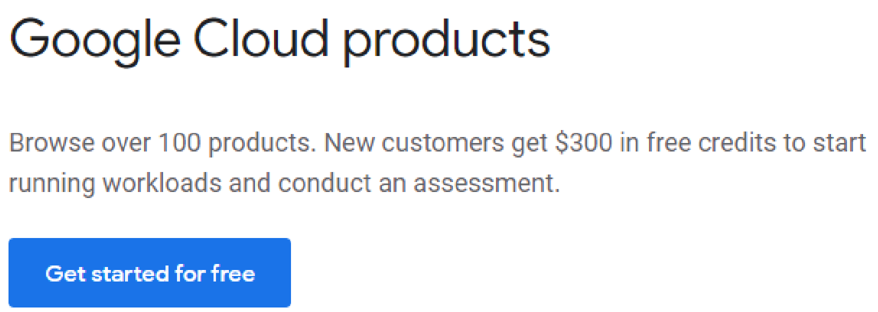


Figure 1 - Google Cloud Platform products catalog showing over 100 different products - https://cloud.google.com/products

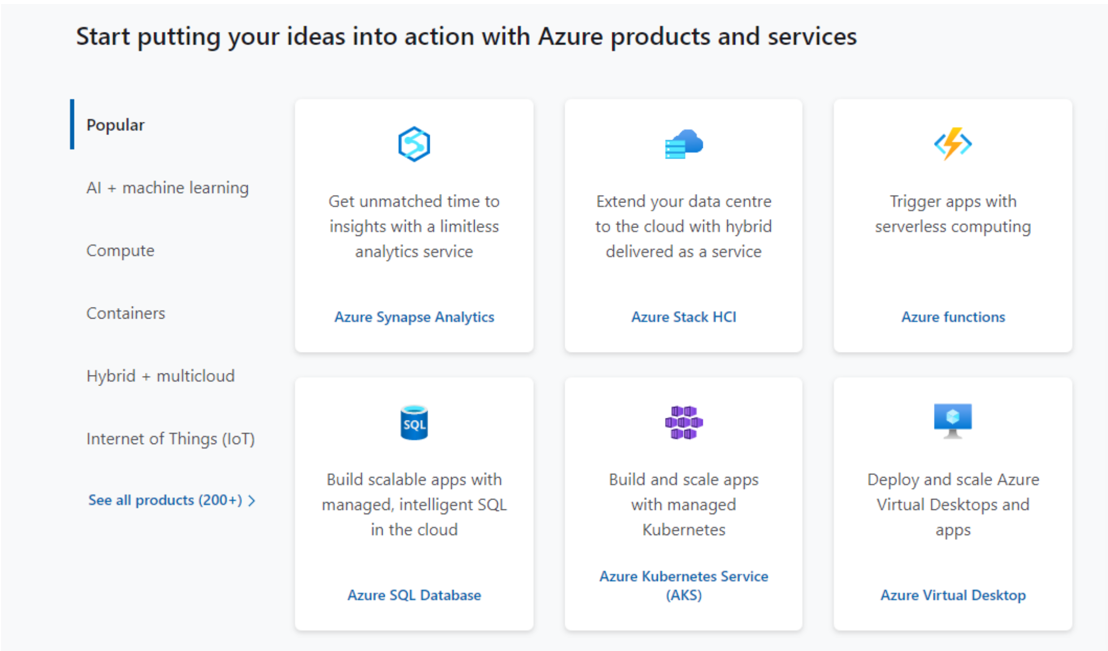


Figure 2 - Microsoft Azure products menu showing over 200 different services - https://azure.microsoft.com/en-us/services/

With this in mind, the three original service models are still the best way to describe what a cloud product actually is. Providers continue to find more and more services but, in the end, they all end up being a sort of subtype of the three main service models, which are(figure 3):

- Infrastructure as a Service (IaaS) - The consumer receives the main hardware resources required for the most common computing activities (e.g., CPU, storage, networks), on top of which it is possible to deploy and run the intended software (this ranges from simple applications to an actual OS). Usually, these resources are delivered as a virtualization platform, and are accessed through the internet. In short, the consumer receives a working infrastructure (working hardware with storage access, network access and others) and can build on top of it [4] [5].
- Platform as a Service (PaaS) - The consumer gets the ability to deploy applications to the cloud infrastructure, without the need to worry about the underlying conditions (such as hardware, type of OS, required software versions, …). The consumer gets control over the deployed application and certain features of the environment on which it runs. The goal is usually to deliver a set of tools that facilitate processes such as app design, app development, app testing, app deployment and app monitoring while making use of the hosting provided by the cloud [1].
- Software as a Service (SaaS) - The consumer gets access to applications running in the provider's cloud infrastructure, usually through the browser. A good example of SaaS is office365 or google apps (google docs, google sheets, …). Software like Microsoft Office were traditionally installed on a personal computer, nowadays, all you need is a subscription and internet access, and you will be able to do just the same using the browser only. In this model the consumer only has one single worry, which is to use the software they acquired. The rest (maintaining the servers, updating, and patching the software, …) is the providers responsibility [5].

Figure 3 - The three main cloud service models and which areas are controlled by the provider/client

As said before, more and more parts of a traditional application are turning into services, which can be deployed individually. One of these services is Function as a service (FaaS), which will be one of the central points of discussion in the rest of this project, so to start, in short, FaaS is a service model that allows for the execution of code in response to events without the typical underlying complex infrastructure [6] and is arguably the most important part of the serverless model.

3 Serverless

At the time of writing this document, there seemed to be no consensus as to whether Serverless and FaaS should be considered synonymous or not. It seems that the two main arguments can be summarized in - Serverless and FaaS are synonymous - and - FaaS is a subset of serverless. For the sake of the reader, lets preface this with saying that, in the case of this document, only the term serverless will be used and it should be considered as a synonym.

The term “Serverless” refers to a new evolution of PaaS offerings, where rather than deploying and running traditional services, or dedicated VM’s, users can now deploy individual functions and pay only for the time that their code is being executed [7], without having any control over the resources in which said code is being executed. The name “serverless” can give the wrong idea that there are no physical servers in play, but the truth is the servers are there, what changes is that, now, the developers can leave all operational concerns, to the cloud providers. Although the abstraction from server management can make it sound like serverless is the same as a traditional PaaS offering, the two models differ, starting from the fact that, in serverless, the user has zero costs associated with the time that their code is not running, while in traditional PaaS offerings, that time is still billed [8].

Figure 4 – Same as figure 3, with the addition of FaaS to show its position in the services structure

Serverless offerings provide a programming model based on stateless functions, where, unlike in PaaS offerings, developers can write code without the limitation of a pre-packaged application, giving also birth to the name FaaS [8] (figure 4).

The emergence of serverless computing as a new compelling paradigm for the deployment of applications and services, represents the evolution and maturity of cloud technologies. It is not something completely new, it is however being brought up on a different scale thanks to the evolution of technologies such as containers and virtual machines. Based on google trends (Figure 5), we can attest to some degree that cloud computing had its greatest boom around 2010/2011, and in recent years, as it stabilizes as a standard, terms like serverless started gaining traction.

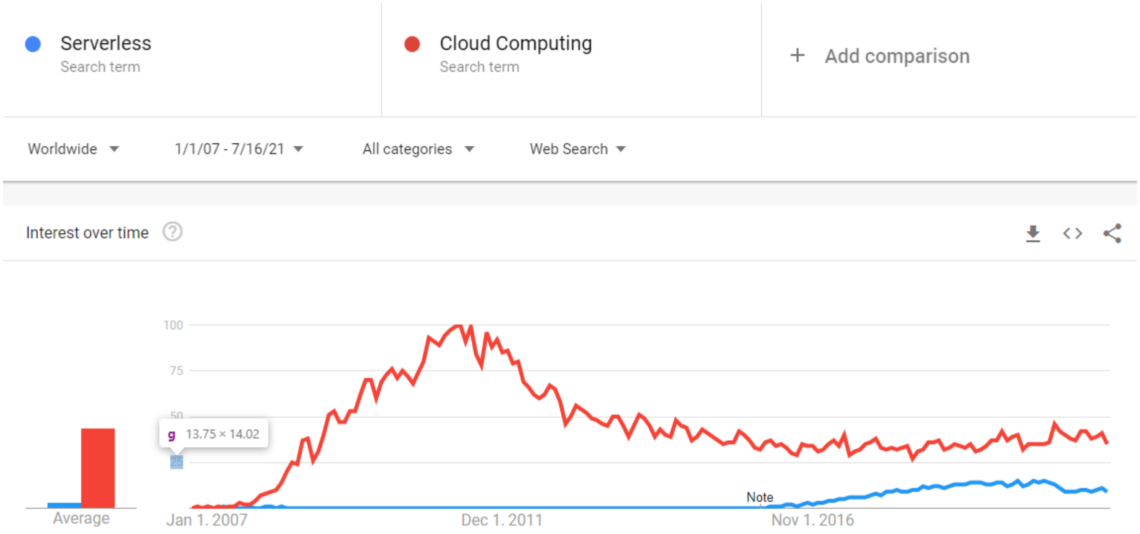


Figure 5 - Interest over time between the terms "Serverless" and "Cloud Computing", according to google trends

In traditional architectures, in order to run their applications and services, developers and admins had to acquire dedicated machines which usually stayed on premises. The cost was high throughout the entire process, starting from the point of acquiring a new machine and ending at the point where that machine is being fully managed and utilized. If an expansion was needed, the process would repeat, adding to the costs not only in monetary terms, but also in terms of the time spent planning for the activities described above. One of the most worrisome trends is the fact that the infrastructure was the same, during both average and peak usage times. With the rise of cloud computing, some of these problems were mitigated, but teams were still needed to plan, develop and maintain certain key areas of cloud services, such as virtual machines. Serverless appears as a new paradigm where the only task to be executed by developers is the actual development of applications. Every single task aside from that, going from managing and monitoring the infrastructure state to scaling and adapting where necessary, becomes a responsibility of the cloud provider [7]. In serverless, at this time, code is written with a focus on event handling, which translates into not paying for a machine that is constantly running. Billing is usually only in effect from the moment that events are triggered, and the function associated starts running. Customers are now billed based on the time that their code is actually being executed, not stand-by times, which means that an application's idle time is not charged. [7]

Serverless brings several benefits for the consumers, providing them with a simplified programming model, removing operational responsibility and reducing costs by billing only execution time and not the traditional fully allocated resources. But it is also beneficial for the cloud provider, bringing the opportunity to control the entire software stack. This also enables costs reduction if the provider can optimize its resources in the most efficient way, and encourages users to stay in the ecosystem, since cloud functions are usually easily integrated with other cloud services [8]. Although the benefits are substantial, serverless does not seem to be even near its full potential, which means that, with its continuous adoption, this model will keep evolving in the foreseeable future [9].

3.1 Serverless Architecture

The Serverless model follows an architecture pattern based on the response to events. This promotes the decoupling of services and allows for the certainty that the code only runs when intended. As a result, developers are challenged to learn how their applications can be built in this model, which has clear differences to the traditional approach. More complex applications may not be possible to be directly migrated to a Serverless model, this may come both from the limitations caused from the infancy of these services, but also from the differences between paradigms [10]. Despite that, more and more common applications of serverless are emerging, some examples are:

- API requests
- Updates to databases
- Voice commands (virtual assistants like Siri [11] and Alexa [12])
- Control of IOT devices, such as smart bulbs in your house

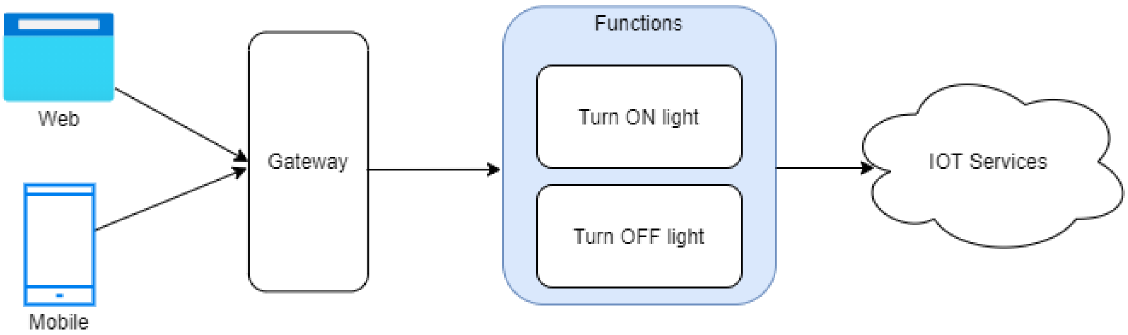


Figure 6 - Example of a basic serverless structure to manage IOT devices

To make a simple example, let's look at figure 6, in this simple example, a request can be sent from a browser or mobile app into the designed gateway. Depending on the chosen platform, this gateway can change, as each platform has different services and different interactions between each of them, but for this explanation, let's think of the gateway as a method you would call through a browser, like a simple request to an API. This gateway will be prepared to understand the request, and act according to it. So, if a request would be sent with the method to turn on the light in a room, that event would have been pre-established to trigger the activation of the designed cloud function, which in turn would connect to the services that would turn on the light.

A serverless platform's basic task is the handling of events. The services must know how to receive and handle events sent from a defined source, determine which function(s) should be dispatched in response, determine if an existing instance can run the function or if a new instance must be created, start the function according to the request, wait for a response, gather the defined logs necessary, deliver the response to the requester, and finally, stop the execution of the function once it is no longer needed [8] (example in figure 7).

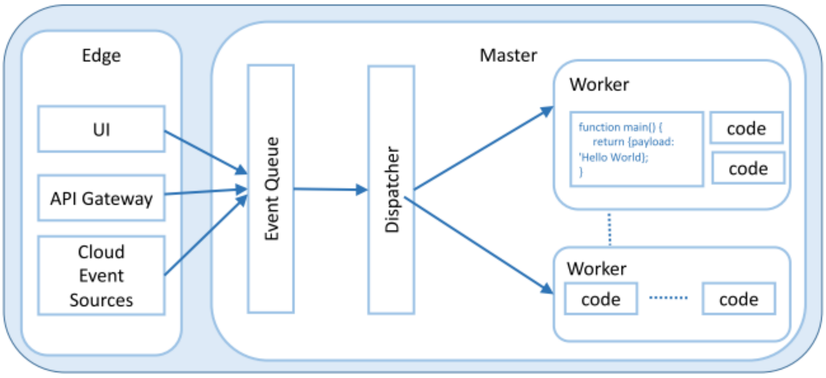


Figure 7 - Example of a serverless architecture – Figure from [8]

The real challenge is to implement such services, while maintaining the standards for cost, scalability, availability, and fault tolerance. The providers need to be able to start a function as soon as possible in the most efficient way, while accounting for the load received and the functions that have finished running, guaranteeing the service regardless of the load [8].

3.2 Serverless Limitations

Since still in its infancy, like all technologies, Serverless has limitations. As a developer, it is necessary to evaluate which projects are adequate for serverless computing, and how to implement them right. Also, it is necessary to consider such limitations, and plan ahead of time how to handle them. Some of the most relevant limitations of the Serverless model are [10]:

- The client resigns control over their infrastructure [10].
- Cost is a challenge for both the provider, who must minimize resource usage for serverless functions, and for the consumer, which has to evaluate which model is adequate for the intended application, serverless or traditional models [8].
- Cold start, which is inevitable when a model has the ability to scale to zero, which means that it doesn't charge customers for idle time [8].
- Resource management, where the provider must guarantee that, independently of the load, the infrastructure can maintain its standards of security, performance, and availability. It may also be a challenge to the developer, since some of the providers now offer the chance to manually set the values needed for certain resources [8].
- Security, which is a big concern considering that several functions of several different users are running on the same platform. Isolation is necessary between functions, especially considering the regulations that are in place regarding user data confidentiality, and also as a measure to avoid functions affecting each other, potentially corrupting both codes [8].
- Adding to the previous point, scalability is also a big concern as providers must guarantee resources for user's functions and future loads, which is made difficult

since the planning of resources is made with little to no information regarding the content of said functions and the applications they support [8].

- Knowledge over available services is a double-edged sword. For Serverless to maintain complex applications, it usually works with other services from the same provider (which creates another possible problem with vendor lock-in), but the range of services has gotten so large that it is easy for a developer to get overwhelmed by the sheer number of options. Also, while highly unlikely, it is possible that said cloud provider does not have a service that can fulfil the client's requirement. This also makes it difficult to make the direct migration of an application from traditional services, into Serverless [8].
- Monitoring can become a challenge, as the only information available is what is provided by the serverless platform [8].

3.3 Serverless Cloud providers

All the main cloud providers have throughout the years built their own serverless offering. Although there are several options, the highlight goes to arguably the three biggest public cloud platforms [13] [14], Microsoft Azure from Microsoft, Google Cloud platform from Google, and Amazon Web Services from Amazon. Microsoft Azure's Serverless offering has the name Azure functions, Google Cloud Platform offers Cloud functions, and finally, Amazon's AWS offers AWS Lambda, probably the most mature of all services. The following table 1 [15] compares some information regarding these services:

	AWS Lambda [15] [16] [17] [18] [19] [20]	Azure Functions [15] [21] [22] [23] [24] [20]	GCP Cloud Functions [15] [25] [26] [27] [28] [29] [20]
Free Monthly Duration (GB-seconds)	400,000	400,000	400,000
Free Monthly Requests	1 Million	1 Million	2 Million
Cost of Each Additional 1 Million Requests	$0.20	$0.20	$0.40
Cost of Each Additional 1 GB-second	$0.0000166667	$0.000016	$0.0000025 (tier 1) $0.0000035 (tier 2)
Duration is Rounded to the Nearest	1ms	1ms	100ms
Supported Languages	C#, Go, Java, Node.js, PowerShell, Python, Ruby +Runtime API	C#, F#, Java, Node.js, PowerShell, Python, TypeScript	Node.js, Python, Go, Java, .NET, Ruby, PHP
Support for Custom Runtimes	Yes, using custom deployment packages or AWS Lambda Layers	Yes, using Azure Functions custom handlers	Yes, using custom Docker images

Average Cold Start (in seconds)	<1	<5	0.5<time<2
Memory	128 MB – 10240 MB	128 MB – 1500 MB (Consumption Plan) 128 MB – 14000 MB (Premium and Dedicated Plans)	128 MB – 8192 MB
Maximum Timeout	15 minutes	10 minutes (Consumption Plan default) Unbounded (Premium and Dedicated Plans default)	9 minutes
Orchestration Service	AWS Step Functions	Durable Azure Functions	GCP Workflows
HTTP Integration Support	Yes, with Amazon API Gateway	Yes, out of the box	Yes, out of the box

Table 1 - Comparison of key characteristics found in the most popular cloud providers

3.4 The cold start problem

Some key limitations have been briefly mentioned before, but the focus of this project is to study ways to mitigate against cold starts in AWS, therefore, it’s useful to delve into cold a start a bit more.

Starting from the beginning, one of the key features of serverless is the ability to pay only when code (or a function) is actually running. In traditional architectures, the whole program is running 24/7, which means, the environment is always ready to respond to the defined requests. Now, in serverless, the infrastructure must be ready to answer the same way, with the challenge that there is no previously defined “program” that knows what to expect, and the same machine will be used constantly by different users, with different objectives and dependencies. In summary, a cold start happens when there is no environment ready to respond to the requested function. This being said, a major part of cold start mitigation is restricted to the provider that controls the infrastructure, which makes sense, seeing as the whole premise for serveless is that the user does not have to worry about such things. Figure 8 shows where the client can do something about it, referring to bootstrapping the runtime, and it is in this area that this project will be focused, finding ways, from the client’s side, to mitigate against cold start where possible, knowing that only AWS can better the rest behind the scenes.

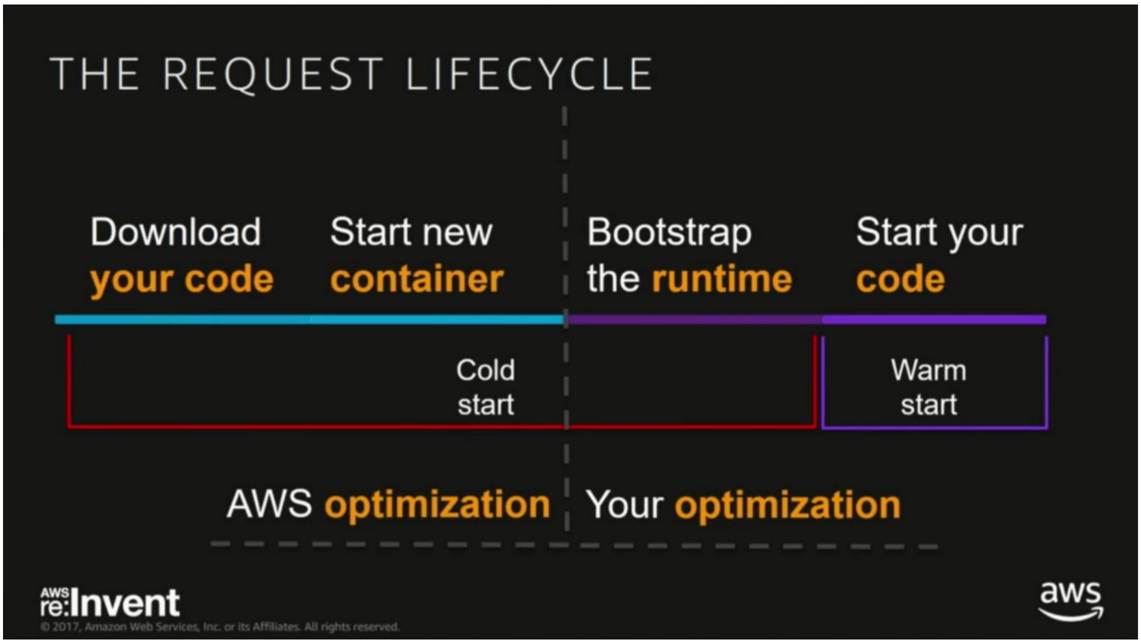


Figure 8 - from AWS re:Invent 2017: Become a Serverless Black Belt: Optimizing Your Serverless Appli (SRV401) - https://www.youtube.com/watch?v=oQFORsso2go&t=485s

4 Research Methodology

When it comes to evaluating what affects cold starts, the first challenge is to find a way to measure said cold starts. AWS makes this easy, by providing that measurement in the form of a metric found in the logs, by the name of "init duration" [30]. Figure 9 shows part of a log file that had several executions of a lambda function in AWS. It should be noted that, in the first execution, there is the metric "Init Duration", indicating that there was in fact a cold start in this execution. The following execution of the same function doesn't display this metric, as the container is pre-warmed (was provisioned previously and is still ready for new executions) by the previous execution and hasn't expired yet.


START RequestId: 386caa29-0052-4656-b053-741db9a3b0b5 Version: $LATEST
END RequestId: 386caa29-0052-4656-b053-741db9a3b0b5
REPORT RequestId: 386caa29-0052-4656-b053-741db9a3b0b5 Duration: 1.13 ms Billed Duration: 2 ms Memory Size: 1024 MB Max Memory Used: 36 MB
Init Duration: 109.59 ms

START RequestId: 954cdc3b-5b55-4041-a72e-362fc10d0861 Version: $LATEST
END RequestId: 954cdc3b-5b55-4041-a72e-362fc10d0861
REPORT RequestId: 954cdc3b-5b55-4041-a72e-362fc10d0861 Duration: 0.84 ms Billed Duration: 1 ms Memory Size: 1024 MB Max Memory Used: 37 MB

Figure 9 - Logs from 2 executions of a lambda function

This study will focus on deploying several lambda functions with different characteristics (such as different programming languages and memory sizes) and evaluating the "Init Duration" metric. For the deployment and management of these functions the serverless framework [31] will be used, helping in the process with pre-built templates [32] from its open source community.

4.1 Initial hypothesis

Before developing the scenarios for testing, a plan must be drawn as to what this investigation is looking for. Therefore, resorting to general knowledge and other studies ([33] [34]), the initial hypothesis and questions to be answered are:

- Python should have the lowest cold start time, while Java should have the highest
- It's expected that the memory size of the deployed container will affect cold start [35], but does it do so in a significant matter?
- Resorting to the last point, considering languages like Python are already so fast, assuming the memory does in fact affect cold start, is it expected that the slowest languages, like Java, will see more benefits than the fastest ones?
- After executing a function, the container should be available for a considerable amount of time before timing out. How long does it take for the container to time out? Does the frequency of the invocations affect how long the container lasts before timing out?
- Cold start times aren't supposed to be billed. Is this verifiable?

4.2 Test Scenarios developed

Once the expectations were defined through the initial hypothesis, several test scenarios were designed to investigate said hypothesis.

A few important notes about the scenarios developed, to justify certain assumptions and choices for testing:

- Most scenarios will have each function executed 100 times (unless stated otherwise). The number 100 was chosen as a number that was fair enough to give relevant results, but also wasn't so high that it would make the testing period too long.

- All tests will be executed with hello world functions mostly generated from templates found in the serverless framework repository. In order to make a fair comparison between programming languages, the code should be kept as similar as possible. With this said, as long as all programming languages are kept in this state, then the comparison should be valid, be it a full-fledged application or a simple hello world statement. Therefore, the code generated from the templates available, which are designed with the same objective despite the programming language used, should be valid for the comparison.

4.2.1 Scenario 1

The first scenario was designed to compare the average cold start time of several similar functions, with the only difference being the language chosen. To do this, a hello world function was deployed in 6 different programming languages:

- Python – using serverless template "aws-python"
- C-sharp – using serverless template "aws-csharp"
- NodeJs – using serverless template "aws-nodejs"
- Ruby – using serverless template "aws-ruby"
- Java-Maven – using serverless template "aws-java-maven"
- Java-Gradle – using serverless template "aws-java-gradle"

Gradle and Maven are build automation tools for Java [36] [37] [38].

These languages were chosen as they are some of the most popular programming languages in the world [39] [40].

All these functions share the same configuration, using the default 6 seconds timeout time and 512 MB of memory.

To build the samples from scenario 1 all functions were executed 100 times. Scenario 1 is expected to confirm the first hypothesis.

4.2.2 Scenario 2

The second scenario evaluates how the amount of allocated memory affects cold starts. To test this, several python functions were deployed. These functions shared the same code to be executed, but each one had a different amount of memory allocated. The amounts chosen to evaluate were:

- 128 MB
- 256 MB
- 512 MB
- 1024 MB
- 2048 MB
- 4096 MB

For the samples from scenario 2, all functions were executed 100 times. Scenario 2 should give a clear answer to the second hypothesis.

4.2.3 Scenario 3

Scenario 3 ends up being a mix of scenario 1 and scenario 2. Functions were deployed with all the languages found in scenario 1, using different memory sizes. The memory sizes were split in:

- Small – 128 MB
- Medium – 512 MB
- Large – 4096 MB

For better comprehension, table 2 displays all the functions deployed and the characteristics associated with this test scenario.

Name of the function (example)	Programming Language	Memory Size (MB)
Hello-world-python-128	Python	128 MB
Hello-world-python-512	Python	512 MB
Hello-world-python-4096	Python	4096 MB
Hello-world-csharp-128	C-Sharp	128 MB
Hello-world-csharp-512	C-Sharp	512 MB
Hello-world-csharp-4096	C-Sharp	4096 MB
Hello-world-nodejs-128	NodeJs	128 MB
Hello-world-nodejs-512	NodeJs	512 MB
Hello-world-nodejs-4096	NodeJs	4096 MB
Hello-world-ruby-128	Ruby	128 MB
Hello-world-ruby-512	Ruby	512 MB
Hello-world-ruby-4096	Ruby	4096 MB
Hello-world-javaGradle-128	Java-Gradle	128 MB
Hello-world-javaGradle-512	Java-Gradle	512 MB
Hello-world-javaGradle-4096	Java-Gradle	4096 MB
Hello-world-javaMaven-128	Java-Maven	128 MB
Hello-world-javaMaven-512	Java-Maven	512 MB
Hello-world-javaMaven-4096	Java-Maven	4096 MB

Table 2 - Functions Deployed in Scenario 3

The objective with scenario 3 is to compare how the different memory sizes affect each programming language, answering the third hypothesis.

4.2.4 Scenario 4

Scenario 4 is built to study the fourth hypothesis. To do this, a python hello world function is deployed with its default characteristics (memory: 1024 MB; timeout: 6 seconds), and executed for approximately 24 hours in two different tests:

- Test 1 – The function is executed every 5 minutes
- Test 2 – The function is executed every hour (60 minutes)

This should give a clear vision as to how long containers last on average and if the frequency of executions affects this duration time. Scenario 4 will generate logs from hundreds of executions and these logs can be used to study the fifth hypothesis as well.

4.3 Basic Setup

This section will display the basic setup, shared across all tests. It will show as an example the creation, deployment, execution, log analysis, and destruction of a "hello world" function. A more detailed, step-by-step explanation of this setup, and a "how to guide" can be found in appendix 1.

The steps visited in the following sections are:

- Installing and configuring the serverless framework to work with AWS
- Creating a function resorting to available templates
- Deploying and executing said function and getting logs from AWS
- Removing the deployed function from AWS

4.3.1 Installing and configuring the serverless framework to work with AWS

For starters the serverless framework must be installed (figure 10)

Figure 10 - Command to install the serverless framework, through npm

Next, for the serverless framework to access AWS, it needs credentials. A user by the name of "serverless-admin" was created with administrator access (figure 11).

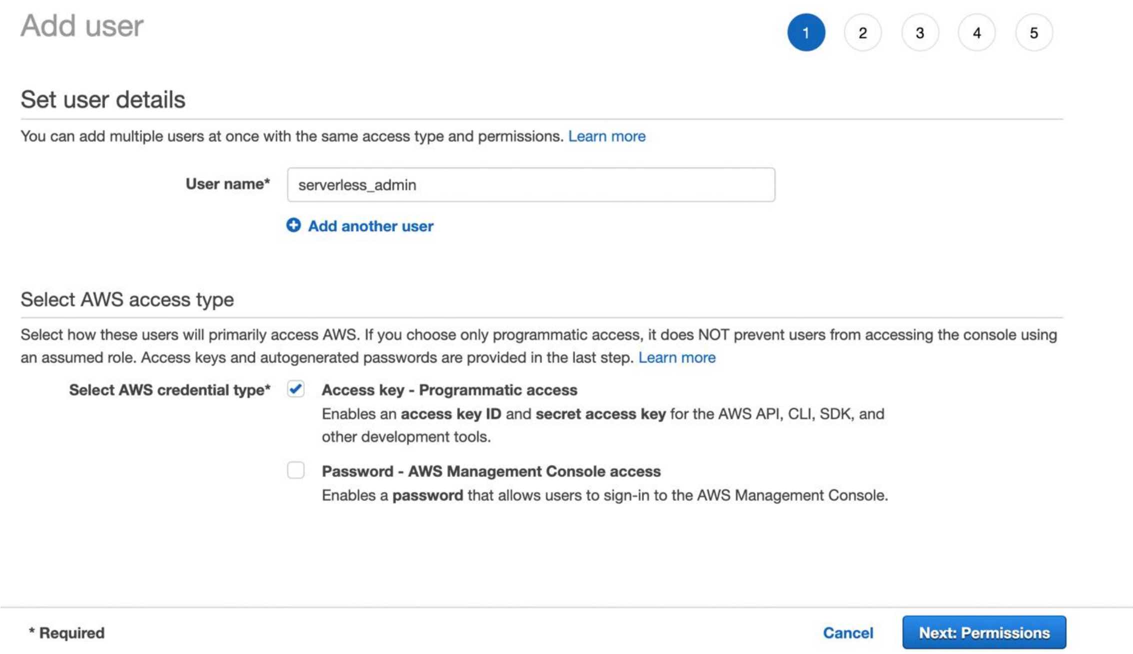


Figure 11 - Creation of a user in AWS

After creating the user, it just needs to be added to the serverless framework and the initial setup is completed (figure 12).

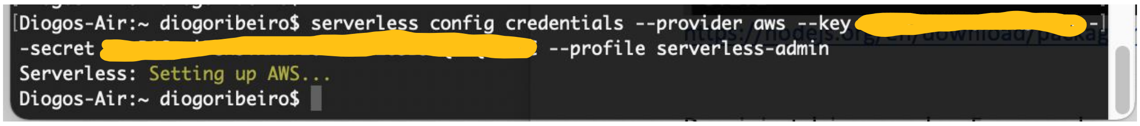


Figure 12 - Giving the serverless framework the created AWS credentials

4.3.2 Creating a function resorting to available templates

Once configured, the framework is ready to be used. To note that, to invoke commands from the serverless framework both the "serverless" command and its abbreviation "sls" can be used. In this case, a python function will be created, resorting to the "aws-python" template (figure 13).

Figure 13 - Creating a function through "aws-python" template

In this case, two files are created "handler.py" (figure 14), and "serverless.yml" (figure 15). Starting with handler.py, this file contains the code of the function which will be executed when a set event triggers it. In this case, the default contents from the template were replaced with a simple "print" and "return" statements.

Figure 14 - handler.py

Then, serverless.yml has the configuration of the service that will be configured in AWS, including the instruction to execute the function "hello", found in the previous handler.py file, when the defined trigger happens. In this case, the only needed changes from its default state were:

- Updating the runtime to Python 3.9 as the previously used python 2 is no longer supported
- Adding the profile "serverless-admin" (which matches the previously configured credentials)
- Adding the region, in this case being "eu-west-3" or "Paris", which is the closest available to Portugal

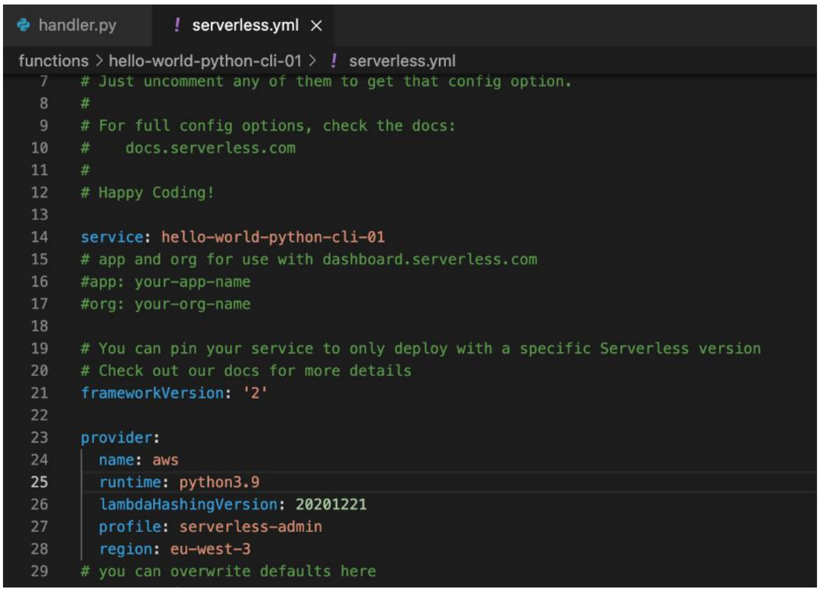


Figure 15 - Serverless.yml

After configuring these two files, the code is ready to be deployed, as seen in figure 16. It should be noted that aside from the lambda function, other services are created in AWS which are required for the correct behavior, such as the creation of a log group to record the information regarding the executions of the deployed function. It's an important information to have, but what's important to understand from this basic example is that the function is now fully deployed and ready to be executed.

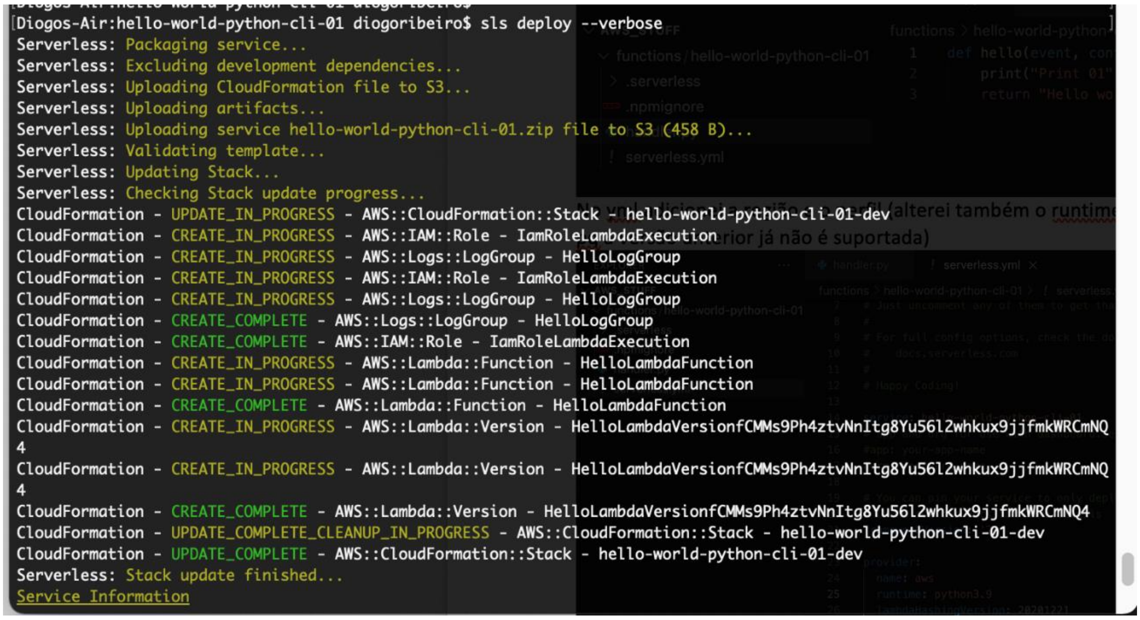
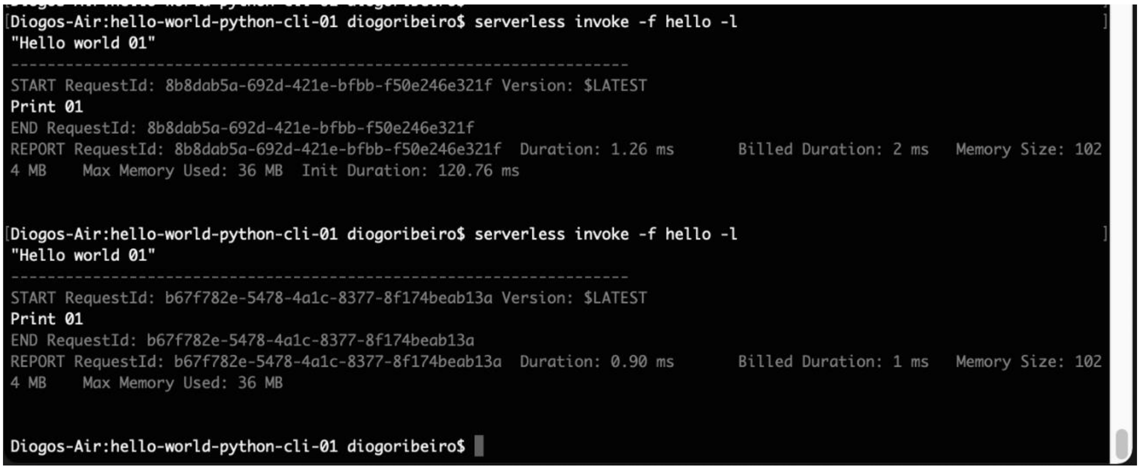


Figure 16 - Deploying the code to AWS

4.3.3 Deploying and executing said function and getting logs from AWS

At this point, the function "hello" is deployed in AWS and ready to be executed. To execute this function from the command line, the command "serverless invoke -f [FUNCTION_NAME]" is used, as seen in figure 17.



```
[Diogos-Air:hello-world-python-cli-01 diogoribeiro$ serverless invoke -f hello -l
"Hello world 01"
--------------------------------------------------------------------
START RequestId: 8b8dab5a-692d-421e-bfbb-f50e246e321f Version: $LATEST
Print 01
END RequestId: 8b8dab5a-692d-421e-bfbb-f50e246e321f
REPORT RequestId: 8b8dab5a-692d-421e-bfbb-f50e246e321f  Duration: 1.26 ms      Billed Duration: 2 ms   Memory Size: 102
4 MB    Max Memory Used: 36 MB  Init Duration: 120.76 ms

[Diogos-Air:hello-world-python-cli-01 diogoribeiro$ serverless invoke -f hello -l
"Hello world 01"
--------------------------------------------------------------------
START RequestId: b67f782e-5478-4a1c-8377-8f174beab13a Version: $LATEST
Print 01
END RequestId: b67f782e-5478-4a1c-8377-8f174beab13a
REPORT RequestId: b67f782e-5478-4a1c-8377-8f174beab13a  Duration: 0.90 ms      Billed Duration: 1 ms   Memory Size: 102
4 MB    Max Memory Used: 36 MB

Diogos-Air:hello-world-python-cli-01 diogoribeiro$
```

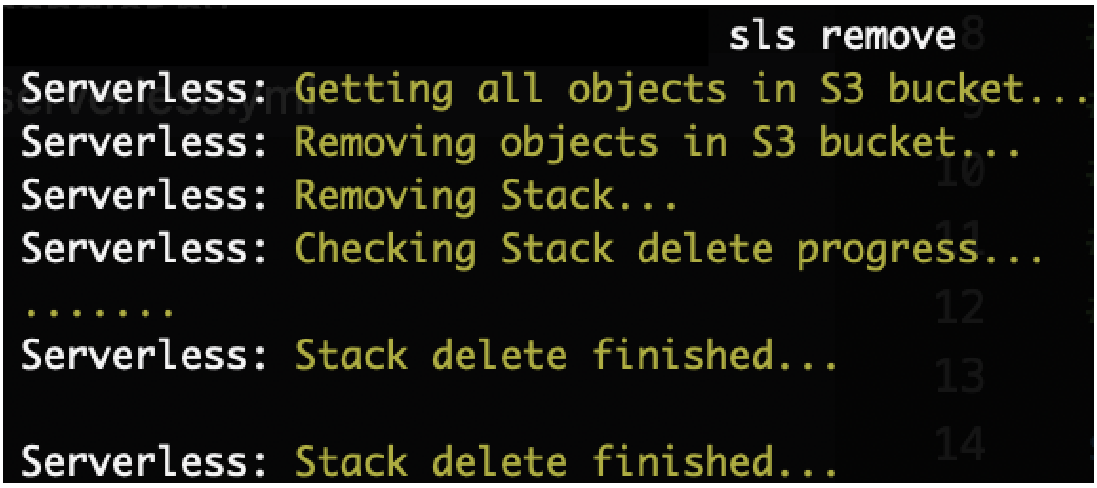
Figure 17 - Two calls to the defined function and the respective logs

Three key points are worth mentioning at this point:

- In this example the flag "-l" was added, indicating the logs should be displayed in real time with the execution. An alternative would be the same command, but instead of "invoke", "logs" would be used, resulting in "serverless logs -f hello"
- As mentioned in the beginning of the chapter, the first execution displays a metric by the name of "Init Duration", indicating the existence of a cold start of 120.76ms. The second execution doesn't display this metric, as the previous environment/container is still ready for execution (warm start)
- In this case, the event that triggered the execution of the lambda function, was the command sent from the CLI, but, in a real word scenario, this could be configured to any of the most common uses (API calls, changes to files or databases, receiving logs from an IOT device, …)

4.3.4 Removing the deployed function from AWS

Finally, to completely remove all deployed services, the command "serverless remove" is used (figure 18).



```
sls remove
Serverless: Getting all objects in S3 bucket...
Serverless: Removing objects in S3 bucket...
Serverless: Removing Stack...
Serverless: Checking Stack delete progress...
.......
Serverless: Stack delete finished...

Serverless: Stack delete finished...
```

Figure 18 - Deleting all services previously deployed to AWS

The previously created files will remain in the local directory, at which point they can be edited, re-deployed or simply deleted, as intended by the user.

5 Research Results

All the data used to calculate the results that will be presented can be found at: https://github.com/TheRealKeiriko/AWS_Serverless_ColdStart_Research

5.1 Scenario 1

After finishing the execution of scenario 1, the results were registered, and an average cold start time was calculated for each programming language. The results from scenario 1 are displayed in figure 19.

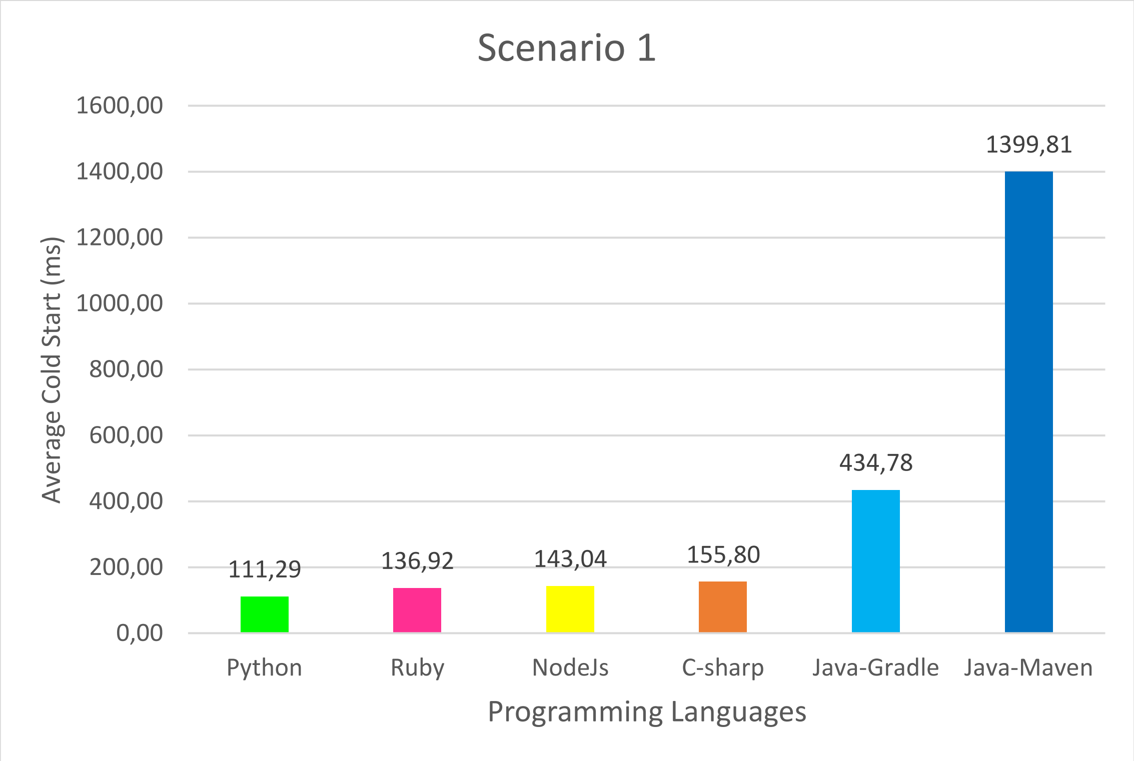


Figure 19 – Results from scenario 1

The results were as expected, confirming python had the lowest cold start time, while java had the highest, although, there is a noticeable difference between choosing gradle or maven. The study of these results also concluded that:

- Maven took over the triple of the time that gradle took to start executing.
- The difference between the cold start times of python, ruby, nodeJs and c-sharp, is much smaller than the difference between any of these and both java executions.
  - Ruby ≈ 1,23Python
  - NodeJs ≈ 1,29Python
  - C-sharp ≈ 1,4Python
  - Java-Gradle ≈ 3,91Python
  - Java-Maven ≈ 12,58Python

5.2 Scenario 2

Scenario 2 was executed, and the results can be seen in figure 20.

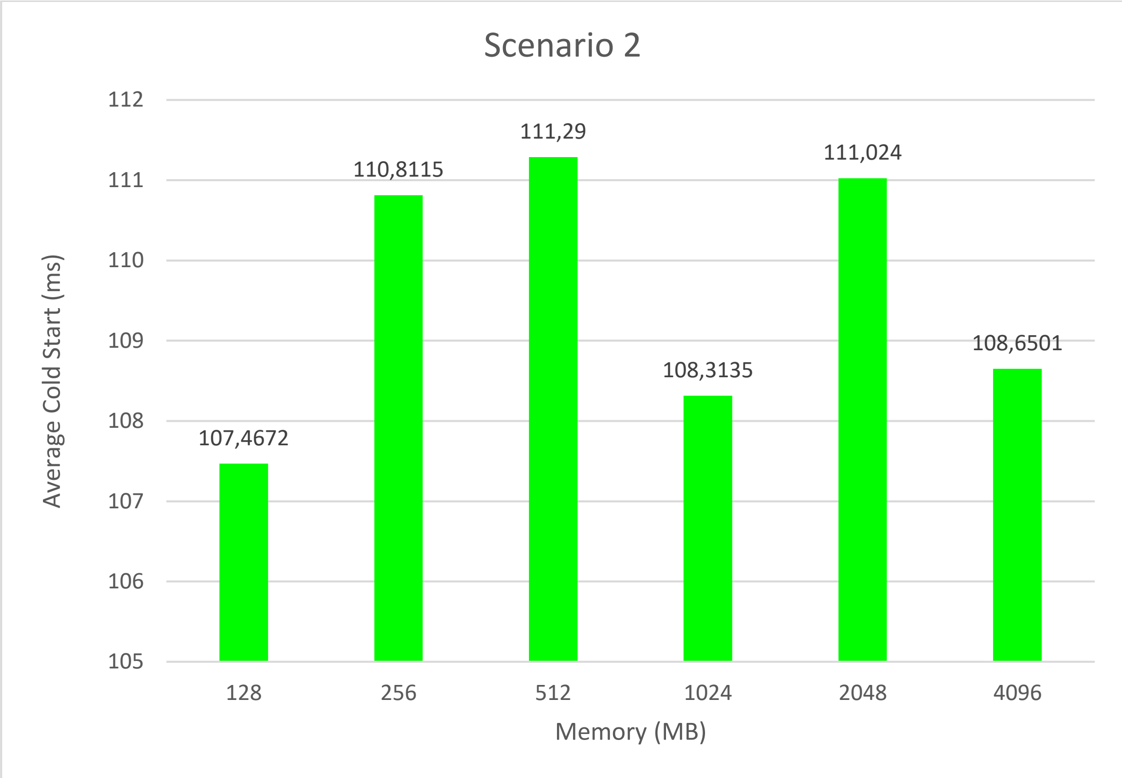


Figure 20 - Results from scenario 2

While not shocking in anyway, the results were still surprising. No good conclusion can be taken as the results are so close to each other, despite the memory size. One metric for context is the difference between the lowest average time (128MB) and the highest (512MB), which shows the highest had a 3,55% increase (approximately) compared to the lowest. All the other results are found between these and have even smaller differences. The most likely reason for this result is that python already has such a low cold start time, and, in result, giving it better resources has little significance. It would be interesting to see a similar test scenario with a more robust code, instead of a simple hello world, to verify if this result changes.

5.3 Scenario 3

Scenario 2 seemed to confirm the suspicion that certain languages benefit more from the allocation of better/more hardware resources than others, as python saw little to no difference. Scenario 3 provides better context for this discussion.

The overall results from scenario 3 can be seen in figure 21.

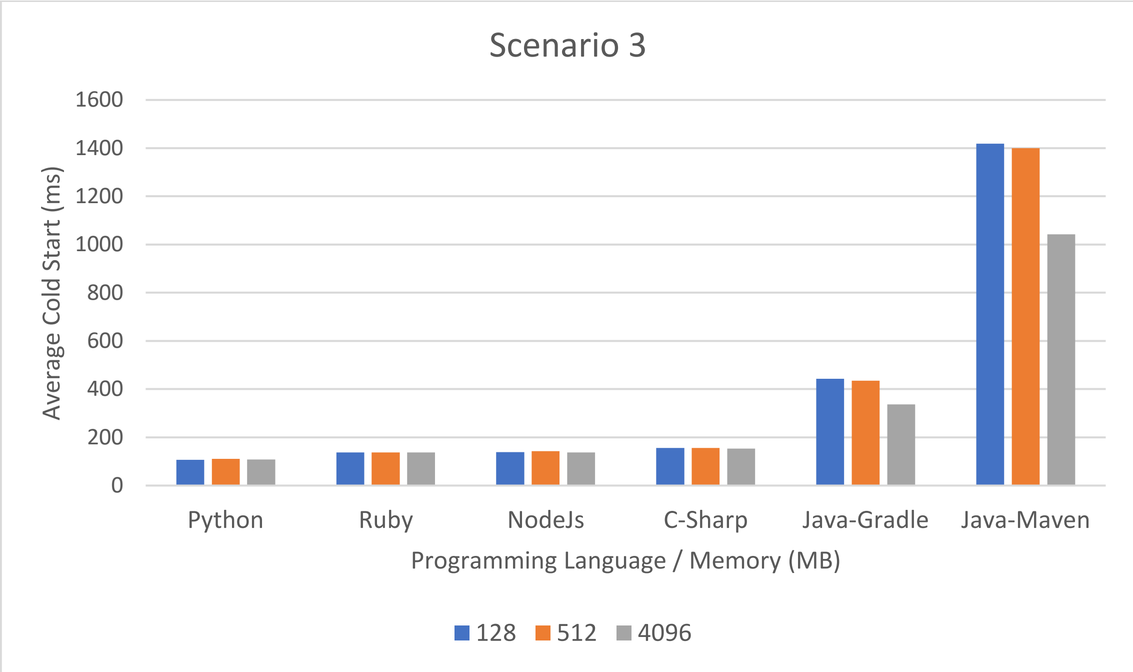


Figure 21 - Results from scenario 3

The overall conclusion from an "eye test" to figure 21 is that only java has a clear, significant improvement, thanks to the allocated resources. Also, using Java-Maven for reference, the improvement in cold start time between the executions with 128MB and 512MB is much smaller than the improvement between the executions with 512MB and 4096MB, which is expected, as the jump in memory is much larger, which in turn means that the jump in resources is also much larger.

Next, a closer look to each programming language is presented, with the objective to verify if the effects are similar across the board.

5.3.1 Python

Starting with the results from python (figure 22)

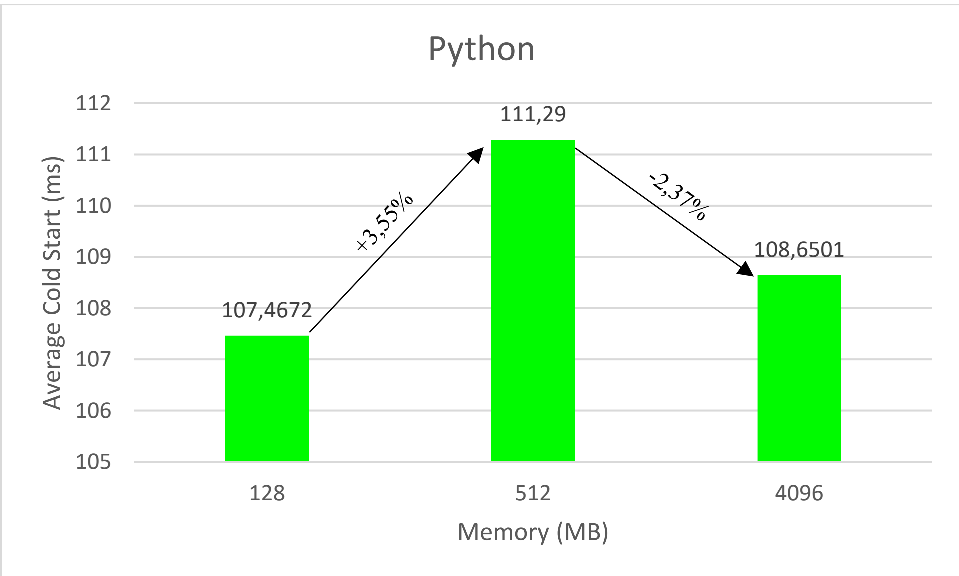


Figure 22 - Results from Scenario 3 (Python Only)

Surprisingly the lowest average cold start time is from the 128MB instance. Although it wasn't verified, as expected, that with more resources the cold start should go down, the difference between each execution is small enough to verify that, it's likely python already has such a low cold start time that adding more resources isn't as effective.

5.3.2 Ruby

Next up is ruby, which results can be seen in figure 23.

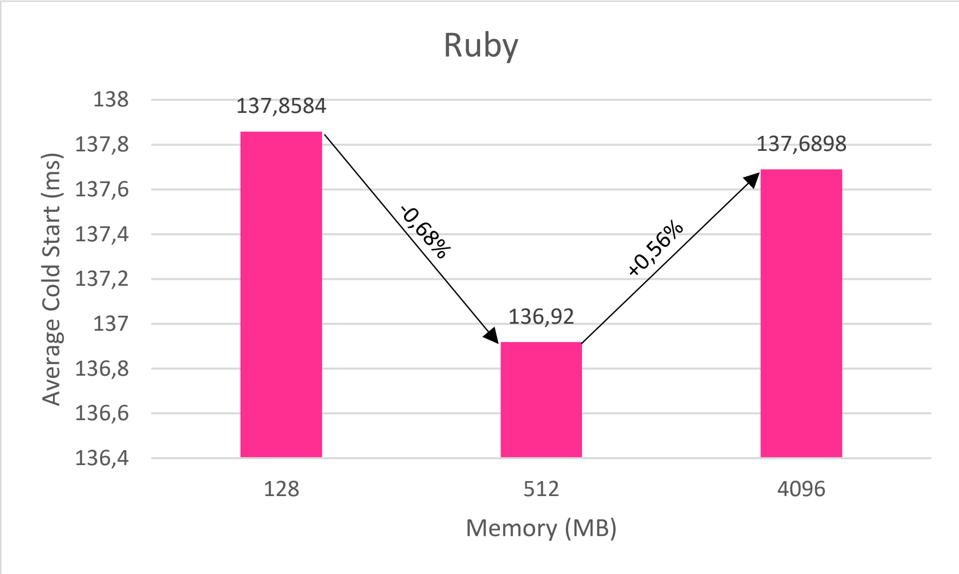


Figure 23 - Results from Scenario 3 (Ruby Only)

Ruby had pretty much the same cold start time for all memory sizes, having the difference between all executions been less than 1%.

5.3.3 NodeJs

For nodejs the results are in figure 24

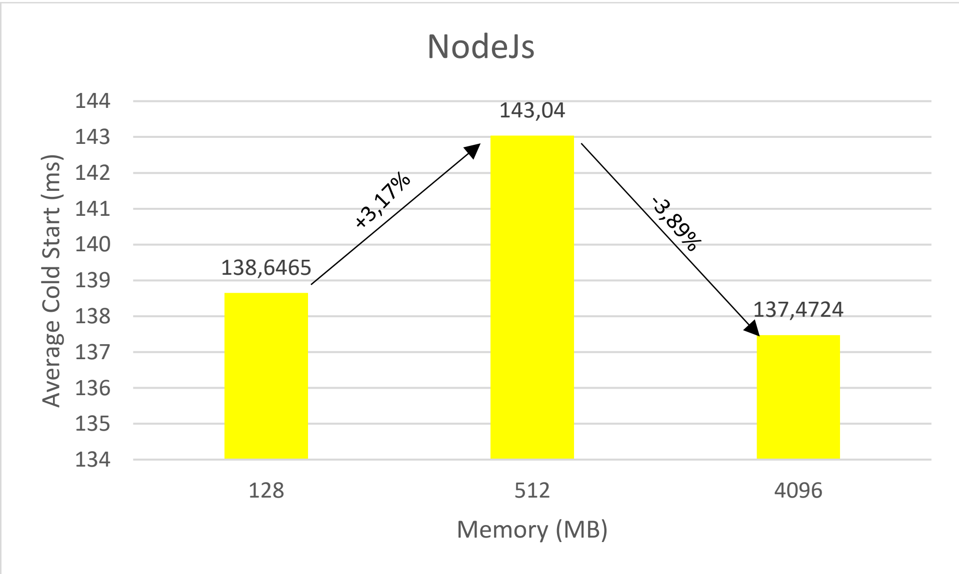


Figure 24 - Results from Scenario 3 (NodeJs Only)

The results seem similar to python at first sight, except that, in this case, the highest memory size had indeed the lowest cold start time. This being said, the result was pretty much equal to the execution with 128MB, having a difference between them of less than 1%.

5.3.4 C-Sharp

C-Sharp shares the similarity with java where it requires a build before deploying to the cloud, unlike python, ruby, and nodejs. Adding this fact to it having a higher cold start time, it should start showing signs of the expected behavior when it comes to the relationship between the cold start time and the allocated resources.

The results can be seen in figure 25.

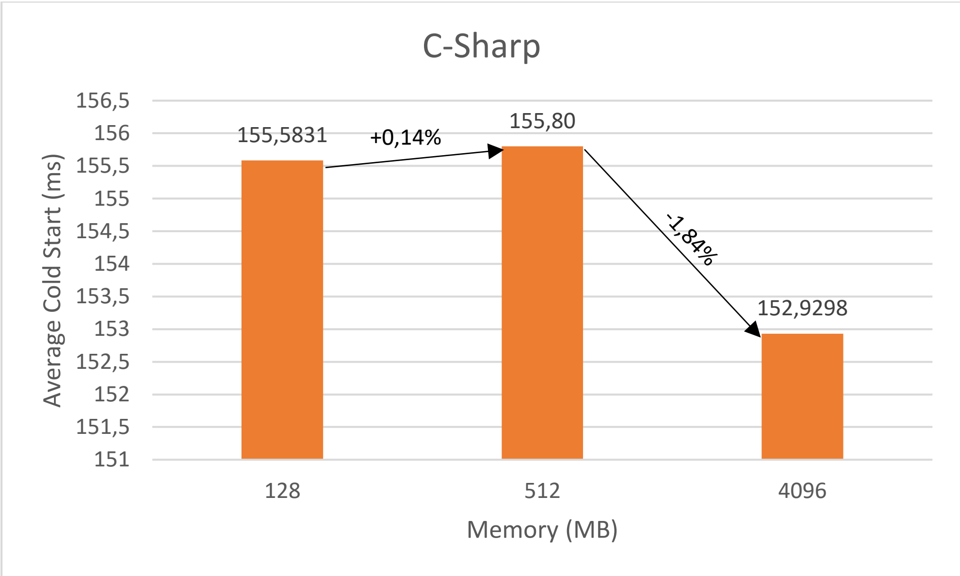


Figure 25 - Results from Scenario 3 (C-Sharp Only)

This figure shows results closer to those originally expected. Although the execution with 512MB ended up having a slightly worse result than the one with 128MB, the results are almost equal, which is expected, as it was already mentioned that the jump in memory is much smaller in this case when compared to the jump from 512MB to 4096MB. That being said, the difference still isn't clear, and the differences are still too small to say that the allocated resources were in fact relevant.

5.3.5 Java-Gradle

Java-Gradle is the first execution with java and the results can be seen in figure 26.

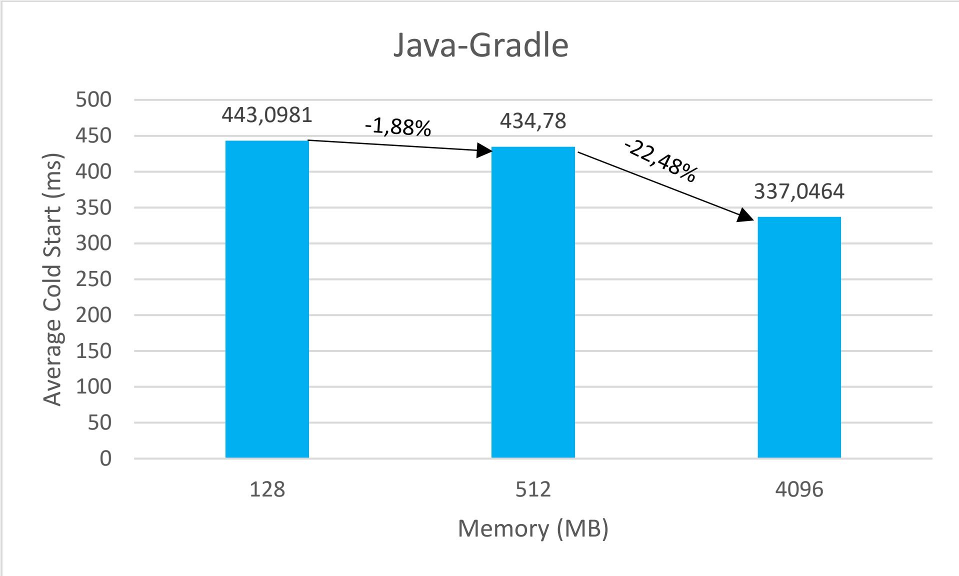


Figure 26 - Results from Scenario 3 (Java-Gradle Only)

Java-Gradle shows results as expected. There's an improvement from 128MB to 512MB and also from 512MB to 4096MB. The first improvement is much smaller, as the jump in memory is also much smaller. Here, the result is much clearer, as there's an improvement of over 20% when the memory is increased from 512MB to 4096MB.

5.3.6 Java-Maven

Finally, Java-Maven, which had the overall highest cold start time, should show the biggest improvement (results in figure 27).

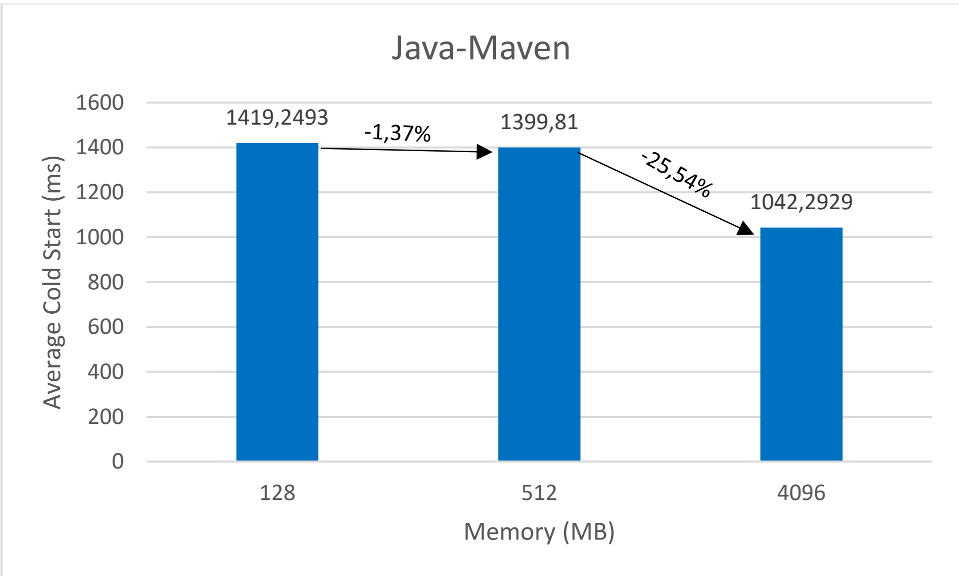


Figure 27 - Results from Scenario 3 (Java-Maven Only)

Java-Maven, just like Java-Gradle, shows results as expected, having improvements both times the memory was raised, being the second improvement significantly larger than the first

5.3.7 Closing thoughts

The results weren't as clear as expected but overall, the programming languages seem to show an improvement when more resources are allocated. The results could maybe be clearer if instead of a hello world function a more robust code was deployed, although, that's just a hypothesis, as there are other factors in question such as the fact that python, ruby, and nodejs don't require a build before deploying, while c-sharp and java do. Perhaps it would've been more adequate to use 1024MB or 2048MB instead of 512 MB in the medium instance, as the jump from 128MB to 512MB was so small that the results ended up being unclear. That being said, the original hypothesis seems to be correct, as the programming language with higher cold start times (java) showed significantly larger improvements than the rest.

5.4 Scenario 4

The analysis of the execution logs of scenario 4 resulted in the following table 3.

Hello-world-5min	**Hello-world-60min**	
15	25	**Number of instances / Executions with cold start**
19	1	**Average number of function executions per instance**
1 hour and 36 minutes	<1 hour	**Average duration of each instance**

Table 3 - Results from Scenario 4

These results allow us to take some interesting conclusions regarding the duration of each instance:

- A function that is executed regularly will keep the instance warm (ready for execution) for a longer duration.
- In the function that executed every 60 minutes every execution suffered a cold start.
- There's probably an optimal interval of time where new executions will keep the instance warm and prevent it from timing out. Although the exact value wasn't calculated here, these results allow that interval of time to be placed between 5 and 60 minutes.

The logs generated from the function that executed every 5 minutes were also used to make a cost analysis, in order to investigate if the cold start time is being billed or not. The analysis resulted in table 4.

Executions with cold start	**Executions without cold start**	
15	271	**Number of executions**
3 ms	1 ms	**Average billed duration**

Table 4 - Cost Analysis of the Results of Scenario 4

Table 4 clearly shows that, on average, the executions that suffer from cold start are billed with a higher duration, which translates in higher costs per execution. This means that it's worth optimizing serverless applications, not only for performance issues, but also to reduce costs and make the whole system more cost efficient.

5.5 Overall results from all scenarios

Having looked at the results from all scenarios, certain key takeaways are now available to help better understand what does indeed affect cold start and can be optimized:

- The choice of programming language matters for both performance and costs
  - Performance – Python had a clear advantage in performance when compared to all other programming languages.
  - Cost: Languages like python don’t seem to benefit as much from more/better hardware resources, therefore, the programming language itself becomes cost efficient as, under the right circumstances, the code can be run both in an instance with a small memory size or in an instance with a larger memory size, allowing lower costs per execution.
- Allocating more resources doesn’t always guarantee a better cold start time. Each case should be studied in order to identify if, in the context of the application, there would be benefits in having more and better resources.
- The programming languages with the lowest baseline cold start time benefit much more from the improvement of resources.
- Functions that are executed periodically with intervals that aren’t too long will have fewer cold starts, as the instances will take longer to time out.
- The existence of cold start has a significant effect in the price paid for each execution, therefore, reducing cold starts doesn’t only improve performance but also reduces the costs of maintaining an application or service running.

6 Conclusion

With this investigation several tests were conducted in order to evaluate what has a significant effect in the cold start time of a lambda function.

Not all results were as clear as expected. Scenarios 2 and 3 would benefit from diversification in order to clarify the results and what they mean for the cold start time. In future work scenario 2 could be explored in other languages, specifically java, in order to clarify the effects of resource allocation. Scenario 3 is in a similar situation where the first jump in memory was too small, making it perhaps inadequate to evaluate the hypothesis in question. In future work this scenario could be ran with more balanced intervals of memory, to get (possibly) clearer results.

That being said, the results still showed that factors such as the programming language and the amount of resources allocated can have a big impact in both performance and cost. Scenario 1 showed a clear difference between the average cold start times of several programming languages, making it a factor to take into account when a new application or service is being deployed. Scenario 4 showed the benefits (or drawbacks) of having executions happening in a set time interval, and showed as well that reducing cold start time will have an impact in reducing the total costs of maintaining an application or service.

7 References

[1] W.-T. Tsai, X. Sun and J. Balasooriya, "Service-Oriented Cloud Computing Architecture," in *2010 Seventh International Conference on Information Technology: New Generations*, Las Vegas, NV, USA, 2010.

[2] T. Diaby and B. B. Rad, "Cloud computing: a review of the concepts and deployment models," *International Journal of Information Technology and Computer Science,* vol. 9, no. 6, pp. 50-58, 2017.

[3] Y. Shaptunova, "4 Best Cloud Deployment Models Overview," 1993-2021 SaM Solutions., 08 April 2020. [Online]. Available: https://www.sam-solutions.com/blog/four-best-cloud-deployment-models-you-need-to-know/. [Accessed 05 April 2021].

[4] P. Mell and T. Grance, *The NIST definition of cloud computing,* 2011.

[5] P. Sareen, "Cloud computing: types, architecture, applications, concerns, virtualization and role of it governance in cloud," *International Journal of Advanced Research in Computer Science and Software Engineering,* vol. 3, no. 3, pp. 533-538, 2013.

[6] IBM Cloud Education, "FaaS (Function-as-a-Service)," IBM, 30 July 2019. [Online]. Available: https://www.ibm.com/cloud/learn/faas. [Accessed 14 07 2021].

[7] G. Adzic and R. Chatley, "Serverless computing: economic and architectural impact," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[8] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski and P. Suter, "Serverless computing: Current trends and open problems," in *Research advances in cloud computing*, Singapore, Springer, 2017, pp. 1-20.

[9] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW),* 2017.

[10] M. Stigler, Beginning Serverless Computing, Apress, 2018.

[11] Apple Inc., "siri," Apple Inc., 2021. [Online]. Available: https://www.apple.com/siri/. [Accessed 20 July 2021].

[12] Amazon.com, Inc. or its affiliates, "alexa," Amazon.com, Inc. or its affiliates, 2021. [Online]. Available: https://developer.amazon.com/en-US/alexa. [Accessed 20 July 2021].

[13] Google Cloud, "Google is named a Leader in 2020 Magic Quadrant for Cloud Infrastructure and Platform Services," Google, 2020. [Online]. Available: https://cloud.google.com/gartner-cloud-infrastructure-as-a-service. [Accessed 19 July 2021].

[14] AWS, "2020 Magic Quadrant for Cloud," Amazon Web Services, Inc. or its affiliates, 2020. [Online]. Available: https://pages.awscloud.com/GLOBAL-multi-DL-gartner-mq-cips-2020-learn.html. [Accessed 19 July 2021].

[15] M. Rifai, "Serverless showdown: AWS Lambda vs Azure Functions vs Google Cloud Functions," 2021 Serverless Heroes, Inc., 23 April 2021. [Online]. Available: https://acloudguru.com/blog/engineering/serverless-showdown-aws-lambda-vs-azure-functions-vs-google-cloud-functions. [Accessed 18 July 2021].

[16] Amazon Web Services, Inc. or its affiliates, "AWS Lambda Features," 2021, Amazon Web Services, Inc. or its affiliates, [Online]. Available: https://aws.amazon.com/lambda/features/?nc1=h_ls. [Accessed 18 July 2021].

[17] Amazon Web Services, Inc. or its affiliates, "AWS Lambda Pricing," 2021, Amazon Web Services, Inc. or its affiliates, [Online]. Available: https://aws.amazon.com/lambda/pricing/. [Accessed 18 July 2021].

[18] Amazon Web Services, Inc. or its affiliates, "AWS Lambda FAQs," 2021, Amazon Web Services, Inc. or its affiliates, [Online]. Available: https://aws.amazon.com/lambda/faqs/. [Accessed 18 July 2021].

[19] Amazon Web Services, Inc. or its affiliates, "Custom AWS Lambda runtimes," 2021, Amazon Web Services, Inc. or its affiliates, [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/runtimes-custom.html. [Accessed 18 July 2021].

[20] M. Shilkov, "Comparison of Cold Starts in Serverless Functions across AWS, Azure, and GCP," Mikhail Shilkov, 5 January 2021. [Online]. Available: https://mikhail.io/serverless/coldstarts/big3/. [Accessed 18 July 2021].

[21] Microsoft, "Azure Functions pricing," Microsoft, [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/functions/. [Accessed 20 July 2021].

[22] Microsoft, "Azure Functions hosting options," Microsoft, 17 August 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale. [Accessed 20 July 2021].

[23] Microsoft, "Azure Functions," Microsoft, [Online]. Available: https://azure.microsoft.com/en-us/services/functions/. [Accessed 20 July 2021].

[24] Microsoft, "Azure Functions HTTP triggers and bindings overview," Microsoft, 14 February 2020. [Online]. Available: https://docs.microsoft.com/en-

us/azure/azure-functions/functions-bindings-http-webhook. [Accessed 20 July 2021].

[25] Google Cloud, "Cloud Functions pricing," Google Cloud, [Online]. Available: https://cloud.google.com/functions/pricing. [Accessed 20 July 2021].

[26] Google Cloud, "Cloud Functions," Google Cloud, [Online]. Available: https://cloud.google.com/functions#section-14. [Accessed 20 July 2021].

[27] Google Cloud, "Workflows," Google Cloud, [Online]. Available: https://cloud.google.com/workflows. [Accessed 20 July 2021].

[28] Google Cloud, "Cloud Functions Execution Environment," Google Cloud, [Online]. Available: https://cloud.google.com/functions/docs/concepts/exec#timeout. [Accessed 20 July 2021].

[29] Google Cloud, "About Custom Runtimes," Google Cloud, [Online]. Available: https://cloud.google.com/appengine/docs/flexible/custom-runtimes/about-custom-runtimes. [Accessed 20 July 2021].

[30] [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/runtimes-context.html.

[31] [Online]. Available: https://www.serverless.com/.

[32] [Online]. Available: https://www.serverless.com/framework/docs/providers/aws/cli-reference/create. [Accessed 6 11 2021].

[33] L. Wang, M. Li, Y. Zhang, T. Ristenpart and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," in *2018 USENIX Annual Technical Conference*, Boston, MA, 2018.

[34] Y. ". Cui, "How does language, memory and package size affect cold starts of AWS Lambda?," A Cloud Guru, 14 June 2017. [Online]. Available: https://acloudguru.com/blog/engineering/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda. [Accessed 09 11 2021].

[35] AWS, "Configuring Lambda function options," Amazon Web Services, Inc. or its affiliates, [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html#configuration-memory-console. [Accessed 10 11 2021].

[36] A. Altvater, "Gradle vs. Maven: Performance, Compatibility, Builds, & More," Stackify, 29 June 2017. [Online]. Available: https://stackify.com/gradle-vs-maven/. [Accessed 16 November 2021].

[37] Gradle Inc., "What is Gradle?," Gradle Inc., [Online]. Available: https://docs.gradle.org/current/userguide/what_is_gradle.html. [Accessed 16 November 2021].

[38] Apache Maven Project, "What is Maven?," The Apache Software Foundation, 15 November 2021. [Online]. Available: https://maven.apache.org/what-is-maven.html. [Accessed 16 November 2021].

[39] Stackscale, "Most popular programming languages in 2021," Stackscale, 13 September 2021. [Online]. Available: https://www.stackscale.com/blog/popular-programming-languages-2021/. [Accessed 15 November 2021].

[40] S. Cass, "Top Programming Languages 2021," IEEE Spectrum, [Online]. Available: https://spectrum.ieee.org/top-programming-languages/. [Accessed 15 November 2021].

8 Appendices

8.1 Appendix 1 – More Detailed Basic Setup Guide

The following setup was done in a M1 Macbook Air.

This document will provide a step-by-step guide that can be followed to deploy a simple hello world lambda function in python.

The only requirement up to this point is to have an AWS account.

The first step is to install the serverless framework. In this case the installation will happen through npm, but other options are available. In this case, the device in use doesn't have npm, therefore, node was installed, providing also npm. To install node, the command used was "brew install node" (figure A1.1)

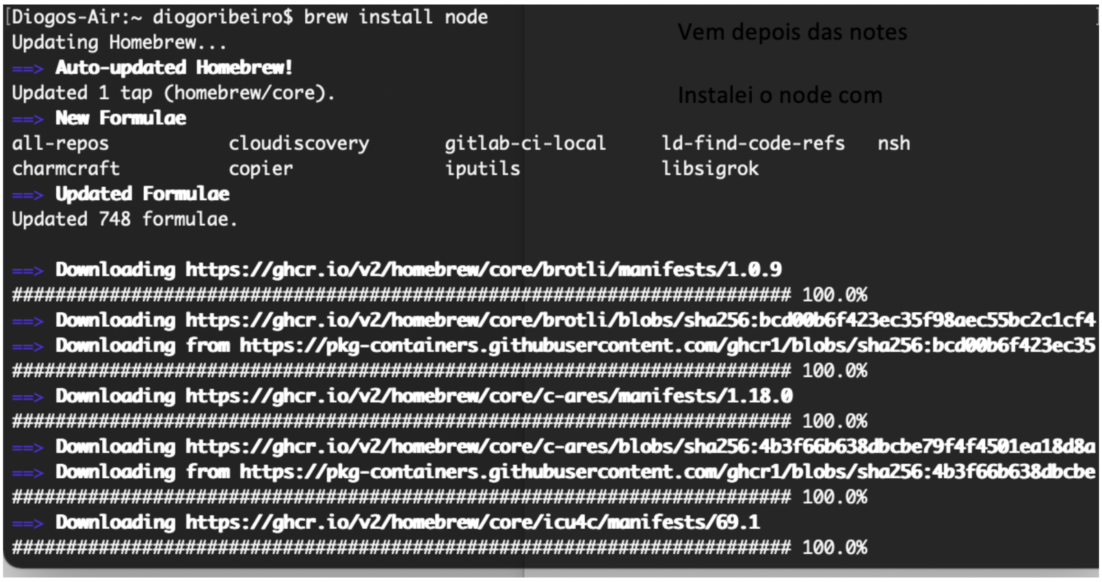


Figure A1.1

After that, it should be confirmed that everything was installed correctly (figure A1.2)

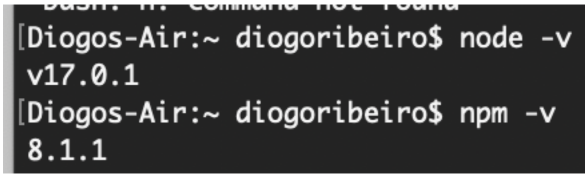


Figure A1.2

Now that npm is installed, the serverless framework can be installed with the command "npm install -g serverless" (figure A1.3)

Figure A1.3

Now, the serverless framework is installed, but it can't access the desired AWS account, in order to deploy the functions. For that a user must be created in AWS through the IAM service.

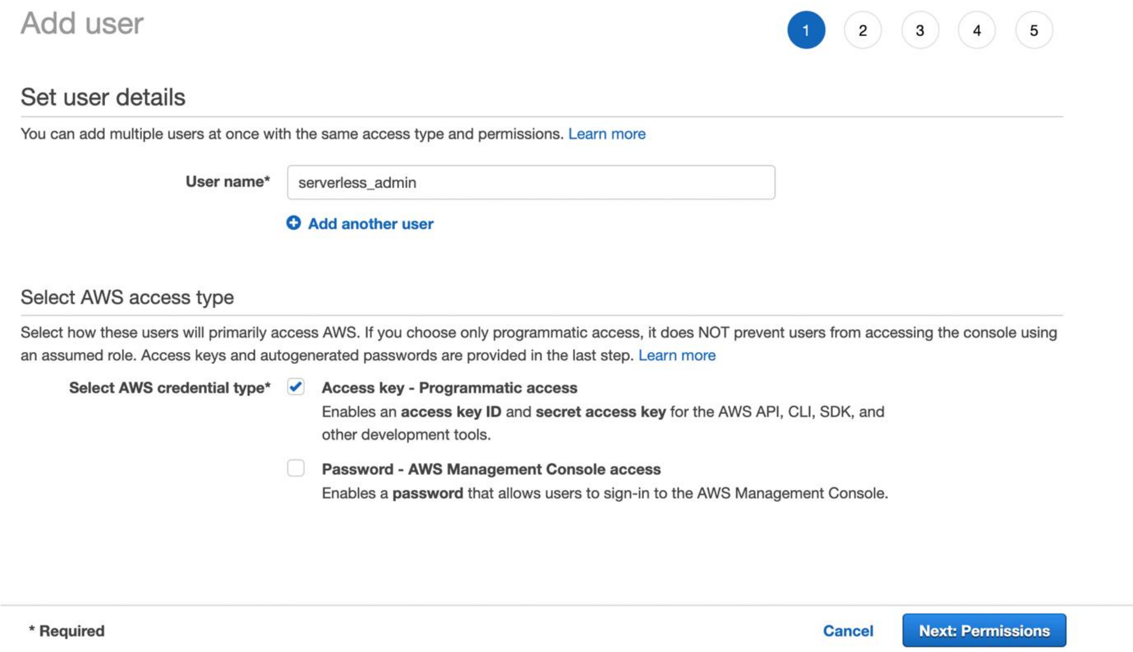


Figure A1.4

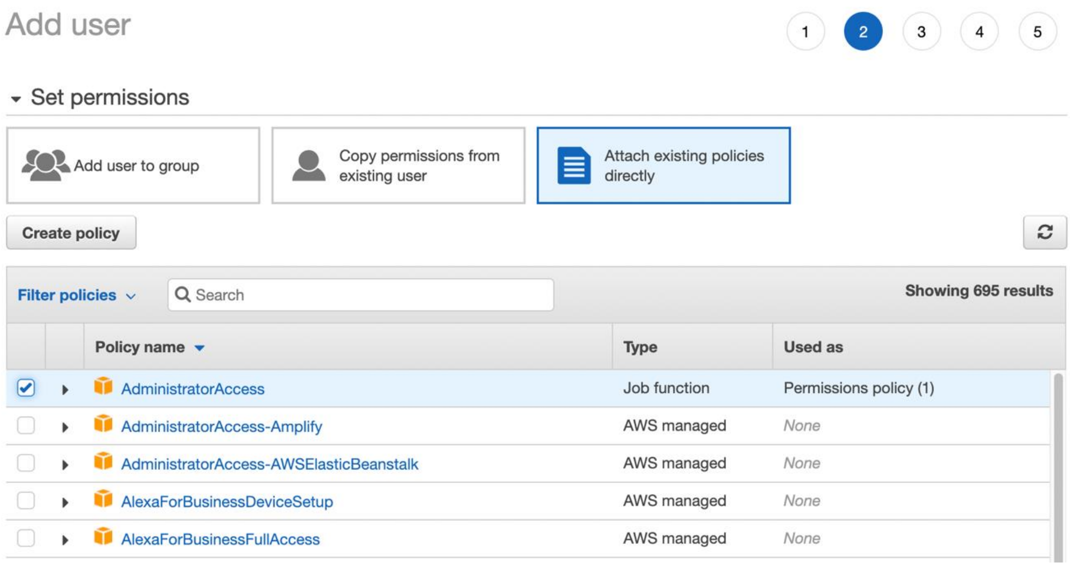


Figure A1.5

In this case the recommended settings were used, so a user with the name “serverless_admin” and programmatic access (figure A1.4) was created with full administrator permissions (figure A1.5). More menus will appear but, at the point of time in which this document was written, the next configurations were ignored and skipped.

After finishing the user creation, credentials will be provided. These credentials will be used to configure the serverless framework (figure A1.6), so it can access and manage the services in the chosen AWS account.

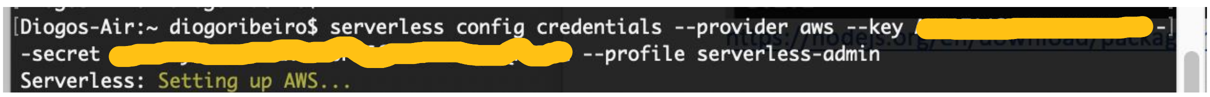


Figure A1.6

The serverless framework is now configured and ready to function properly with the chosen AWS account.

A note before advancing, the "serverless" command can be shortened to "sls", which means commands like "serverless deploy", can also be written as "sls deploy".

Now, before deploying a function, the code must be written. The serverless framework community makes several templates available, which can be used to facilitate the starting point for development. In this example, the template "aws-python" will be used to generate the files necessary. To create the files, resorting to the template, the command used is "sls create --template aws-python --path hello-world-python-cli-01" (figure A1.7). In this case the extra "--path" flag was added to indicate where the files should be generated.

```
Diogos-Air:functions diogoribeiro$ sls create --template aws-python --path hello-world-python-cli-01
Serverless: Generating boilerplate...
Serverless: Generating boilerplate in "/Users/diogoribeiro/Desktop/AWS_Stuff/functions/hello-world-python-cli-01"
Serverless: Successfully generated boilerplate for template: "aws-python"
Diogos-Air:functions diogoribeiro$ ls
hello-world-python-cli-01
Diogos-Air:functions diogoribeiro$ ls hello-world-python-cli-01/
handler.py      serverless.yml
```

Figure A1.7

The files "handler.py" (figure A1.8) and "serverless.yml" (figure A1.9) were generated. Handler.py is the file where the code from the actual function is written. Serverless.yml is the file with the necessary configurations for AWS. Serverless.yml can be configured for several use cases, in this case, only a hello world function will be deployed so there isn't much need to explore these configurations. That being said, some ideas of what can be configured here are:

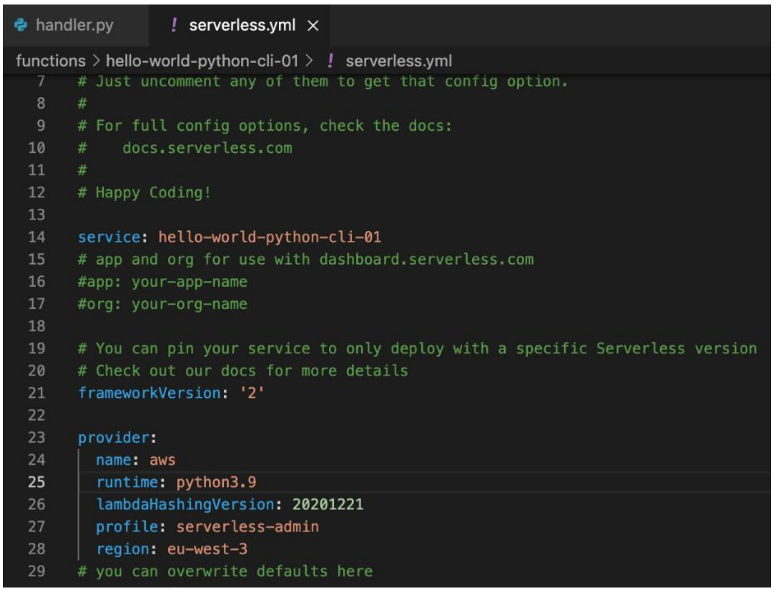
- Permissions the function has
- Region where the code will be deployed
- What will trigger the function execution (API call, DB field modification, …) and also which function will be executed with said trigger (in this case the function executed will be "hello", located in handler.py)
- Amount of RAM allocated
- Timeout duration
- How the function interacts with other services (for example which information will be generated in the logs)

In this case, the code automatically generated for handler.py was deleted and replaced with a simple hello world example (figure A1.8)

```
handler.py ×    ! serverless.yml
functions > hello-world-python-cli-01 > handler.py > hello
def hello(event, context):
    print("Print 01")
    return "Hello world 01"
```

Figure A1.8

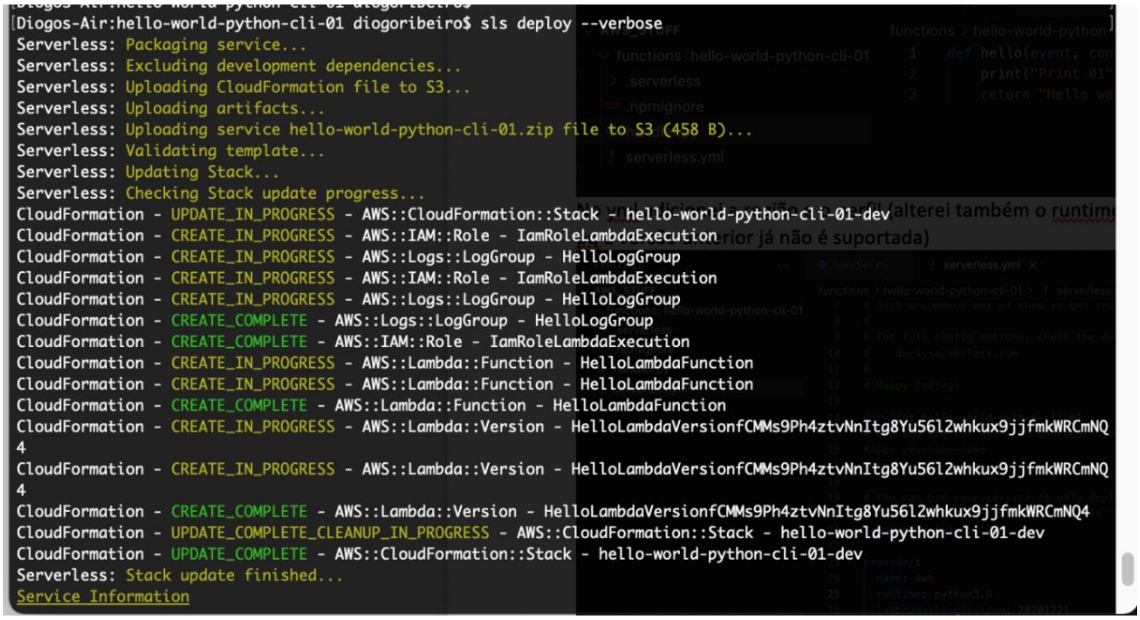
In serverless.yml the region was set to eu-west-3 (Paris), the profile was set to “serverless-admin”, and the runtime was set to python3.9 (figure A1.9)



```
handler.py    ! serverless.yml ×
functions > hello-world-python-cli-01 > ! serverless.yml
# Just uncomment any of them to get that config option.
#
# For full config options, check the docs:
#    docs.serverless.com
#
# Happy Coding!

service: hello-world-python-cli-01
# app and org for use with dashboard.serverless.com
#app: your-app-name
#org: your-org-name

# You can pin your service to only deploy with a specific Serverless version
# Check out our docs for more details
frameworkVersion: '2'

provider:
  name: aws
  runtime: python3.9
  lambdaHashingVersion: 20201221
  profile: serverless-admin
  region: eu-west-3
# you can overwrite defaults here
```

Figure A1.9

Once that’s done, the code is ready to be deployed to the cloud with the command “sls deploy” – figure A1.10 and A1.11 - (in this case the “--verbose" flag was added in order to get the logs from the deployment). Reading the logs, more services were created in AWS, not just the lambda function. These services are necessary for everything to run properly, and the serverless framework handles everything for the client who’s deploying the function



```
Diogos-Air:hello-world-python-cli-01 diogoribeiro$ sls deploy --verbose
Serverless: Packaging service...
Serverless: Excluding development dependencies...
Serverless: Uploading CloudFormation file to S3...
Serverless: Uploading artifacts...
Serverless: Uploading service hello-world-python-cli-01.zip file to S3 (458 B)...
Serverless: Validating template...
Serverless: Updating Stack...
Serverless: Checking Stack update progress...
CloudFormation - UPDATE_IN_PROGRESS - AWS::CloudFormation::Stack - hello-world-python-cli-01-dev
CloudFormation - CREATE_IN_PROGRESS - AWS::IAM::Role - IamRoleLambdaExecution
CloudFormation - CREATE_IN_PROGRESS - AWS::Logs::LogGroup - HelloLogGroup
CloudFormation - CREATE_IN_PROGRESS - AWS::IAM::Role - IamRoleLambdaExecution
CloudFormation - CREATE_IN_PROGRESS - AWS::Logs::LogGroup - HelloLogGroup
CloudFormation - CREATE_COMPLETE - AWS::Logs::LogGroup - HelloLogGroup
CloudFormation - CREATE_COMPLETE - AWS::IAM::Role - IamRoleLambdaExecution
CloudFormation - CREATE_IN_PROGRESS - AWS::Lambda::Function - HelloLambdaFunction
CloudFormation - CREATE_IN_PROGRESS - AWS::Lambda::Function - HelloLambdaFunction
CloudFormation - CREATE_COMPLETE - AWS::Lambda::Function - HelloLambdaFunction
CloudFormation - CREATE_IN_PROGRESS - AWS::Lambda::Version - HelloLambdaVersionfCMMs9Ph4ztvNnItg8Yu56l2whkux9jjfmkWRCmNQ
4
CloudFormation - CREATE_IN_PROGRESS - AWS::Lambda::Version - HelloLambdaVersionfCMMs9Ph4ztvNnItg8Yu56l2whkux9jjfmkWRCmNQ
4
CloudFormation - CREATE_COMPLETE - AWS::Lambda::Version - HelloLambdaVersionfCMMs9Ph4ztvNnItg8Yu56l2whkux9jjfmkWRCmNQ4
CloudFormation - UPDATE_COMPLETE_CLEANUP_IN_PROGRESS - AWS::CloudFormation::Stack - hello-world-python-cli-01-dev
CloudFormation - UPDATE_COMPLETE - AWS::CloudFormation::Stack - hello-world-python-cli-01-dev
Serverless: Stack update finished...
Service Information
```

Figure A1.10

```
Serverless: Stack update finished...
Service Information
service: hello-world-python-cli-01
stage: dev
region: eu-west-3
stack: hello-world-python-cli-01-dev
resources: 6
api keys:
  None
endpoints:
functions:
  hello: hello-world-python-cli-01-dev-hello
layers:
  None

Stack Outputs
HelloLambdaFunctionQualifiedArn: arn:aws:lambda:eu-west-3:620393413902:function:hello-world-python-cli-01-dev-hello:1
ServerlessDeploymentBucketName: hello-world-python-cli-0-serverlessdeploymentbuck-1uimko8k6oxb6
```

Figure A1.11

Once deployed, the function can now be executed (figure A1.12). In this case, the trigger for this execution will be a command, but, in a real use case scenario, several other options are available. The command used is “sls invoke -f [FUNCTION_NAME]” (in this case, the -l flag was added, in order to display the execution logs).

```
Diogos-Air:hello-world-python-cli-01 diogoribeiro$ serverless invoke -f hello -l
"Hello world 01"
--------------------------------------------------------------------
START RequestId: 8b8dab5a-692d-421e-bfbb-f50e246e321f Version: $LATEST
Print 01
END RequestId: 8b8dab5a-692d-421e-bfbb-f50e246e321f
REPORT RequestId: 8b8dab5a-692d-421e-bfbb-f50e246e321f  Duration: 1.26 ms       Billed Duration: 2 ms   Memory Size: 102
4 MB    Max Memory Used: 36 MB  Init Duration: 120.76 ms


Diogos-Air:hello-world-python-cli-01 diogoribeiro$ serverless invoke -f hello -l
"Hello world 01"
--------------------------------------------------------------------
START RequestId: b67f782e-5478-4a1c-8377-8f174beab13a Version: $LATEST
Print 01
END RequestId: b67f782e-5478-4a1c-8377-8f174beab13a
REPORT RequestId: b67f782e-5478-4a1c-8377-8f174beab13a  Duration: 0.90 ms       Billed Duration: 1 ms   Memory Size: 102
4 MB    Max Memory Used: 36 MB


Diogos-Air:hello-world-python-cli-01 diogoribeiro$
```

Figure A1.12

The function was executed successfully as we can see from the return of the “Hello world 01” statement. Notice how the first execution indicates an “Init Duration” while the second one doesn’t. This means that the first execution suffered from a cold start, and a new environment had to be provisioned in order to execute the function

Now, to delete everything that has been deployed to the cloud, the command “serverless remove” is used (figure A1.13). This deletes the services in the cloud, not the local files.

```
Diogos-Air:toDel diogoribeiro$ sls remove
Serverless: Getting all objects in S3 bucket...
Serverless: Removing objects in S3 bucket...
Serverless: Removing Stack...
Serverless: Checking Stack delete progress...
.......
Serverless: Stack delete finished...

Serverless: Stack delete finished...
```

Figure A1.13

When changing the code of an already deployed function, the function must be re-deployed to the cloud. The code is changed locally, and then sent to the cloud. For this, the instinct can be to use the command “sls deploy” once again. This will do the trick but will also be very inefficient. This command will re-deploy the entire stack of services and not just the new function code. If no other services were altered aside from the lambda function, the command “serverless deploy function -f [FUNCTION_NAME]” will be much better, as it will update only the code of the lambda function.

8.2 Appendix 2 – Example configuration (test scenario 2)

Handler.py:

```
handler.py ×
C: > Users > Utilizador > Downloads > handler.py
def hello(event, context):
    return "Hello World"
```

Serverless.yml:

C: > Users > Utilizador > Downloads > ! serverless.yml

```yaml
service: hello-world-python

frameworkVersion: '2'

provider:
  name: aws
  runtime: python3.9
  lambdaHashingVersion: 20201221
  profile: serverless-admin
  region: eu-west-3

functions:
  hello-128:
    handler: handler.hello
    memorySize: 128
  hello-256:
    handler: handler.hello
    memorySize: 256
  hello-1024:
    handler: handler.hello
    memorySize: 1024
  hello-2048:
    handler: handler.hello
    memorySize: 2048
  hello-4096:
    handler: handler.hello
    memorySize: 4096

```

8.3 Appendix 3 – Example execution script (test scenario 2)

ExecutionScript.sh:

C: > Users > Utilizador > Downloads > $ ExecutionScript.sh

```bash
#!/bin/bash

start=$(date)

for i in {1..100}
do
    echo "$i/100 - Starting"
    cd hello-world-python
    sls deploy #--verbose
    sleep 1
    echo "Invoke 128 - $i"
    sls invoke -f hello-128
    echo "Invoke 256 - $i"
    sls invoke -f hello-256
    echo "Invoke 1024 - $i"
    sls invoke -f hello-1024
    echo "Invoke 2048 - $i"
    sls invoke -f hello-2048
    echo "Invoke 4096 - $i"
    sls invoke -f hello-4096
    sleep 10
    sls logs -f hello-128 > output_log_128.txt
    sls logs -f hello-256 > output_log_256.txt
    sls logs -f hello-1024 > output_log_1024.txt
    sls logs -f hello-2048 > output_log_2048.txt
    sls logs -f hello-4096 > output_log_4096.txt
    tail -2 output_log_128.txt | cut -f 6 >> ../init_time_128.txt
    tail -2 output_log_256.txt | cut -f 6 >> ../init_time_256.txt
    tail -2 output_log_1024.txt | cut -f 6 >> ../init_time_1024.txt
    tail -2 output_log_2048.txt | cut -f 6 >> ../init_time_2048.txt
    tail -2 output_log_4096.txt | cut -f 6 >> ../init_time_4096.txt
    rm output_log_*
    sls remove
    cd ..
    echo "$i/100 - Done"

done
```

```bash

cat init_time_128.txt | sed '/^$/d' > init_processed_128.txt
cat init_processed_128.txt | cut -d " " -f 3 > init_processed_128_time_only.txt
cat init_time_256.txt | sed '/^$/d' > init_processed_256.txt
cat init_processed_256.txt | cut -d " " -f 3 > init_processed_256_time_only.txt
cat init_time_1024.txt | sed '/^$/d' > init_processed_1024.txt
cat init_processed_1024.txt | cut -d " " -f 3 > init_processed_1024_time_only.txt
cat init_time_2048.txt | sed '/^$/d' > init_processed_2048.txt
cat init_processed_2048.txt | cut -d " " -f 3 > init_processed_2048_time_only.txt
cat init_time_4096.txt | sed '/^$/d' > init_processed_4096.txt
cat init_processed_4096.txt | cut -d " " -f 3 > init_processed_4096_time_only.txt

rm init_time_128.txt
rm init_processed_128.txt
rm init_time_256.txt
rm init_processed_256.txt
rm init_time_1024.txt
rm init_processed_1024.txt
rm init_time_2048.txt
rm init_processed_2048.txt
rm init_time_4096.txt
rm init_processed_4096.txt

finish=$(date)

echo "Started: $start"
echo "Finished: $finish"
```