1- CLOUD COMPUTING

1.1 Introduction:

The idea of "cloud" computing traces back to the definition of the concept of utility computing proposed by John McCarthy in 1961:

"If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility. ... The computer utility could become the basis of a new and important industry."

According to this proposition, a set of public IT capabilities could be seen and exploited "as a service", on a pay-per-use basis such as other utilities like electricity and gas. Before 2002, when Amazon launched its Amazon Web Services (AWS) platform, companies could only deploy their services on their own servers and make them accessible to customers with a client-server approach. So, the servers had to be managed and maintained by the companies themselves.

Nowadays, many public vendors such as Amazon, Google and Microsoft offer a wide range of solutions that allow customers to access and "rent" IT resources through the Internet. These solutions differ from the point of view of abstraction level and the possibility of configuration of the underlying capabilities. In fact, for cloud computing we do not refer only to a collection of remote computing resources, but also to all the instruments and mechanisms to dynamically manage those resources.

1.2 Definition:

According to NIST definition, "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

This model is composed of five main characteristics:

- *On-demand self-service*: every consumer can obtain access to cloud resources automatically at any time, without the need for human interaction.
- *Broad network access*: cloud resources can be used over the network through standard mechanisms that promote the access via many heterogeneous clients (e.g., laptops, mobile phones and tablets).
- *Resource pooling*: cloud resources are pooled in order to serve multiple consumers. Resources can be either physical or virtual and they are dynamically assigned according to consumer demand. The location is transparent to the customers, which can only specify the desired location at a high level of abstraction (e.g., country, state or datacenter).
- *Rapid elasticity*: capabilities can be elastically provisioned and released to scale rapidly outward or inward according to demand. Resources appear unlimited to the customers.
- *Measured service*: given that the resources are assigned on a pay-per-use basis, cloud systems control and optimize resource use by leveraging a metering capability. Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

1.2 Benefits and drawbacks:

Cloud computing is not a one-size-fits-all affair. We can certainly say that it presents many advantages compared to on-premise solutions, such as:

- Reduced investments and proportional costs: The most common economic rationale for investing in cloud-based IT resources is in the reduction or outright elimination of up-front IT investments, namely hardware and software purchases and ownership costs. In this way, enterprises can start small and gradually increase their resource allocation as the demand for it grows. Moreover, the reduction of up-front capital expenses allows for the capital to be redirected to the core business investment.
- Increased scalability: By providing pools of IT resources, along with tools and technologies designed to leverage them collectively, clouds can instantly and dynamically allocate and de-allocate IT resources to cloud consumers, on-demand or via the cloud consumer’s direct configuration.

- Increased availability and reliability: Cloud providers generally offer “resilient” IT resources for which they are able to guarantee high levels of availability. Furthermore, the modular architecture of cloud environments provides extensive failover support that increases reliability.

Nevertheless, some aspects of cloud systems make them not suitable for every situation. First of all, the moving of business data to the cloud means that the responsibility over data security becomes shared with the cloud provider. Furthermore, centralization and cumulation of data on service providers side make them more likely to be attacked by malicious agents. So, if security is a key constraint to a given service, cloud computing could not be the right solution. Another drawback can be identified in reduced operational governance control. In fact, cloud consumers are usually allotted a level of governance control that is lower than that over on-premise IT resources. This can introduce risks associated with how the cloud provider operates its cloud, as well as the external connections that are required for communication between the cloud and the cloud consumer.

1.3 Architecture:

To provide IT resources “as a service”, cloud computing systems have a structure based on components that can communicate with each other via well defined interfaces. The main components are four:

- One cloud platform, that consists of an anywhere available interface accessed via web to interact with the real or virtual internal infrastructure.
- One virtualisation infrastructure and a management system for the control, monitoring, and billing for client requests.
- one internal memory system typically obtained via a database.
- one internal manager to handle external requests (management, queuing, and controlling).

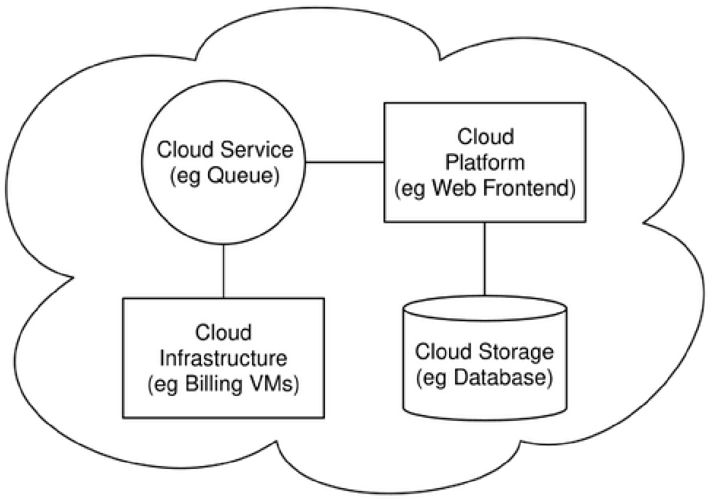

Cloud Service (eg Queue)
Cloud Platform (eg Web Frontend)
Cloud Infrastructure (eg Billing VMs)
Cloud Storage (eg Database)

Figure 1.1: The four main components of cloud computing systems.

1.4 Service models:

Cloud systems offer different models of service, based on the desired level of abstraction, ease of management and need of control of the customers on the underlying components. In traditional IT, an organization consumes IT-assets by purchasing them, installing them, managing and maintaining them in its own on-premise data center. The idea behind cloud computing systems is to provide resources (e.g., servers, virtual machines, databases, applications, …) "as a service" through Internet. In this way, customers don't have to install anything, and they can use cloud provider's services on a pay-per-use basis.

Service models form a layered architecture on top of a physical layer managed by the cloud provider. The standard service models are three, but many other intermediate levels exist. Starting from the lower level of abstraction, we find:

1) **Infrastructure as a Service (IaaS)**: at this layer, cloud-hosted computing infrastructure is accessed on-demand and configurable much the same way as on-premise hardware. The difference is that cloud providers host, manage and maintain computing resources. Typically, IaaS customers can choose between virtual machines (VMs) hosted on shared physical hardware or bare metal

servers on dedicated machines. So, customers are able to deploy and run arbitrary software, which can include operating systems and applications.
From the technical point of view, cloud providers use a hypervisor to run the VMs as guests. Pools of hypervisors can support large numbers of VMs and the ability to scale services up and down according to the demand. Many Linux containers run in isolated partitions of a single Linux kernel that executes directly on the physical layer. In this way, cloud providers can manage many clients at the same time.

2) **Platform as a Service (PaaS)**: this solution offers a cloud-based platform for developing, running and managing applications. So, PaaS vendors provide a development environment for developers. This environment includes pre-configured resources such as servers, operating systems, databases, runtimes and middlewares. Clients can only have control on the deployed applications and on the settings for the application-hosting environment. Cloud providers offer toolkits and standards to facilitate the development of applications and channels for distribution and payment.
3) **Software as a Service (SaaS)**: at this layer, resources are simple applications available via web. So, customers can use applications running on a cloud infrastructure, on which they don’t have control. Applications are accessible from heterogeneous devices through either a light client interface, such as a web browser, or a program interface.

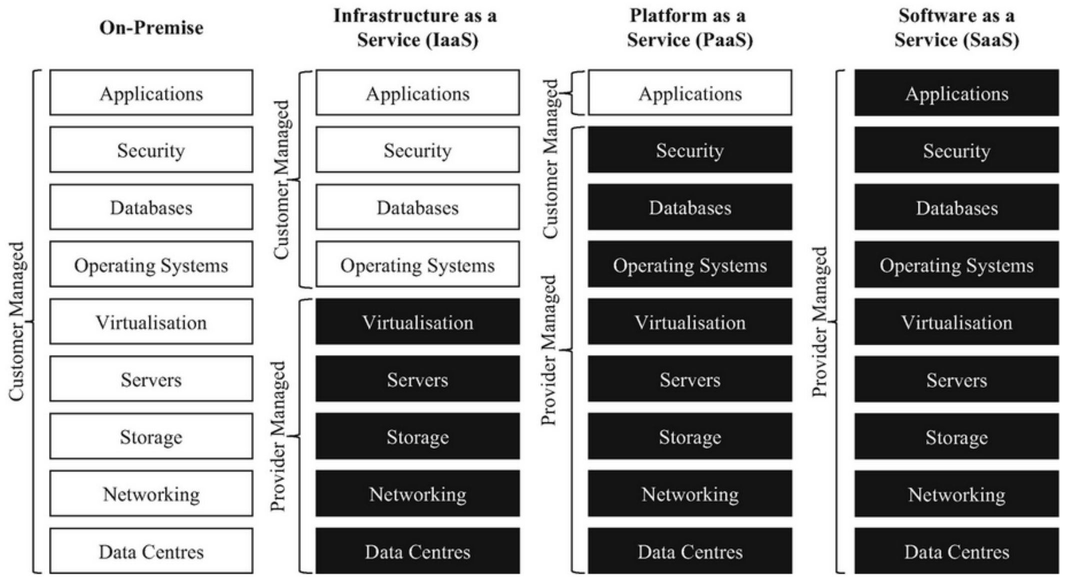


Figure 1.2: Main types of service models

1.5 Deployments models:

Even if previous characteristics are related to a generic public cloud, different deployment models are possible. Among them, the most interesting are:

- *Public cloud*: the cloud infrastructure is provisioned for open use by the general public.
- *Private cloud*: the cloud infrastructure is provisioned for exclusive use by a single organization. The advantage is that it is possible to define strategies very tailored to specific issues and to obtain more flexibility in the management of hardware resources. Certainly, it is a more expensive solution, and it doesn't offer the same guarantees of public clouds in case of faults and extreme situations.
- *Hybrid cloud*: The cloud infrastructure is a composition of two or more distinct clouds that remain separated entities but are bounded together by standardized or proprietary technologies that enable data and application portability. Such a solution can be adopted, for example, to keep the storage of confidential data physically separated from other storage or to load balance the workload between different clouds.
- *Community cloud*: The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.

2-SERVERLESS COMPUTING

2.1 Introduction:

According to the definition provided by Red Hat, “Serverless is a cloud-native development model that allows developers to build and run applications without having to manage servers.” It doesn’t mean that there are no more servers behind our services, but their management is completely demanded to cloud providers. So, end users have only to concentrate on the business logic of their applications.

The main difference between serverless and traditional cloud computing is that the cloud provider is responsible for managing the scaling of the deployed apps, as well as maintaining the cloud infrastructure. Serverless apps are deployed in containers that automatically launch on demand when called. In this way, apps are launched only when needed, with the possibility of the so called “zero scaling”: no computational resources are allocated if there are no requests.

Serverless encompasses two main models:

1- **BaaS**: initially, serverless referred to the usage of plug-and-play cloud services such as databases, authentication services and so on. These types of services are called Backend as a Service.
2- **FaaS**: serverless can also mean applications where server-side logic is still written by developers, but it’s run in stateless compute containers that are event-triggered, ephemeral and fully managed by the providers. This model is called Function as a Service.

2.2 Function as a Service:

Fundamentally, FaaS is computational model based on events. “Functions” are triggered upon the arrival of predefined events, and they are run on ephemeral containers that are terminated at the end of the execution. For this reason, FaaS is designed mainly for stateless computation. The management of the horizontal scaling, containers' lifecycle and external resources is automatically provided by the infrastructure.

The general architecture of FaaS platforms includes at least the following components:

- **Trigger**: it creates a bridge between external events and those manageable by the FaaS infrastructure. Events can be generated even by sources defined by the provider and HTTP requests via API gateway.
- **Controller**: it associates functions with events received from the trigger. It manages the lifecycle of functions and other components of the FaaS infrastructure. Users provide configurations (called workflows) specifying which functions to activate in response to certain events. The controller is responsible for triggering the workflows.
- **Invoker**: it receives the events from the controller and instantiates the functions with their execution environments to produce the correspondent output.

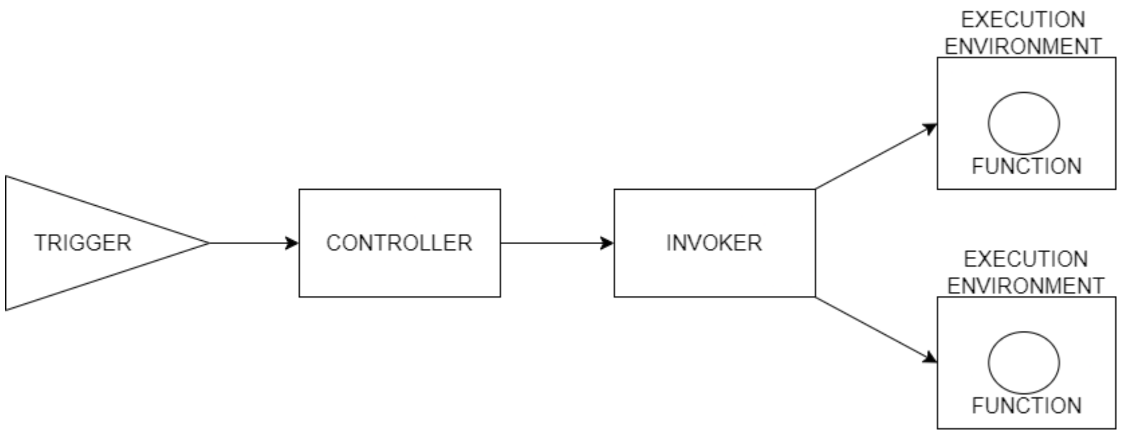


Figure 2.1: FaaS general architecture

Following the classification of the service models described in chapter 1.3, we could place FaaS between the second and third layer, as it is shown in the figure below.

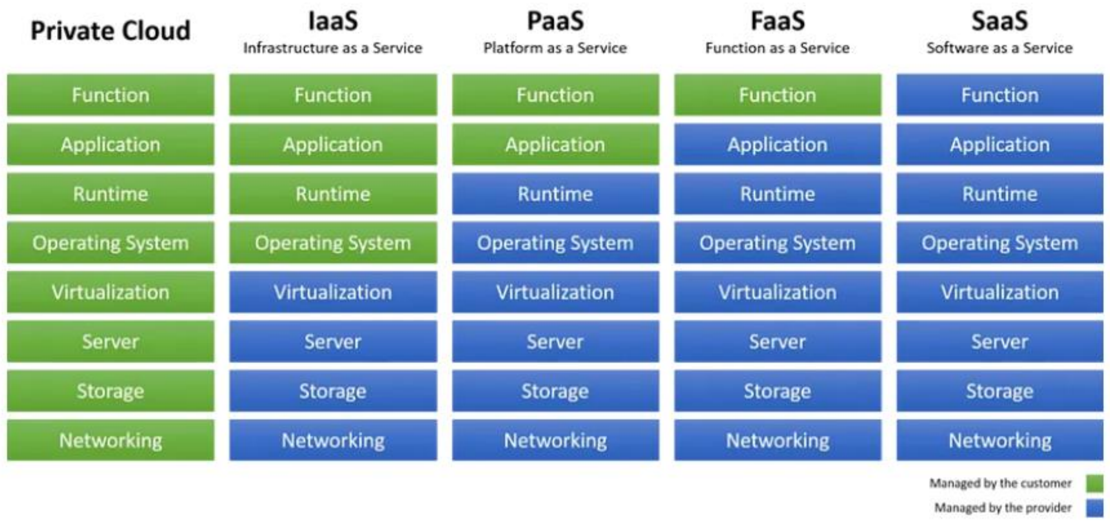


Figure 2.2: FaaS service model

2.3 Benefits and use cases:

Given the characteristics previously described, it is clear that FaaS is mostly suitable for highly asynchronous and stateless scenarios, due to its event driven architecture and the short lifetime of functions. Many use cases include also situations in which there is an unbalanced and infrequent workload, since the scalability of the functions is completely managed by the provider and it's also possible to have no instances running. In fact, the greatest benefit of using FaaS platforms is that we only have to pay for the real time of execution of our functions, instead of paying for the hardware infrastructure even when our application is not running.

Keeping in mind these characteristics, the most common use cases include:

- APIs for web and mobile applications: given that FaaS is mostly suitable for event-driven applications, it is a great solution for RESTful applications. For example, the big majority of commonly used websites already use functions to call server-side APIs or to process user inputs. While containers can also perform these tasks, functions shine when there are high amounts of fluctuating traffic. Serverless APIs are easy to build and maintain and are able to easily scale to meet demand.

- Multimedia and data processing: the straightforward nature of FaaS also allows for easy intake and processing of large amounts of data, meaning that robust data pipelines can be built with little to no maintenance of infrastructure.
- Internet of Things: The Internet of Things refers to devices now common in our homes that connect to the internet to perform functions. These devices are increasingly using FaaS to execute their tasks, only sending and receiving data when triggered by an event. This saves money for businesses since they don't have to pay for computing power they aren't using.

2.4 **Drawbacks**:

Despite the clear advantages brought by the adoption of FaaS model, there are some issues that are intrinsically related to its architecture and its paradigm, such as:

- Vendor lock-in: all serverless vendors implement their features differently from others, so, if you want to switch vendors, you will probably need to modify functions' code.
- Security: since serverless functions consume input data from a variety of event sources, the attack surface considerably increases. Furthermore, our application can be composed of many functions, each one accompanied by its own configuration file. So, functions' configuration gains importance in order to limit the access points to the system.
- No in-server state: we can't assume that the local state from one invocation of a function will be available to another invocation of the same function. In theory, it is possible only if the same instance of the function is kept alive by subsequent requests or events for as long as we need its state. This can be a problem if we need a reliable and fast way to access state. We are forced to use third party services with higher-latency access compared to local caches and on-machine persistence.

Another class of drawbacks concerns the current state of the art of serverless computing. The previous ones are likely always going to exist, and they can only be mitigated. The following issues will probably be resolved in the near future:

- Execution duration: current FaaS platforms allow functions to run only for a certain predetermined period. Afterward, these functions are terminated even if they are still running. So, we have to predict their execution time and ensure that it is under the threshold.

- Cold starts: given that we don't have a server constantly running our code, the platform will need to execute the following steps when a function must be instantiated:
    1) Allocate an underlying VM resource to host the function
    2) Instantiate a Linux container that will run the code on the VM
    3) Copy the code to the container
    4) Start the language Runtime we specified
    5) Load the code
    6) Instantiate the function

  These steps reduce the throughput of our functions.

- Testing: it is quite simple to test a single function because it is in essence a piece of code that doesn't need to use specific libraries or to implement particular interfaces to work. Problems come when we want to perform integration tests on serverless applications. In fact, this kind of application usually leverages externally provided systems to achieve certain features such as persistence and access control. So, it isn't probably enough to perform tests in a local environment to ensure that apps work properly.

2.5 Best practices:

To overcome the drawbacks of adopting FaaS architecture, users' communities and the principal platform vendors have compiled a list of best practices to follow when developing functions. For example, AWS Lambda suggests to:

- Separate handler from core logic to make more unit-testable functions
- Use environment variables to pass operational parameters to the functions
- Write idempotent code to ensure that duplicate events are handled the same way
- Minimize the complexity of the dependencies needed by the functions

In particular, to achieve the last point of the previous list and to simplify the testing phase of our applications' lifecycle it can be useful to resort to the concept of "dependency injection". We will discuss about it and how to implement it in a serverless scenario in the rest of this work, focusing on already existent solutions and open source FaaS platforms. First of all, we will discuss what dependency injection is, how it works, and which benefits it can bring.

3- DISTRIBUTED DEPENDENCY INJECTION

3.1 Introduction:

In general, when we want to develop a new application, we don't simply want to obtain working code, but we want to do it in the most effective, efficient, modular and understandable way possible. To this end, over the years, a series of rules and best practices to follow have been created, which are called "design patterns".

Mark Seemann has defined dependency injection as "a set of software design principles and patterns that enable us to develop loosely coupled code". The idea behind this principle is that we don't want our classes and modules to be dependent on other components by directly instantiating them, but we would like to have them "injected" by someone else. In this way, we can create non-hard-coded dependencies that lead to more flexible and maintainable code. Traditionally, dependency injection is used in object-oriented applications, where the dependencies are classes, and the injector is an object itself. So, the main components needed to implement the pattern are:

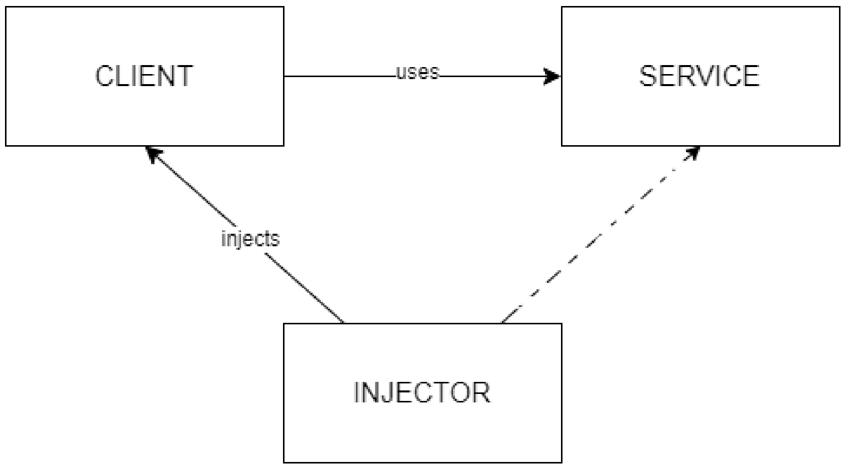


Figure 3.1: the main components involved in dependency injection

- Client: the class that needs dependency. Usually, annotations are used to indicate which dependencies are needed and in which point they can be injected (constructor, setter or another method).

- Service: the class that is needed. It is often the implementation of a more general interface to satisfy the dependency inversion principle. In this way, the client won't have an explicit dependency on the service class.

- Injector: the class that creates the service and injects it into the client. It can be implemented in many languages thanks to the mechanisms they offer, such as reflection and annotations. Other solutions are ready-to-use frameworks and containers such as Spring for the Java world.

The typical workflow of dependency injection mechanism can be described as follow:

1) Scan: the injector scans the class path for classes decorated with predefined annotations that indicate the necessity of dependency injection
2) Wire: the injector inspects constructor and methods for annotations that indicate the points where to execute the injection of the matching dependencies from a component registry.
3) Init: the injector inspects injected components for annotations that indicate methods that must be executed before and/or after their initialization or destruction (these methods are called "lifecycle callbacks").

In distributed and heterogeneous systems, the basic components of applications are services, not objects. In general, every service can be implemented with its own technology and can be located on any node. So, dependency management can be very difficult given that a single service could not have visibility of the entire system. Furthermore, scalability and efficiency constraints are more relevant compared to traditional applications, especially in a serverless scenario in which other problems already affect the performances. In the following chapter we will discuss a starting architecture for the implementation of distributed dependency injection. After some tests on public clouds offers for FaaS computing, we will work on open-source serverless platforms to optimize the performances and see which solution will fit better our expectations.

3.2 Architecture:

Keeping in mind what we have just said and what dependency injection needs to work, a first architecture can be organized as follow:

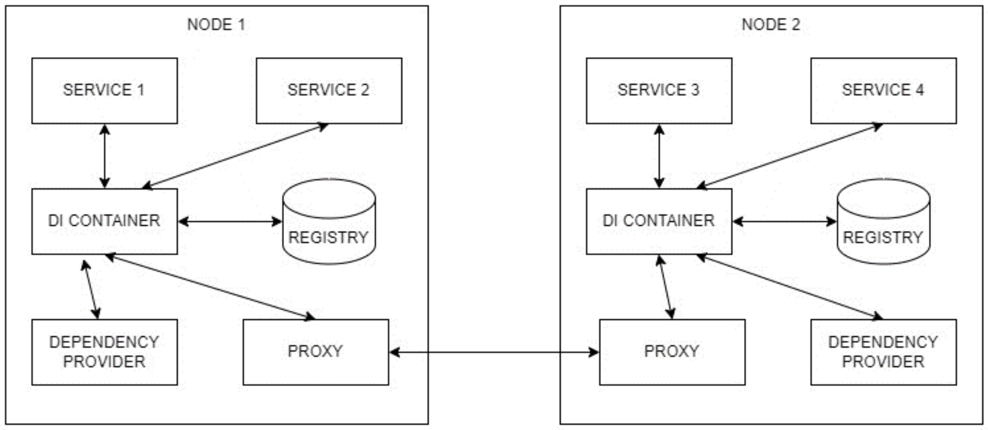


Figure 3.2: first architecture of distributed dependency injection

The main components are:

- Services: the basic components of distributed systems. They can be either microservices, functions, databases, etc.
- DI container: it is the component (another service or a simple SDK) that exposes an API for the registration and retrieve of dependencies and that performs the injection at runtime. Dependencies configurations can be provided by the user during the deployment via configuration file. The injector can read the file during its initialization and store the information in the registry. Another solution could be to statically insert the dependencies into the register in an "offline phase".
- Registry: it provides a centralized repository to store information about dependencies such as identifier, name, type, configuration parameters and location. It is possible to use a single repository for the entire system or one repository for every node/partition.
- Dependency provider: it is the service that must be injected.
- Proxy: it enables transparent inter-node communication to retrieve non-local dependencies.

The typical workflow includes the following steps:

1) First of all, the DI container is initialized and can obtain information on the dependencies to inject by reading a configuration file and store it in the registry.
2) Every time a new dependency provider is initialized, it registers itself by using the API provided by the container.

3) When a service needs a dependency to be injected, it sends a request to the DI container through its API.
4) The DI container will retrieve the dependency from the registry and return it back to the service.

In the following chapters we will focus on how to implement dependency injection in a FaaS scenario, starting with the implementation of a simple first architecture on the main cloud providers.

3.3 First experiments on public clouds:

To get a starting point on which to work and a first idea of the performances of the previous architecture, we disposed some mock functions to deploy and execute on two of the main cloud providers that offer FaaS solutions, AWS and Microsoft Azure. First of all, we will present the main characteristics of these two platforms. Then, we will focus on tests and performance evaluation.

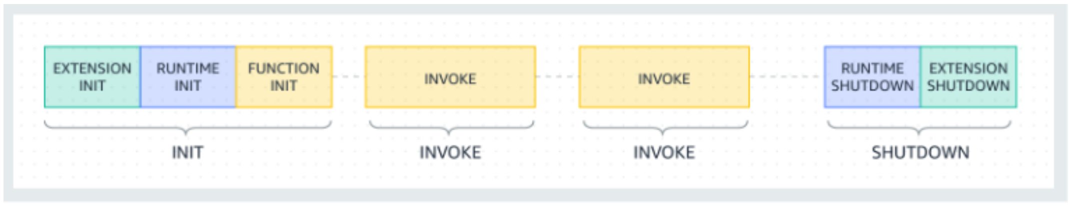
3.3.1 AWS Lambda:

AWS Lambda is a serverless compute service provided by Amazon Web Services (AWS) that allows you to run code without provisioning or managing servers. With AWS Lambda, you can execute code in response to events or triggers such as changes in data, shifts in system state, or user actions. It's highly scalable, automatically handling compute capacity and scaling according to the workload. So, it presents all the characteristics of FaaS presented before. In advance, AWS Lambda is strongly integrated with all the AWS ecosystem, which comprehends, for example, S3 (Amazon Simple Storage Service), DynamoDB (Amazon NoSQL database), API Gateway (to expose services as web APIs) and CloudWatch (logging and monitoring service).

The main components of AWS Lambda are:

- Function: it is the resource that you can invoke to run code and process the incoming events.
- Trigger: it is a resource or configuration that invokes a Lambda function.
- Event: it is a JSON-formatted document that contains data for a Lambda function to process. The runtime converts the event to an object and passes it to the function code compared to the programming language that supports.

- Execution environment: it provides a secure and isolated runtime for Lambda functions. It provides lifecycle support for the function and for any associated extension. For instance, the lifecycle of a Lambda function consists of the following phases, each started by an event generated by the platform:



  1) INIT: Lambda starts all extensions, bootstraps the runtime and runs the function's static code.
  2) INVOKE: Lambda sends an Invoke event to the runtime and to each extension. In case of error, the runtime is shut down and can be reused for another more rapid invocation.
  3) SHUTDOWN: The entire Shutdown phase is capped at 2 seconds. If the runtime or any extension does not respond, Lambda terminates it via a signal (SIGKILL). After the function and all extensions have completed, Lambda maintains the execution environment for some time in anticipation of another function invocation. However, Lambda terminates execution environments every few hours to allow for runtime updates and maintenance (even for functions that are invoked continuously). Reusing the execution environment implies that objects declared outside the function's handler method remain initialized and the content of /tmp remains.
- Deployment package: Lambda functions can be deployed in two ways:
  1) A .zip file archive that contains the code and its dependencies.
  2) A container image that is compatible with the Open Container Initiative specification.
- Runtime: it provides a language-specific environment that runs in an execution environment. The runtime relays invocation events, context information, and responses between Lambda and the function. Lambda offers different runtimes ready to be used.

Lambda provides a programming model that is common to all the runtimes. The programming model defines the interface between code and the Lambda system. It is possible to tell Lambda the entry point to the function by defining a *handler* in the function configuration. The runtime passes in objects to the handler that contain the invocation *event* and the *context*, such as the function name and request ID. Lambda scales the function by running additional instances of it as demand increases, and by stopping instances as demand decreases. This model leads to variations in application architecture, such as:

- Incoming requests might be processed out of order or concurrently.
- There isn't the certainty that an instance of a function will be long lived.
- The size of deployment packages should be the smallest possible.

AWS Lambda offers the possibility to use the service with a free trial account. Of course, it presents some limitations, such as a limited number of function's invocations per month. Another limitation consists of having a maximum of ten instances of a function running at the same time. For our purpose, a free trial account is sufficient.

3.3.2 Azure functions:

Azure Functions is Microsoft Azure's serverless computing service that allows to run small pieces of code (functions) without worrying about infrastructure, much like AWS Lambda, and following the FaaS model. The main differences between the two offers are:

- AWS Lambda relies on various forms of triggers to invoke functions, such as HTTP requests or events generated by other services (for example S3 uploads or changes in DynamoDB tables). Azure offers an additional way to invoke functions: declarative bindings. They allow for easy input/output connections to other Azure services without the need to write code. Bindings can be defined statically in a configuration file inside the deployment package.
- Lambda has a maximum execution timeout of 15 minutes per invocation, while Azure offers support for long-running tasks.
- AWS Lambda automatically scales based on the number of concurrent invocations. Each time a new event triggers the function, Lambda will create a new instance to handle the request. If there are many simultaneous requests, Lambda automatically launches additional instances. Azure Functions also scale automatically based on the number of incoming events or requests, but it offers more plans to manage the scaling at different granularities.

- Azure Functions' free plan doesn't have the limit of ten instances of a function at the same time.

Thanks to the following tests, we will be able to evaluate the performance difference between the two services.

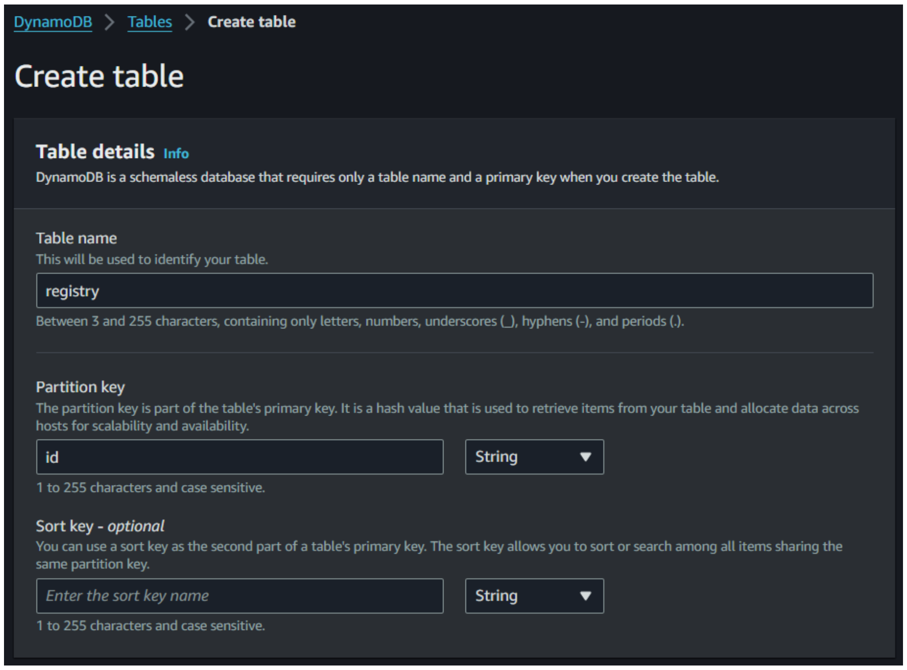
3.3.3 First architecture on AWS Lambda:

Before implementing the architecture presented in [3.2] on a FaaS platform, we need to make a couple of considerations. First of all, the DI container should be always available when a service (a function) needs to use it. A solution could be to implement the container as a function itself. In this way, we could assign an URL to it to make it globally accessible via HTTP protocol. Of course, it would have all the typical drawbacks of the functions, such as cold starts and ephemeral state. To keep it as simple as possible, we decided to implement the injector as a class that exposes methods to perform registration and retrieval of services. Another consideration is about the registry. Given that we are using an AWS tool, it is a good idea to use DynamoDB as storage service because it is already well integrated with Lambda. In fact, AWS offers ready-to-use SDKs for all the programming languages supported by Lambda runtimes.

For the tests, we have chosen to implement the functions in three programming languages (Java, JavaScript and Go) to evaluate the performances with different types of runtimes. In the following paragraphs we will describe all the steps to execute those tests. The only prerequisite is to have created a free trial account on AWS. All the code can be found at:

https://github.com/FabioGentili99/project-work-on-Distributed-Systems-M

First of all, we need to create a DynamoDB table and to initialize it with the services we need to inject. To create it, we need to navigate to its service page and create a new table:


DynamoDB > Tables > Create table

Create table

Table details Info

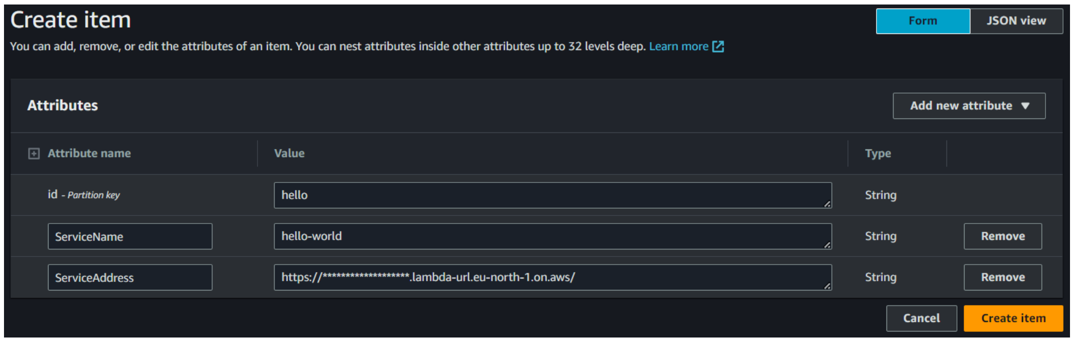
DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.

Table name
This will be used to identify your table.
registry
Between 3 and 255 characters, containing only letters, numbers, underscores (_), hyphens (-), and periods (.).

Partition key
The partition key is part of the table's primary key. It is a hash value that is used to retrieve items from your table and allocate data across hosts for scalability and availability.
id | String
1 to 255 characters and case sensitive.

Sort key - *optional*
You can use a sort key as the second part of a table's primary key. The sort key allows you to sort or search among all items sharing the same partition key.
Enter the sort key name | String
1 to 255 characters and case sensitive.

Once the creation is completed, we can populate it with the services that need to be injected. A service can be represented as a tuple of (at least) two elements, name and address. To insert entries in the table, we need to create new items. Every item is composed of a set of attribute-values couples:


Create item

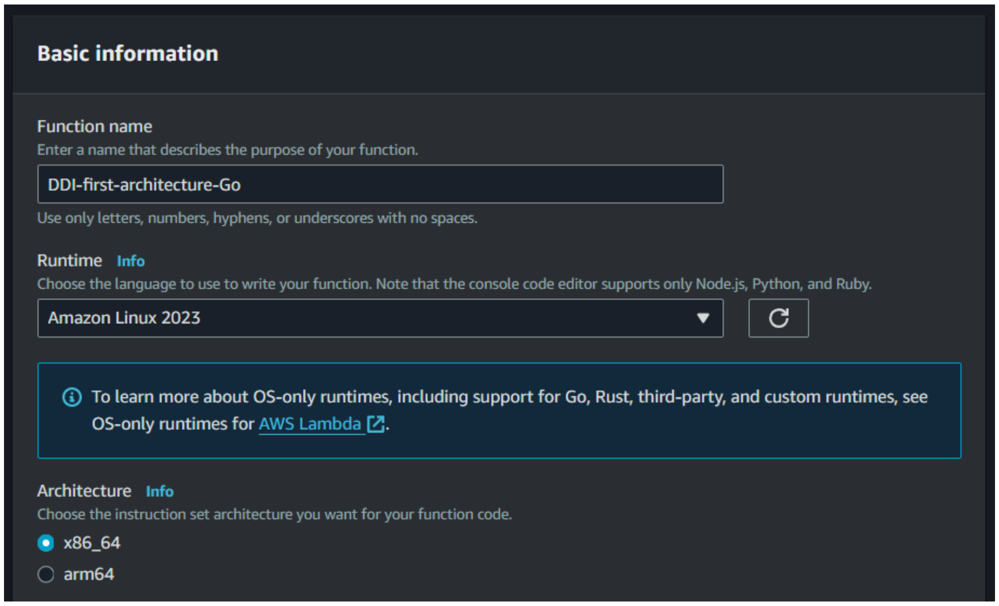
You can add, remove, or edit the attributes of an item. You can nest attributes inside other attributes up to 32 levels deep. Learn more

Attributes

Attribute name	Value	Type
id - *Partition key*	hello	String
ServiceName	hello-world	String
ServiceAddress	https://******************.lambda-url.eu-north-1.on.aws/	String

We initialized the table with an item that contains the URL of a simple hello-world Go function that will be used in the tests as dependency provider.

Now that the registry is set-up, we can pass to the functions. Here we only illustrate the case of the Go function, since the procedure is the same for all the languages. The only difference is in the deployment package’s structure, as we will see. So, the first step is to create a Lambda function. It is possible either to proceed directly form the AWS portal or command line interface (CLI):



Basic information

Function name
Enter a name that describes the purpose of your function.
DDI-first-architecture-Go
Use only letters, numbers, hyphens, or underscores with no spaces.

Runtime Info
Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.
Amazon Linux 2023

To learn more about OS-only runtimes, including support for Go, Rust, third-party, and custom runtimes, see OS-only runtimes for AWS Lambda.

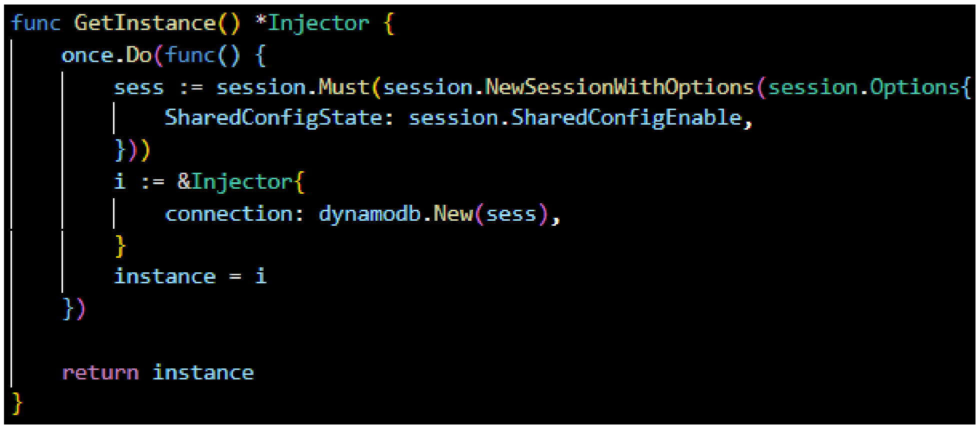
Architecture Info
Choose the instruction set architecture you want for your function code.
[✓ x86_64]
arm64

It is possible to check the option of enabling the function URL, so that Lambda assigns a globally valid URL to invoke the function. Lambda doesn't offer a specific runtime for Go functions and, in general, for all the languages that are compiled to native code or in case we want to define our specific runtime. So, we need to use an OS-only runtime.

The Injector class contains a field that represents the connection to the database. Lambda offers and SDK to interact with DynamoDB, so we just need to import it with:

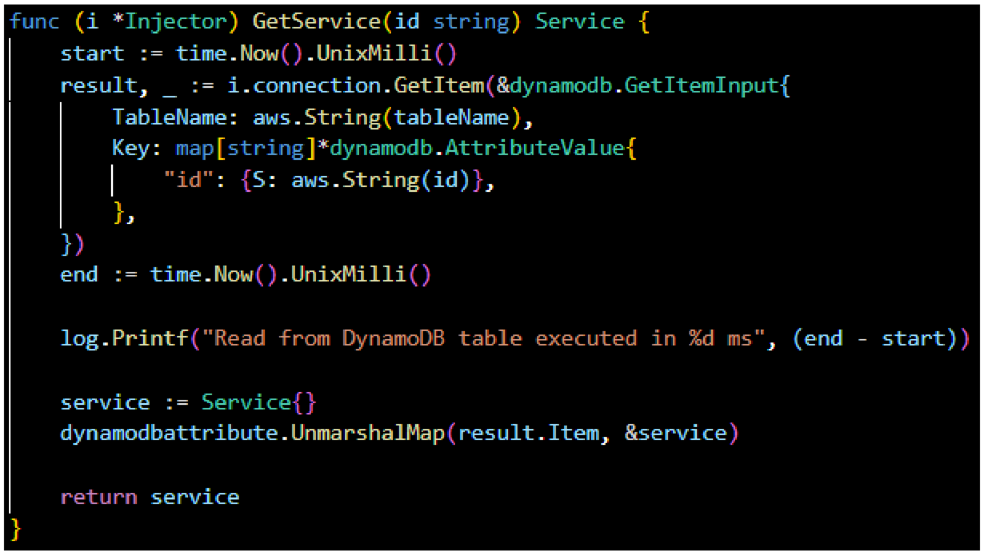
```
go get "github.com/aws/aws-sdk-go/service/dynamodb"
```

We used the singleton pattern to implement the constructor in order to not instantiate a new connection to the DB every time the same instance of the function is invoked.



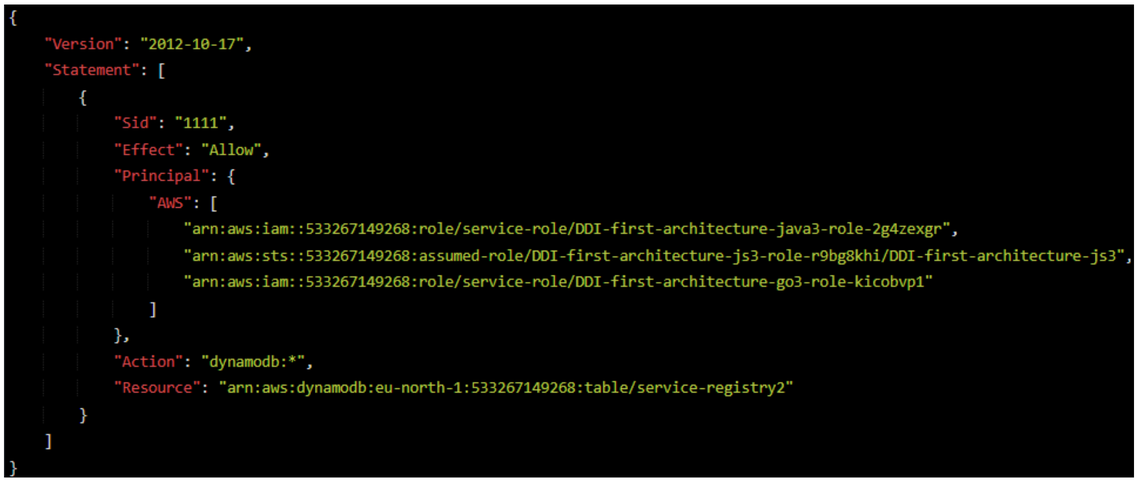
```
func GetInstance() *Injector {
    once.Do(func() {
        sess := session.Must(session.NewSessionWithOptions(session.Options{
            SharedConfigState: session.SharedConfigEnable,
        }))
        i := &Injector{
            connection: dynamodb.New(sess),
        }
        instance = i
    })

    return instance
}
```

The methods that execute the registration and retrieval of services are implemented using the methods `PutItem()` and `GetItem()` on the connection. This last method takes as parameter the identifier of the item that we defined during the creation of the DynamoDB table. During the retrieval, we keep track of the time needed to access the database in order to compare the efficiency compared to CosmosDB. Lambda allows us to log strings and JSON-formatted messages directly to the CloudWatch service in a transparent way.



```go
func (i *Injector) GetService(id string) Service {
    start := time.Now().UnixMilli()
    result, _ := i.connection.GetItem(&dynamodb.GetItemInput{
        TableName: aws.String(tableName),
        Key: map[string]*dynamodb.AttributeValue{
            "id": {S: aws.String(id)},
        },
    })
    end := time.Now().UnixMilli()

    log.Printf("Read from DynamoDB table executed in %d ms", (end - start))

    service := Service{}
    dynamodbattribute.UnmarshalMap(result.Item, &service)

    return service
}
```

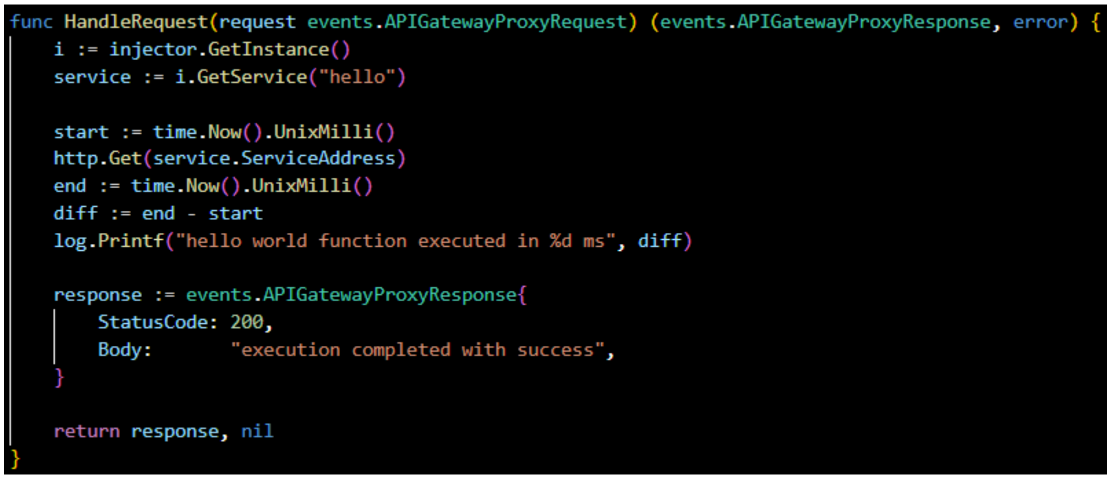
To allow our functions to access the DynamoDB table, we need to add the permissions to it. So, from the AWS portal we have to navigate to the table created previously and create a policy file in JSON format. It will contain the ARN identifier of all the AWS services that can access the table and the relative allowed operations.



```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "1111",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws:iam::533267149268:role/service-role/DDI-first-architecture-java3-role-2g4zexgr",
                    "arn:aws:sts::533267149268:assumed-role/DDI-first-architecture-js3-role-r9bg8khi/DDI-first-architecture-js3",
                    "arn:aws:iam::533267149268:role/service-role/DDI-first-architecture-go3-role-kicobvp1"
                ]
            },
            "Action": "dynamodb:*",
            "Resource": "arn:aws:dynamodb:eu-north-1:533267149268:table/service-registry2"
        }
    ]
}
```

In this way, we are saying that our three functions can perform any action on the table.

The main class of each Lambda function must contain a handler method that will be the one to be invoked when the function is triggered. In this case, we instantiate an Injector object and we call the `GetService()` method passing the identifier of the service that we inserted in the database before. In this way, we obtain a Service object that contains all the attributes of the item. At this point, we can invoke the hello-world Lambda function by sending an HTTP request at its URL. We keep track of the time needed to invoke the function and terminate its execution as we have seen before.

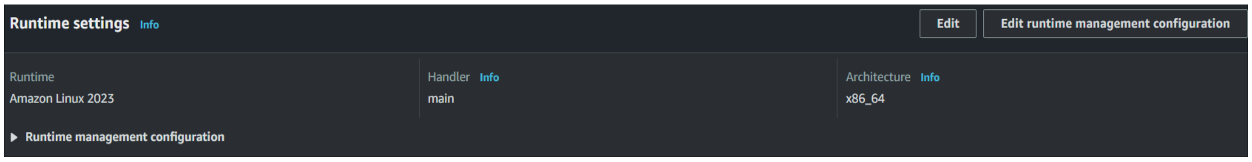


```go
func HandleRequest(request events.APIGatewayProxyRequest) (events.APIGatewayProxyResponse, error) {
    i := injector.GetInstance()
    service := i.GetService("hello")

    start := time.Now().UnixMilli()
    http.Get(service.ServiceAddress)
    end := time.Now().UnixMilli()
    diff := end - start
    log.Printf("hello world function executed in %d ms", diff)

    response := events.APIGatewayProxyResponse{
        StatusCode: 200,
        Body:       "execution completed with success",
    }

    return response, nil
}
```

To deploy the function to Lambda, we need to pack it in a zip file. Each runtime has its own rules on which should be the structure of the zip file in order to execute the code. In the case of Go functions, we need to compile the module containing the handler in a file called “bootstrap”:

```
$env:GOOS = "linux"

$env:GOARCH = "amd64"

go build -o bootstrap main.go
```

Then, we can put the compiled file in a zip archive and upload it to Lambda in the home page of the function or through CLI. The last step is to configure the runtime declaring which is the module that implements the handler. In this way, every time the function is triggered, Lambda knows which is the entry point to invoke the business logic.


Runtime settings Info

Edit | Edit runtime management configuration

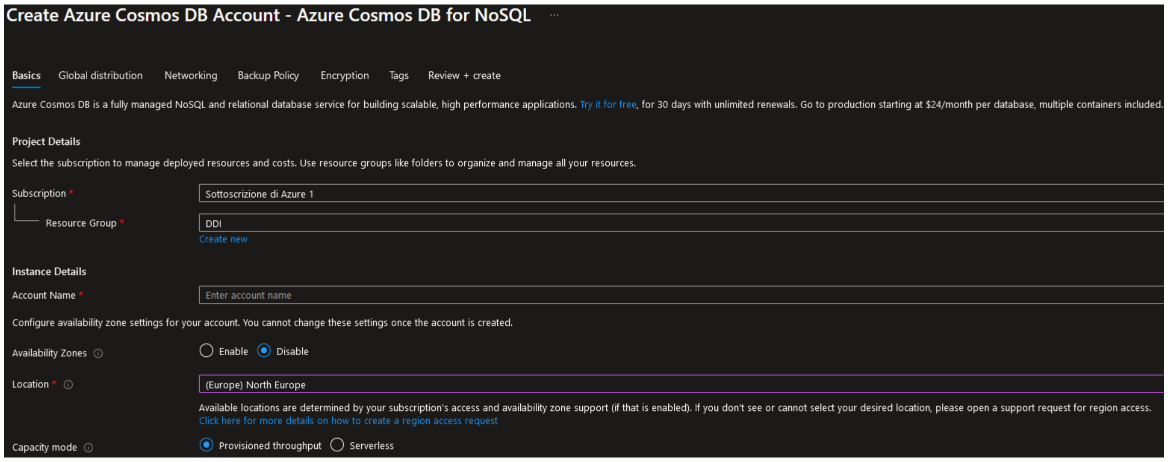
Runtime	Handler	Architecture
Amazon Linux 2023	main	x86_64

▶ Runtime management configuration

We can find the function’s URL in the overview section.

3.3.4 First architecture on Azure Functions:

We followed the same procedure to create a Go function on Azure, but with some differences. First of all, we need to create a CosmosDB table directly from the Azure portal.


Create Azure Cosmos DB Account - Azure Cosmos DB for NoSQL

Basics | Global distribution | Networking | Backup Policy | Encryption | Tags | Review + create

Azure Cosmos DB is a fully managed NoSQL and relational database service for building scalable, high performance applications. Try it for free, for 30 days with unlimited renewals. Go to production starting at $24/month per database, multiple containers included.

Project Details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *: Sottoscrizione di Azure 1

Resource Group *: DDI

Create new

Instance Details

Account Name *: Enter account name

Configure availability zone settings for your account. You cannot change these settings once the account is created.

Availability Zones: Enable [✓ Disable]

Location *: (Europe) North Europe

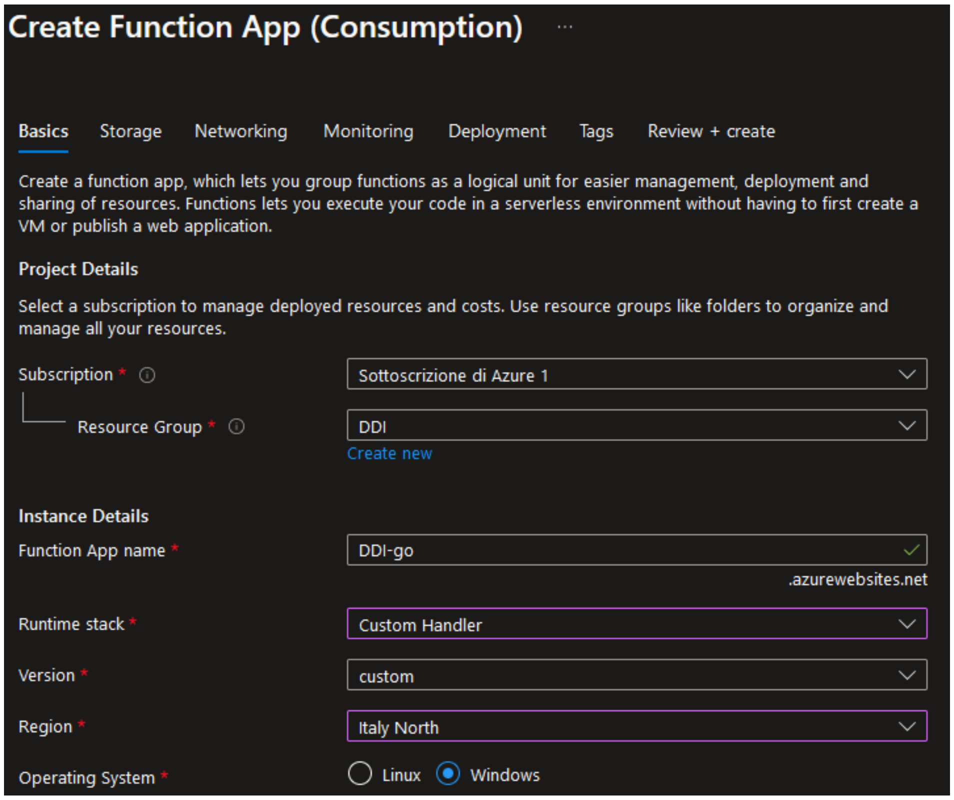
Available locations are determined by your subscription's access and availability zone support (if that is enabled). If you don't see or cannot select your desired location, please open a support request for region access.
Click here for more details on how to create a region access request

Capacity mode: [✓ Provisioned throughput] Serverless

Once the table is created, we can create a container inside it with the id parameter as partition kay, and initialize it with an item containing the id, name and address of a simple hello-world Go function to inject. It is possible to insert new items defining them with the JSON syntax.

Azure services are well integrated with Visual Studio Code. In fact, with the Azure Functions extension, it is possible to create, locally test and easily deploy our functions. So, we use it to develop our tests. As we have done for Lambda, we will present the procedure for the Go function, given that it is almost the same for all the runtimes.

The first step is to create an Azure Function App from the portal. It is a sort of “container”, since that it can contain more than one function, that can host their execution.


Create Function App (Consumption)

Create a function app, which lets you group functions as a logical unit for easier management, deployment and sharing of resources. Functions lets you execute your code in a serverless environment without having to first create a VM or publish a web application.

Project Details

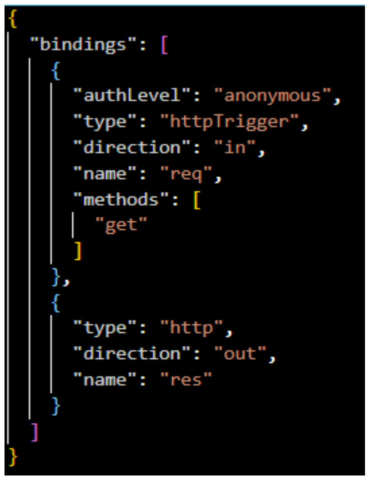
Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *	Sottoscrizione di Azure 1
Resource Group *	DDI
	Create new

Instance Details

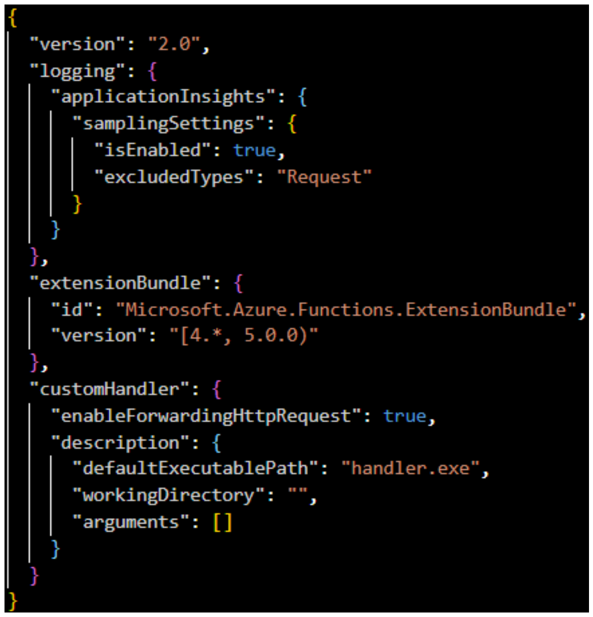
Function App name *	DDI-go
	.azurewebsites.net
Runtime stack *	Custom Handler
Version *	custom
Region *	Italy North
Operating System *	Linux [✓ Windows]

In the “Networking” section, we need to enable public access in order to assign a public URL to the functions and, in “Monitoring”, we can enable Application Insights to log our information. At this point, we can focus on the local development of the function. After creating a directory for the project, we can open Visual Studio and select the command `Azure Functions: Create New Project`. We need to select Custom Handler as runtime for the same reason as for Lambda. Then we select HTTP trigger as template for the project and we insert the function’s name. Some JSON files will be automatically generated. In particular, in `function.json` we can define the input/output bindings: in our case, the input is composed by an HTTP GET request without authorization needed and the output is a HTTP response.



```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": [
        "get"
      ]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "res"
    }
  ]
}
```

The `host.json` metadata file contains configuration options that affect all functions in a function app instance, like extensions and logging parameters. In our case, we enable logging to Application Insight and we define the path of the executable file for the custom handler. It is a `.exe` executable file because we have chosen Windows as operating system for the function app.



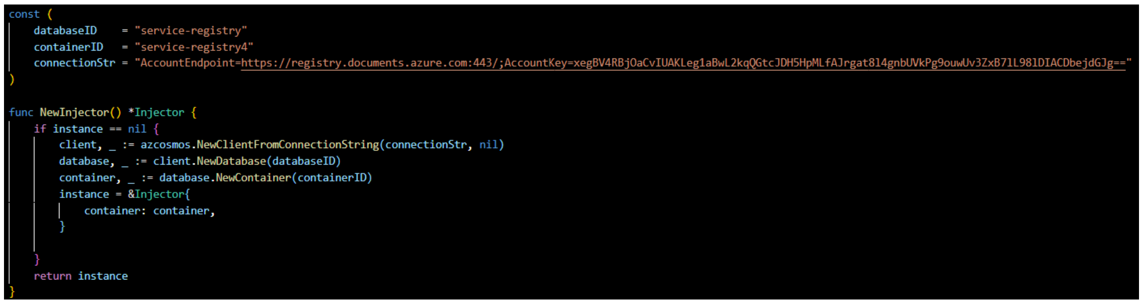
```
{
  "version": "2.0",
  "logging": {
    "applicationInsights": {
      "samplingSettings": {
        "isEnabled": true,
        "excludedTypes": "Request"
      }
    }
  },
  "extensionBundle": {
    "id": "Microsoft.Azure.Functions.ExtensionBundle",
    "version": "[4.*, 5.0.0)"
  },
  "customHandler": {
    "enableForwardingHttpRequest": true,
    "description": {
      "defaultExecutablePath": "handler.exe",
      "workingDirectory": "",
      "arguments": []
    }
  }
}
```

In `local.settings.json` it is possible to define all the configuration options used by local development tools. For example, we can define environment variables, customize the host process for local execution and define the origins allowed for cross-origin resource sharing (CORS).

After setting up the JSON configuration files, we can start coding the Injector module. As we have seen for AWS Lambda, Azure Functions provides an SDK to work with CosmosDB. To use it, we need to run the command

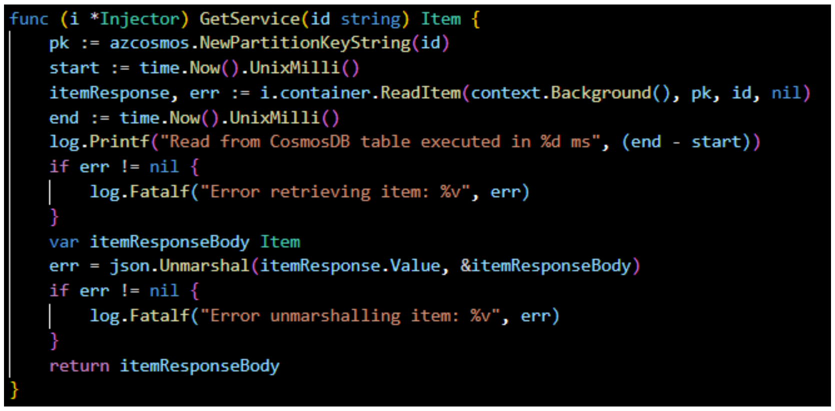
```
go get "github.com/Azure/azure-sdk-for-go/sdk/data/azcosmos"
```

To create a connection to the Cosmos container, we need to instantiate a client passing a connection string as parameter. The connection string can be found on the Azure portal, in the “Keys” section of the database we have created before.



```go
const (
    databaseID    = "service-registry"
    containerID   = "service-registry4"
    connectionStr = "AccountEndpoint=https://registry.documents.azure.com:443/;AccountKey=xegBV4RBjOaCvIUAKLeg1aBwL2kqQGtcJDH5HpMLfAJrgat814gnbUVkPg9ouwUv3ZxB71L981DIACDbejdGJg=="
)

func NewInjector() *Injector {
    if instance == nil {
        client, _ := azcosmos.NewClientFromConnectionString(connectionStr, nil)
        database, _ := client.NewDatabase(databaseID)
        container, _ := database.NewContainer(containerID)
        instance = &Injector{
            container: container,
        }

    }
    return instance
}
```

We can easily implement service registration and retrieval logic using `CreateItem()` and `ReadItem()` methods of the class `ContainerClient`. We need to marshal and unmarshal items to and from JSON format, since that is the data format on CosmosDB.



```go
func (i *Injector) GetService(id string) Item {
    pk := azcosmos.NewPartitionKeyString(id)
    start := time.Now().UnixMilli()
    itemResponse, err := i.container.ReadItem(context.Background(), pk, id, nil)
    end := time.Now().UnixMilli()
    log.Printf("Read from CosmosDB table executed in %d ms", (end - start))
    if err != nil {
        log.Fatalf("Error retrieving item: %v", err)
    }
    var itemResponseBody Item
    err = json.Unmarshal(itemResponse.Value, &itemResponseBody)
    if err != nil {
        log.Fatalf("Error unmarshalling item: %v", err)
    }
    return itemResponseBody
}
```

For the handler class, there are some differences compared to AWS Lambda. In fact, a custom handler must be implemented as a web server that keeps listening on the port assigned by the Function app.

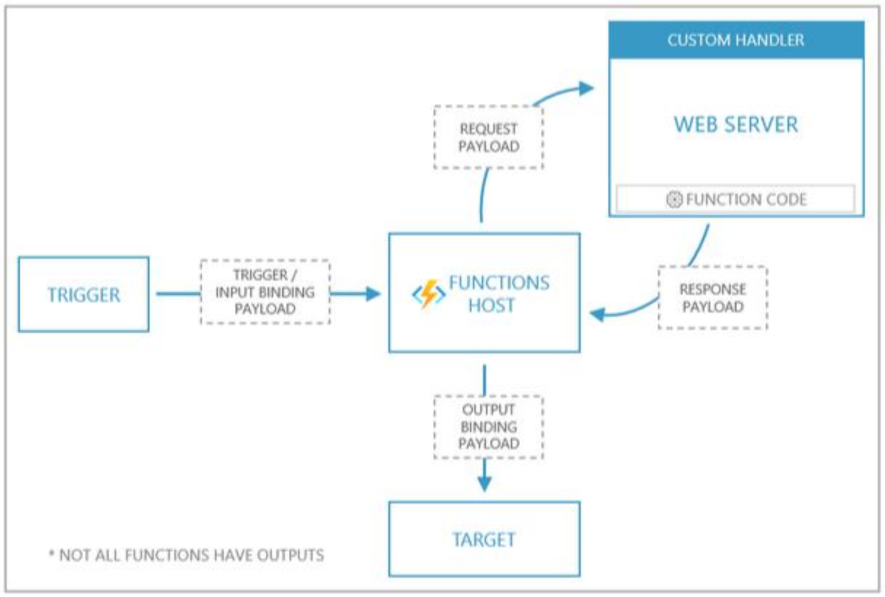
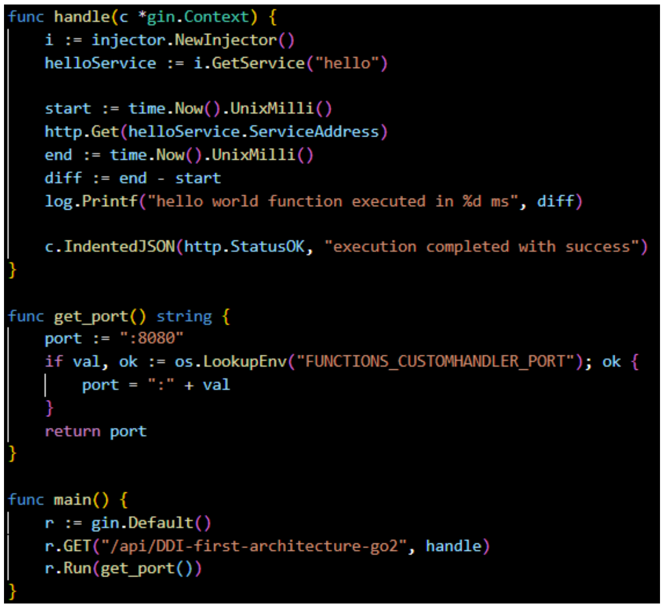


Figure 3.3: Azure Functions custom handler workflow

So, Function app host passes the requests to the web server, which will invoke the right function in response to it. We implemented the web server using the gin-gonic library. Function app assigns the port to the custom handler process when it is started and it is stored in an environmental variable, so we can get it inside the code looking up FUNCTIONS_CUSTOMHANDLER_PORT value. The business logic code is the same as for the Lambda Go function that we have seen before.



```go
func handle(c *gin.Context) {
    i := injector.NewInjector()
    helloService := i.GetService("hello")

    start := time.Now().UnixMilli()
    http.Get(helloService.ServiceAddress)
    end := time.Now().UnixMilli()
    diff := end - start
    log.Printf("hello world function executed in %d ms", diff)

    c.IndentedJSON(http.StatusOK, "execution completed with success")
}

func get_port() string {
    port := ":8080"
    if val, ok := os.LookupEnv("FUNCTIONS_CUSTOMHANDLER_PORT"); ok {
        port = ":" + val
    }
    return port
}

func main() {
    r := gin.Default()
    r.GET("/api/DDI-first-architecture-go2", handle)
    r.Run(get_port())
}
```

It is possible to compile the function running the command:

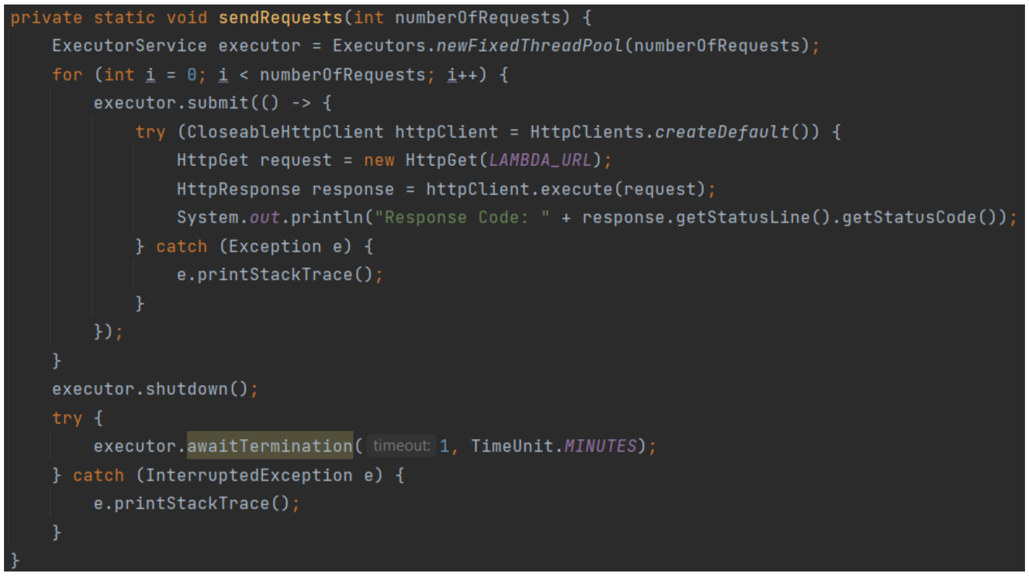
```
$env:GOOS = "windows"

$env:GOARCH = "amd64"

go build handler.go
```

In this way, we can generate an executable file named `handler.exe`, as we declared in `host.json`. We can easily deploy the project to the Azure app using the Visual Studio Code extension, without zipping it.

3.3.5 Stress testing:

To test the performance of the distributed dependency injection architecture on the two different platforms, we disposed a stress test to recreate a typical workload on the functions. It simulates a situation of constant number of HTTP requests per second which is followed by a gradual increase until a peak is reached. In this way, we can test how the scalability is managed either in a standard situation as in a critic one, and which are the access times of DynamoDB and CosmosDB.

The stresser is implemented in Java using the `java.util.concurrent` library to instantiate parallel threads that can create more requests at the same time, and the `org.apache.http` library to send those requests.



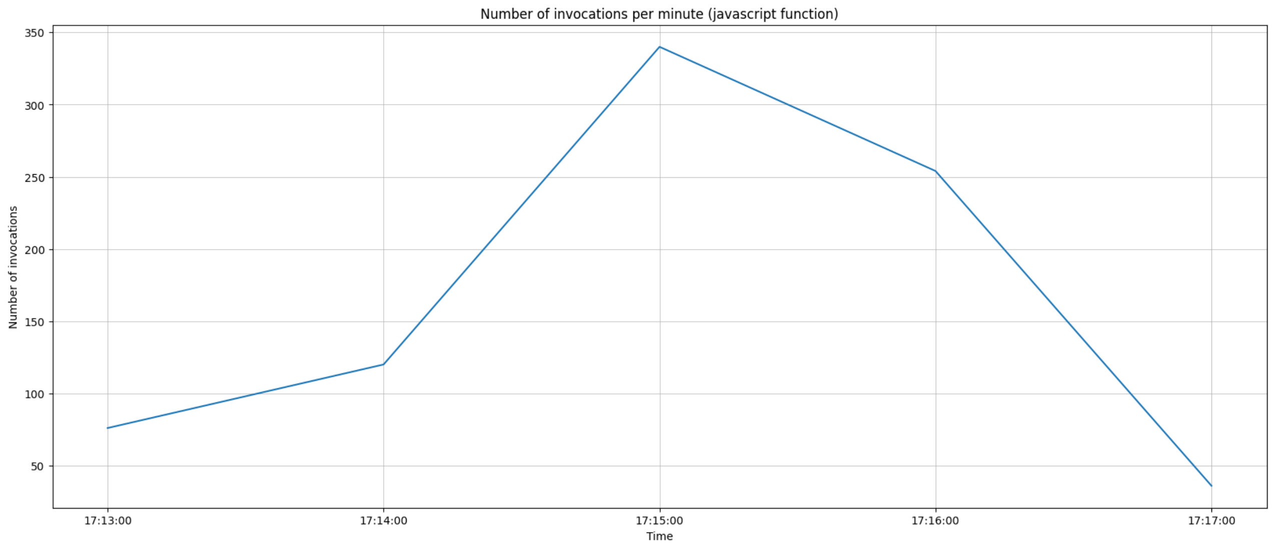
```java
private static void sendRequests(int numberOfRequests) {
    ExecutorService executor = Executors.newFixedThreadPool(numberOfRequests);
    for (int i = 0; i < numberOfRequests; i++) {
        executor.submit(() -> {
            try (CloseableHttpClient httpClient = HttpClients.createDefault()) {
                HttpGet request = new HttpGet(LAMBDA_URL);
                HttpResponse response = httpClient.execute(request);
                System.out.println("Response Code: " + response.getStatusLine().getStatusCode());
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
    }
    executor.shutdown();
    try {
        executor.awaitTermination( timeout: 1, TimeUnit.MINUTES);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

3.3.6 Results on AWS Lambda:

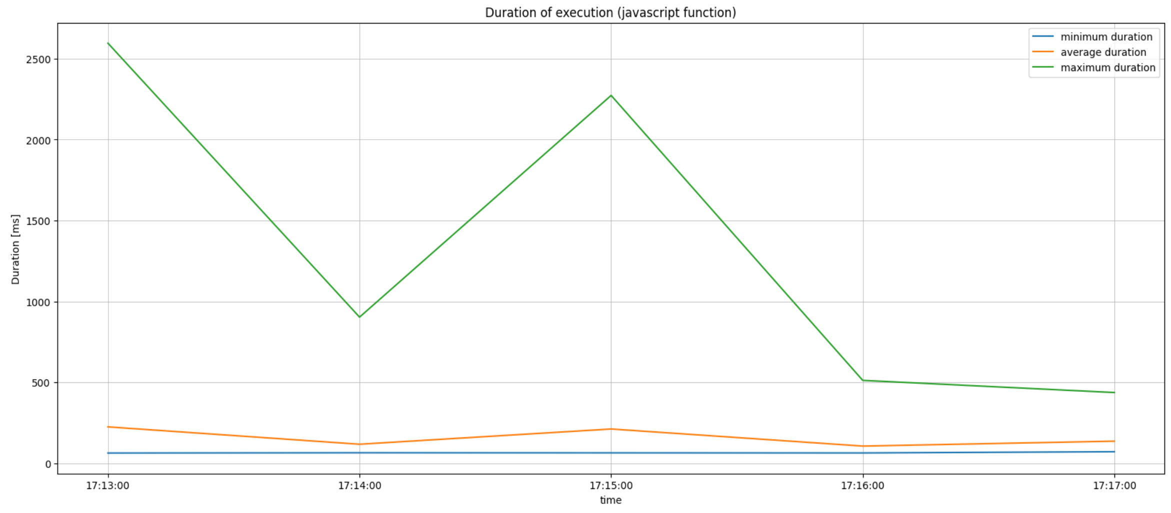
After executing the stresser passing the URL of our three functions, it is possible to collect metrics (number of invocations, duration, logs, etc.) and export them as `.csv` files using the CloudWatch service. Starting from the `.csv` files, we created the following graphics using the `matplotlib` Python library. We decided to keep track of the number of invocations per minute, the execution duration and the throttles, since that the free trial plan of AWS Lambda doesn't allow to have more than ten instances of the same function running simultaneously.

Javascript function:

- Invocations: the invocation scheme is similar to all the functions, and it consists of an initial phase in which the requests are constant in time, followed by a peak phase and a descendent one.

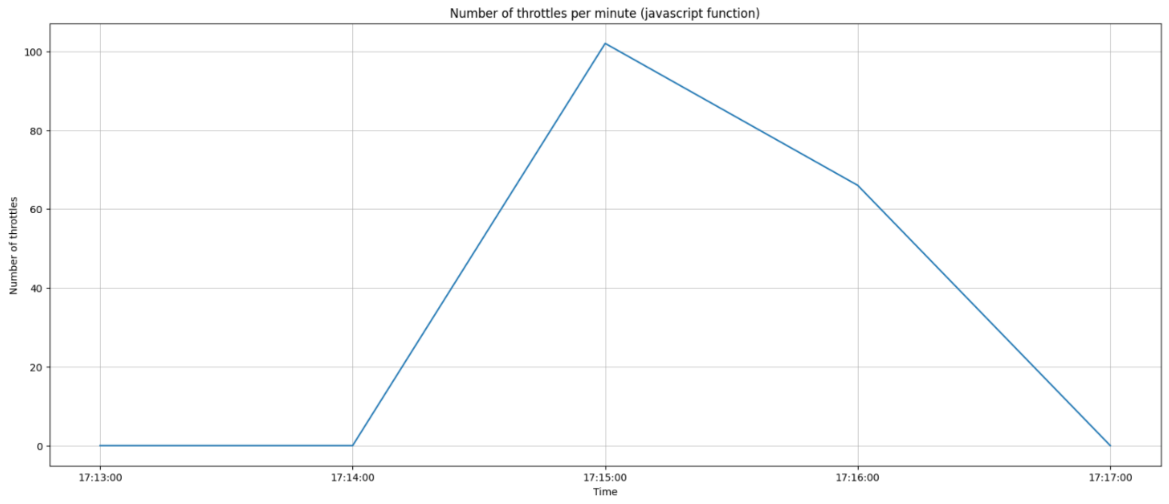


- Duration: the duration metric is divided in three sub-metrics to keep track of the maximum, minimum and average function duration in every minute. In this way, we can see the effect of slow starts on the global duration.

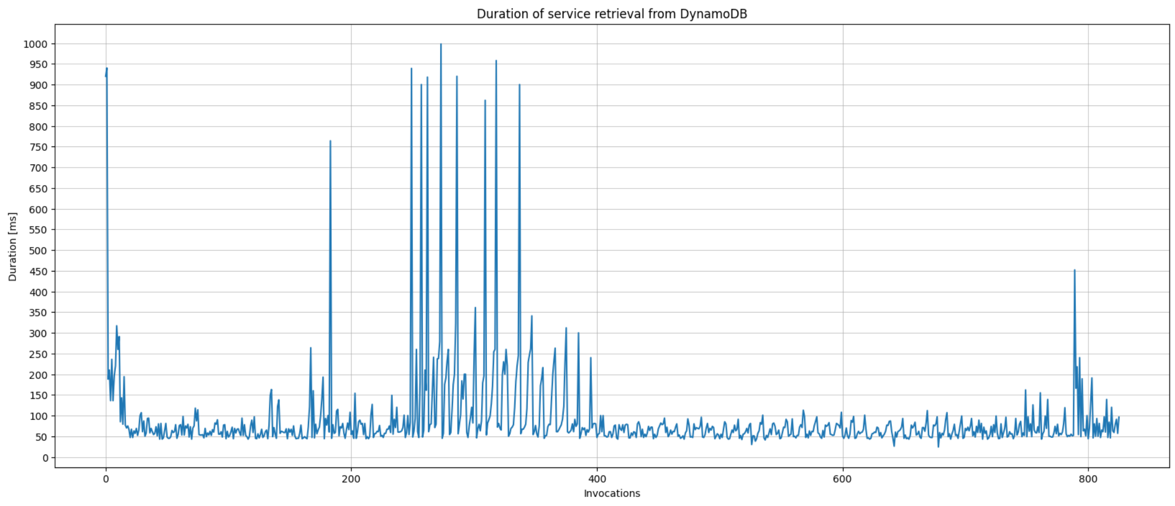


As we can see, we have a peak of function duration both at the start of the test and when we reach the peak of requests per minute. This means that AWS Lambda had to start new function instances to deal with the incremented workload. Nevertheless, the average duration ranges between 225 ms and 105 ms, with minimums of 65 ms.

- Throttles: as we said before, we can only have a maximum of ten instances running at the same time, either because we are using a free tier account or because we set a threshold to limit the resource usage. So, it is possible that the threshold isn't enough to afford all the incoming requests. In this case, AWS Lambda's runtimes discard the excess requests and reply with a `ThrottlingException`.

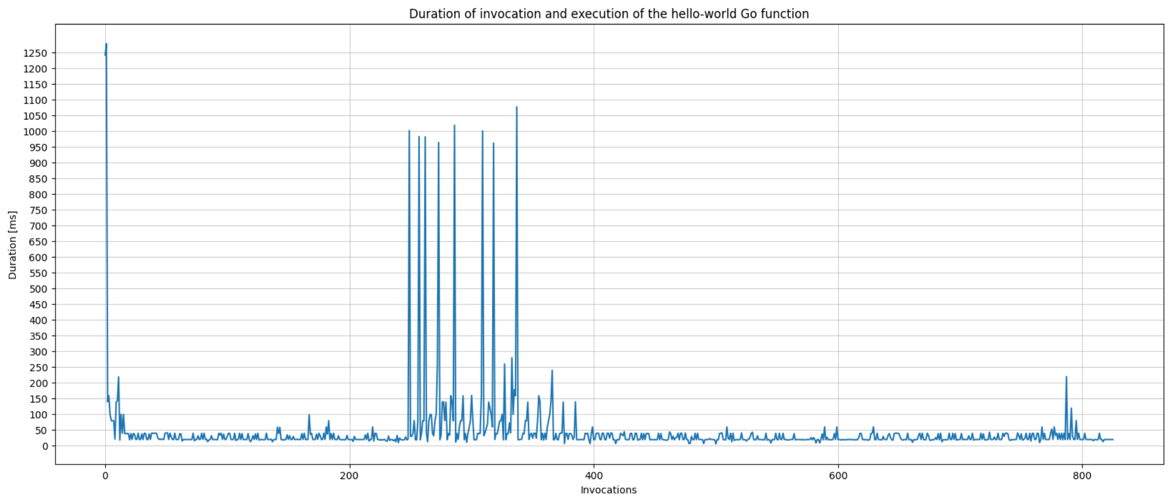


- DynamoDB access duration: given that we implemented the SDK with the singleton pattern, a new connection to the database is created only the first time the function is invoked. So, there will be only one new connection for every instance of the function.



As we can see, there are ten peaks of almost 1 s in correspondence of the creation of the ten function instances. Nevertheless, most accesses to DynamoDB took between 50 ms and 100 ms.

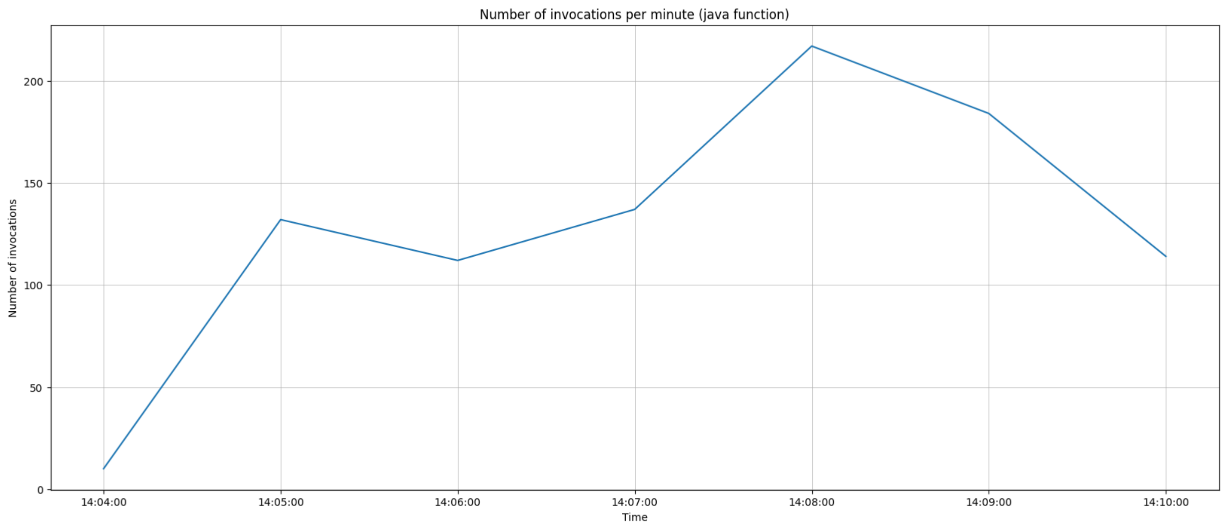
- Service binding duration: we have also kept track of the time needed to invoke and execute the hello world Go function injected by our SDK.



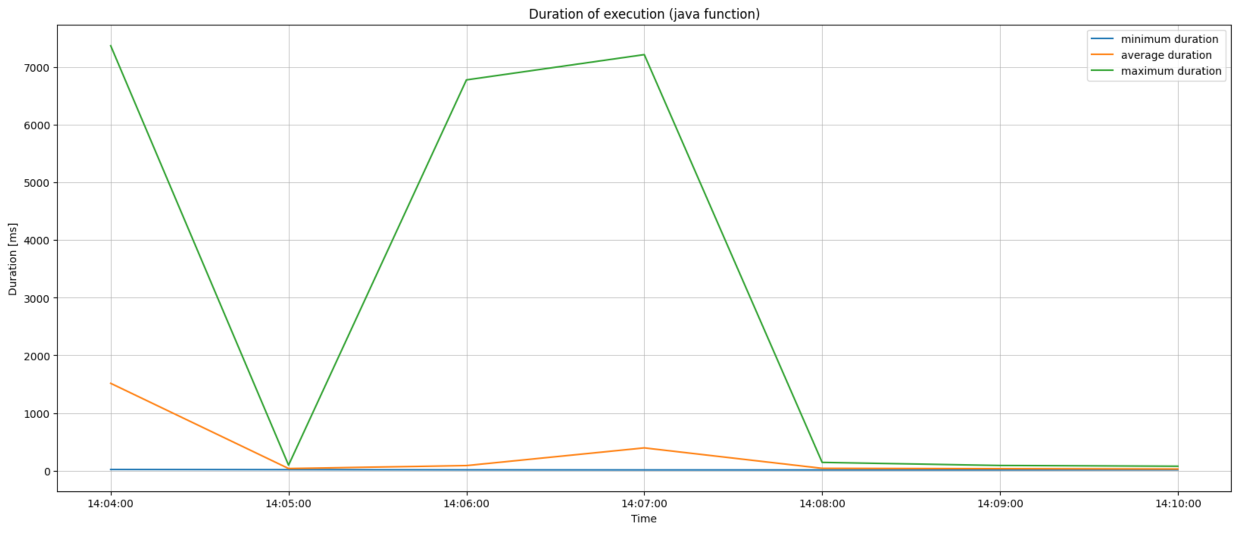
In this case, the ten peaks are caused by the slow starts of the hello-world function instances.

Java function:

- Invocations:

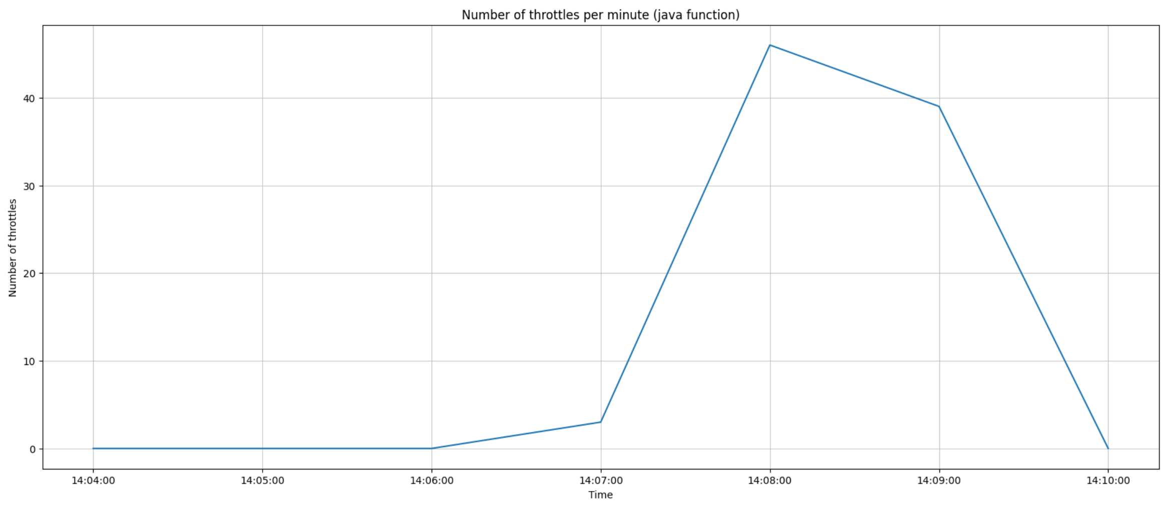


- Duration:



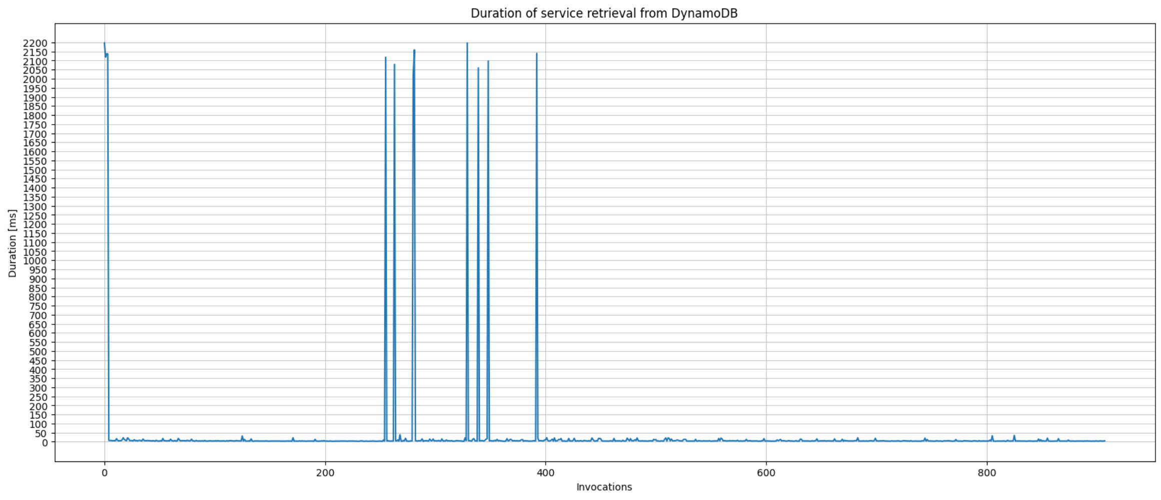
In this case, we have peaks of over 7 s of duration, with minimums of barely 12 ms. This means that cold starts affect Java functions in a worst way compared to Javascript functions, but they are faster in other cases.

- Throttles:



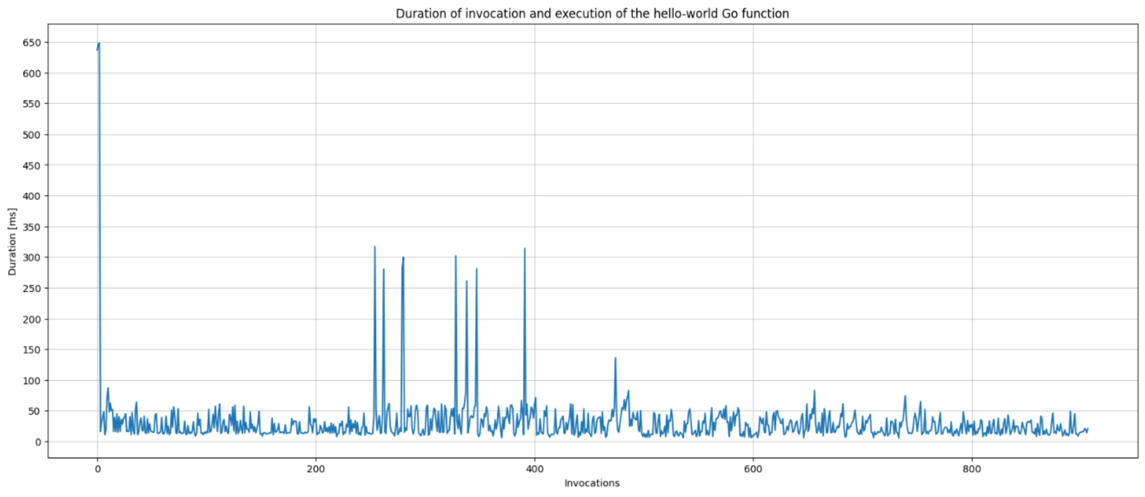
Throttles are less compared to the Javascript function. This means that AWS Lambda was able to manage in a better way the incoming requests with the limit of ten instances. In other words, the Java SDK is faster than the Javascript one.

- DynamoDB access duration:



Despite the fact that it takes more time to create a connection to DynamoDB with the Java SDK, the average time to access data is less than 10 ms.

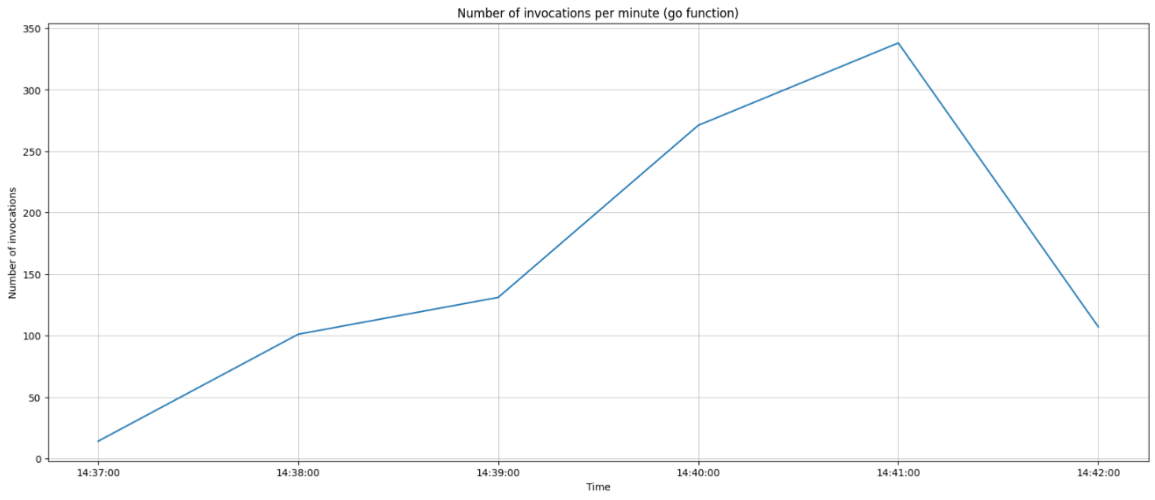
- Service binding duration:



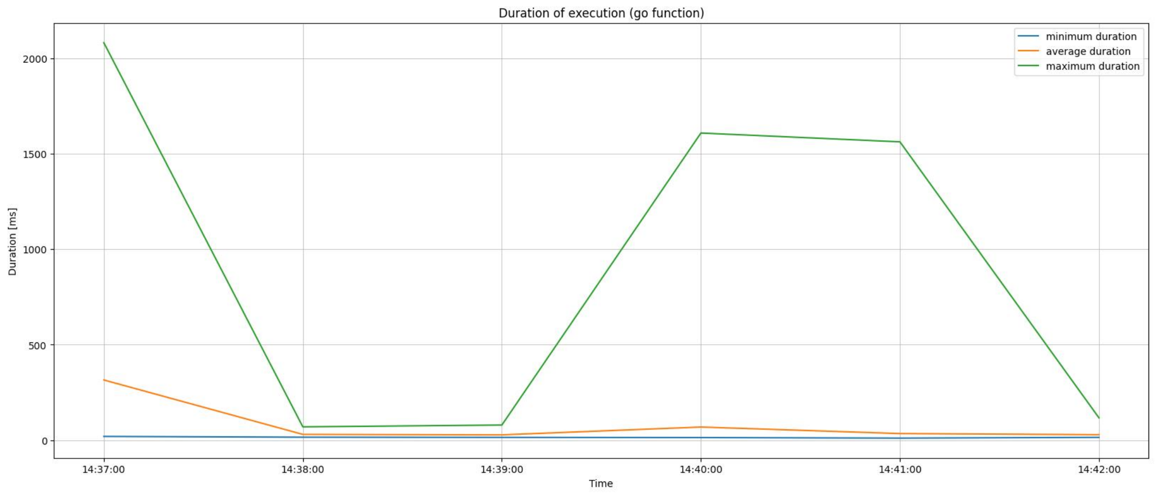
The binding is considerably faster than the previous case, probably because the `fetch()` method takes more time to send the HTTP request and to store the response.

Go function:

- Invocations:

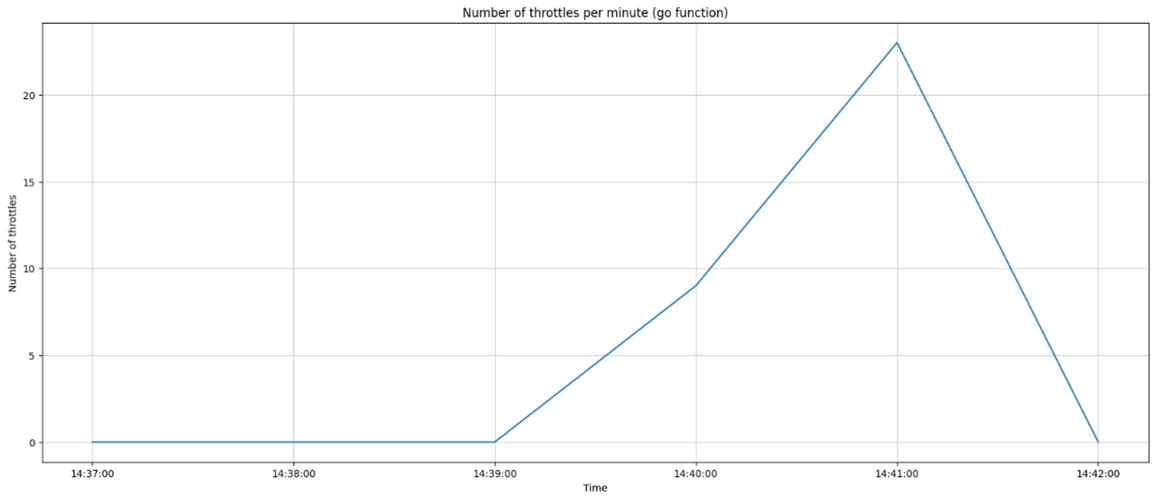


- Duration:



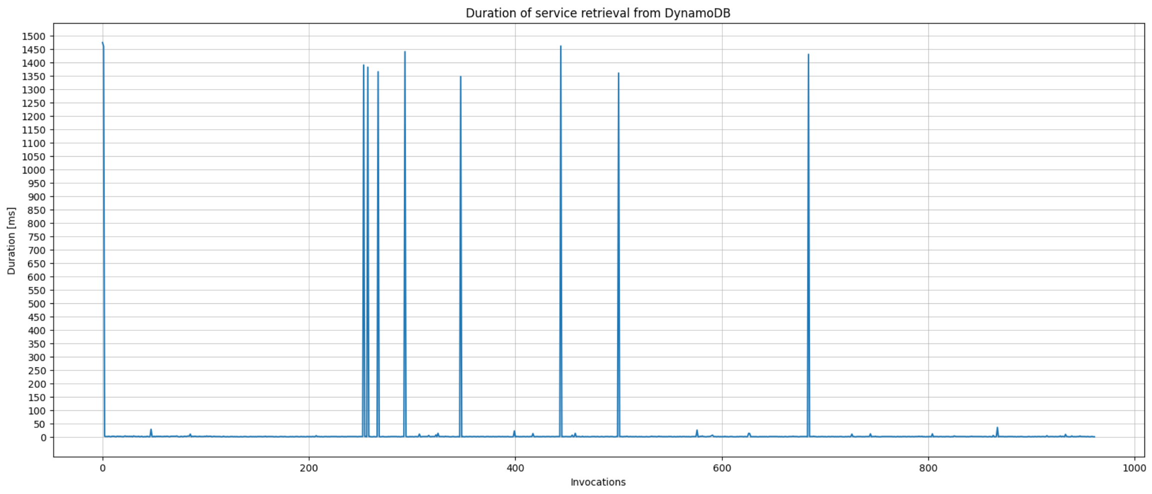
In this case, we have the fastest cold starts and the best average duration, while the minimums are comparable to the Java function’s ones.

- Throttles:



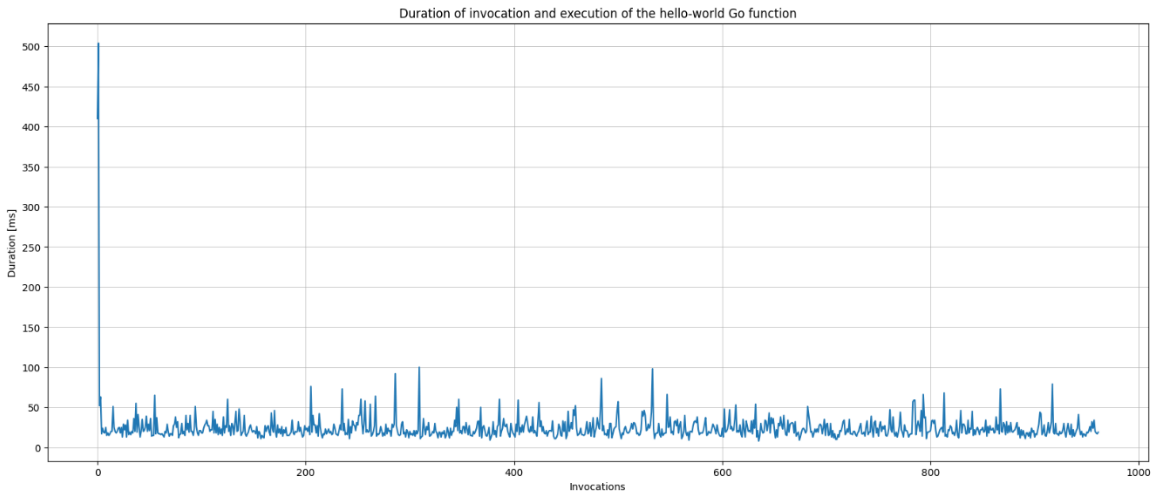
We have the smallest number of throttles, so we can state that the Go function had the best performance on this test.

- DynamoDB access duration:



The creation of new connection is slower compared to the Javascript case, but accesses on existing connections takes less than 5 ms to complete in most of the invocations.

- Service binding duration:



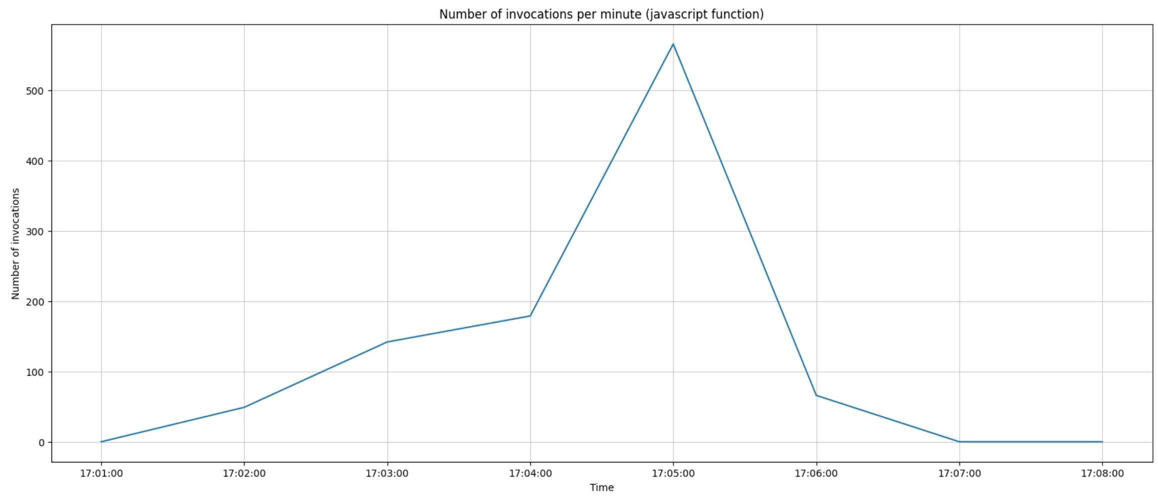
It took only two instances of the hello-world function to serve all the incoming requests.

3.3.7 Results on Azure Functions:

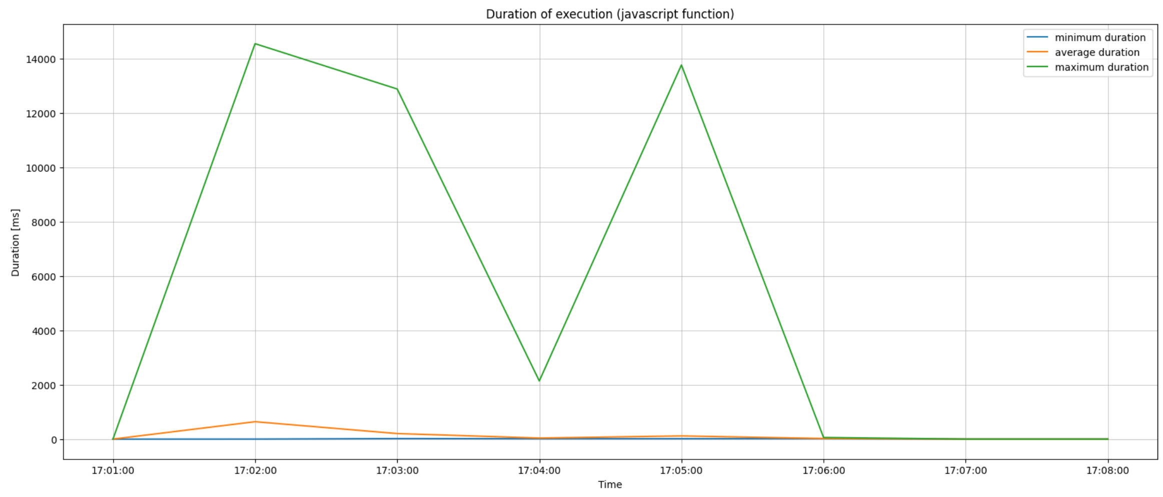
It is possible to create graphics based on many available metrics and export them in `.xlsx` format through Application Insight service. In this case we don't track the throttles because there isn't a limit in the concurrent execution of functions.

Javascript function:

- Invocations:

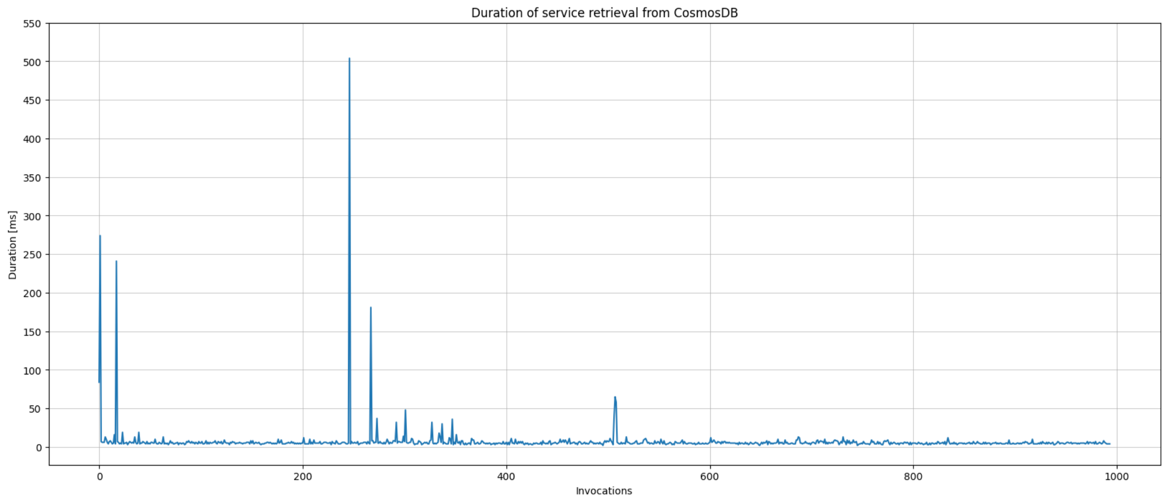


- Duration:



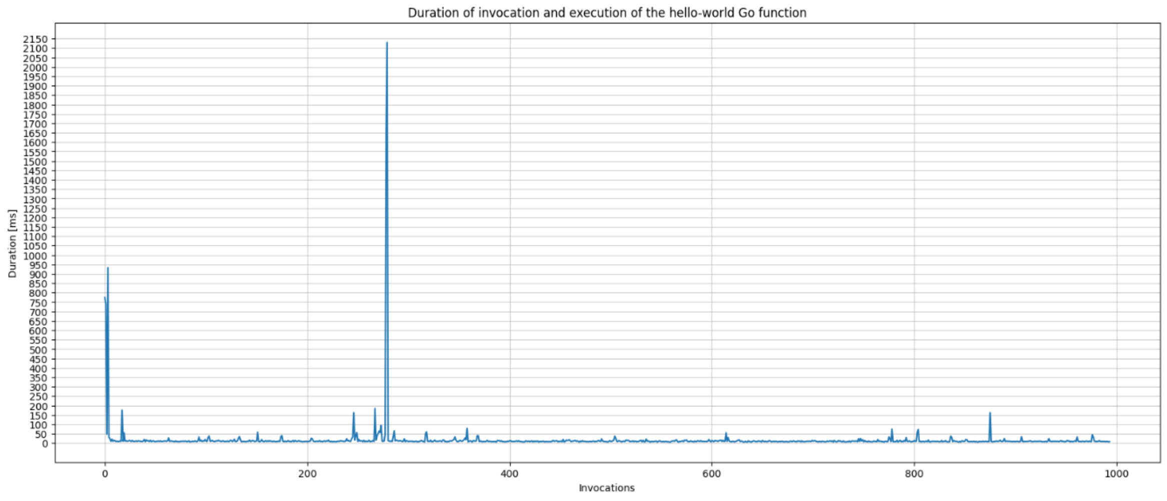
Cold starts cause big execution times (more than 14 s) for the Javascript function, but the minimums are stable around 15 ms.

- CosmosDB access duration:



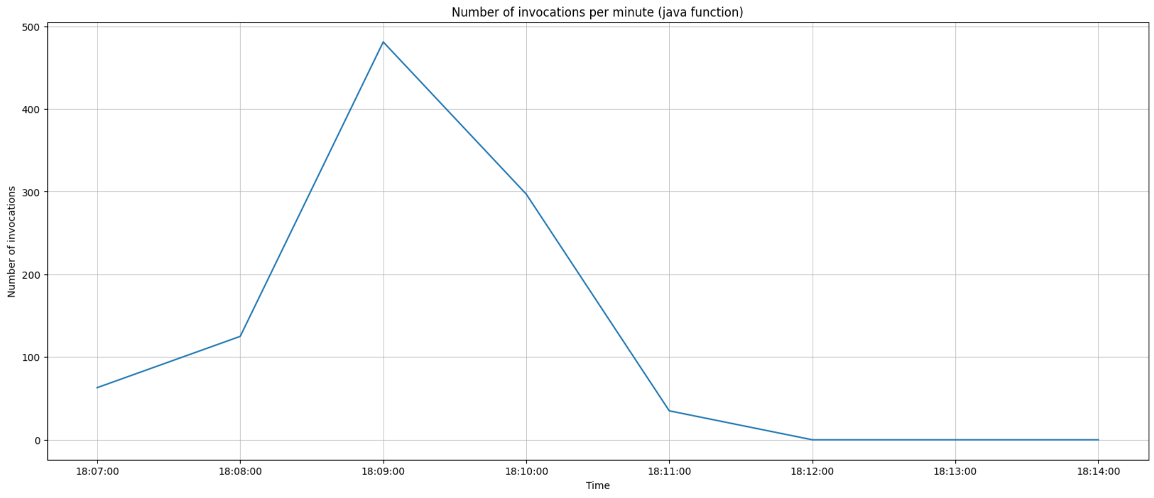
Accesses to CosmosDB are considerably faster than the ones to DynamoDB. Furthermore, we can notice that only five connections were created during the execution of the stresser. This means that only five instances of the function were started to serve the requests, differently from the AWS Lambda case.

- Service binding duration:

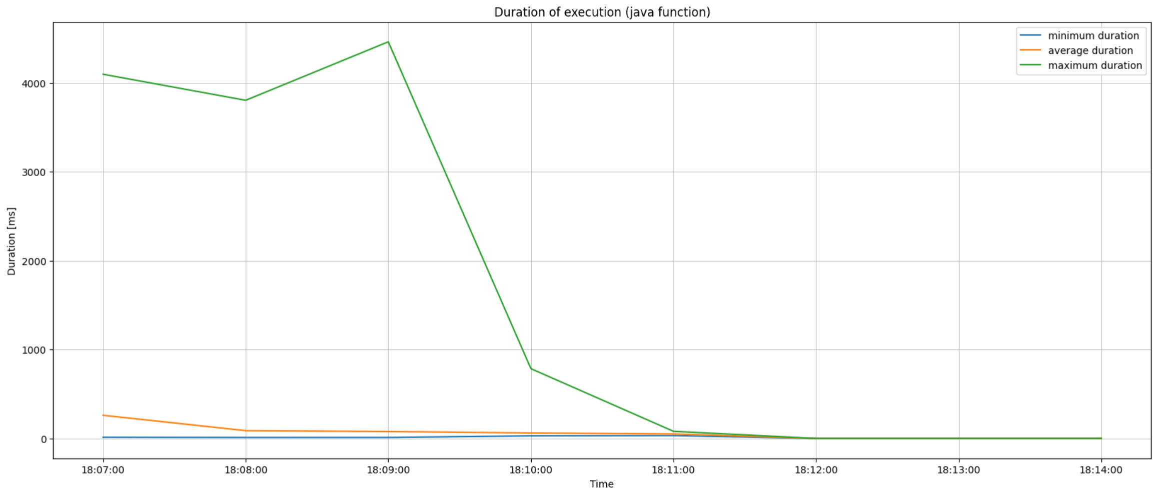


Java function:

- Invocations:

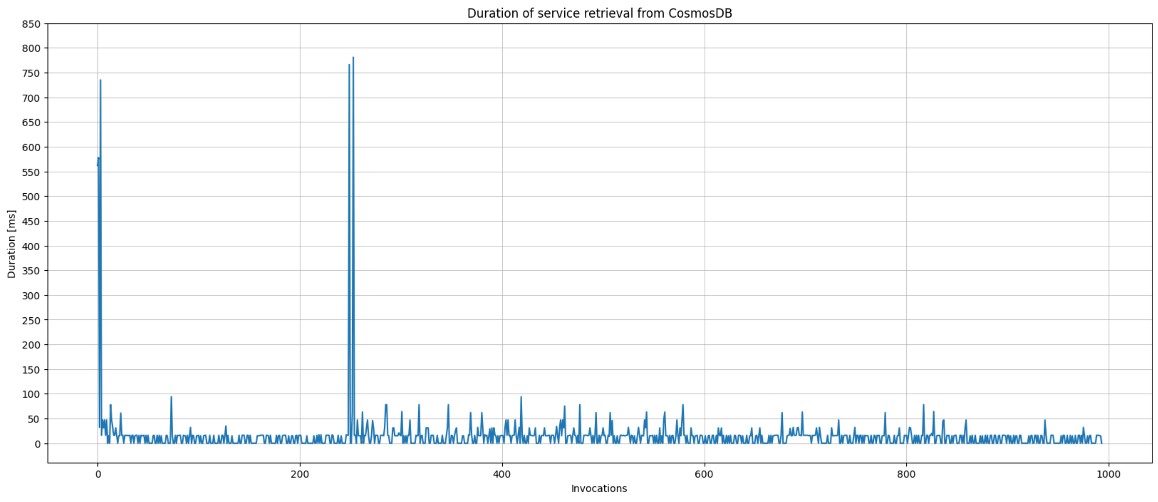


- Duration:



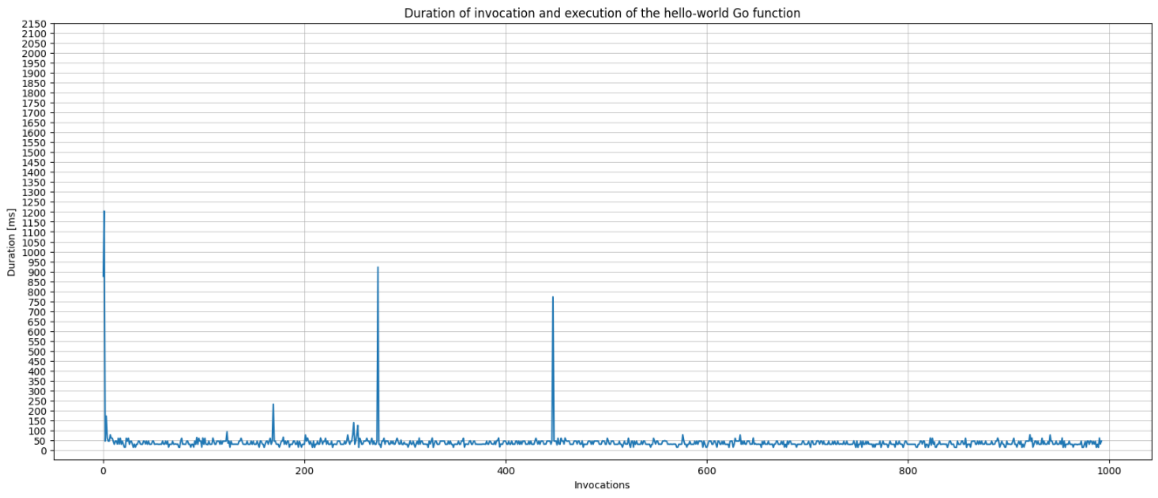
In this case the cold starts are faster than the ones of AWS Lambda and minimums range between 10 ms and 30 ms.

- CosmosDB access duration:



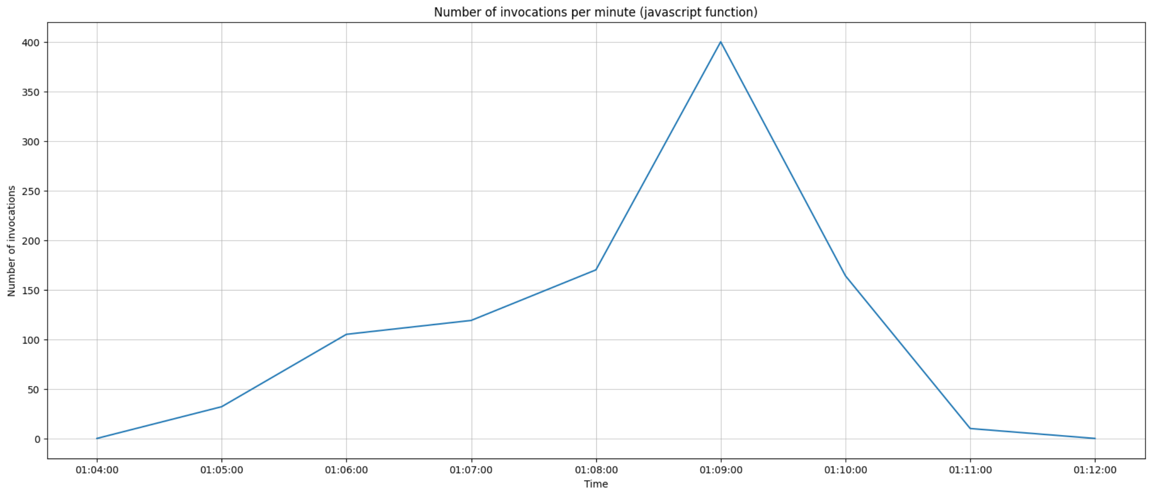
Creating new connections to CosmosDB takes less time compared to DynamoDB. Furthermore, there are minimums of less than 1 ms when accessing the database. Also in this case, only five connections were initialized.

- Service binding duration:

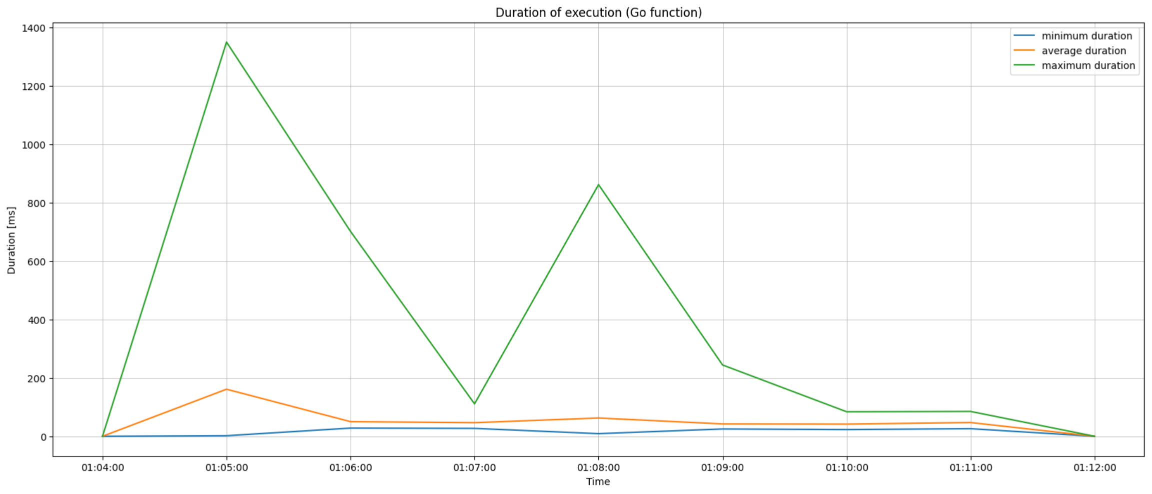


Go function:

- Invocations:

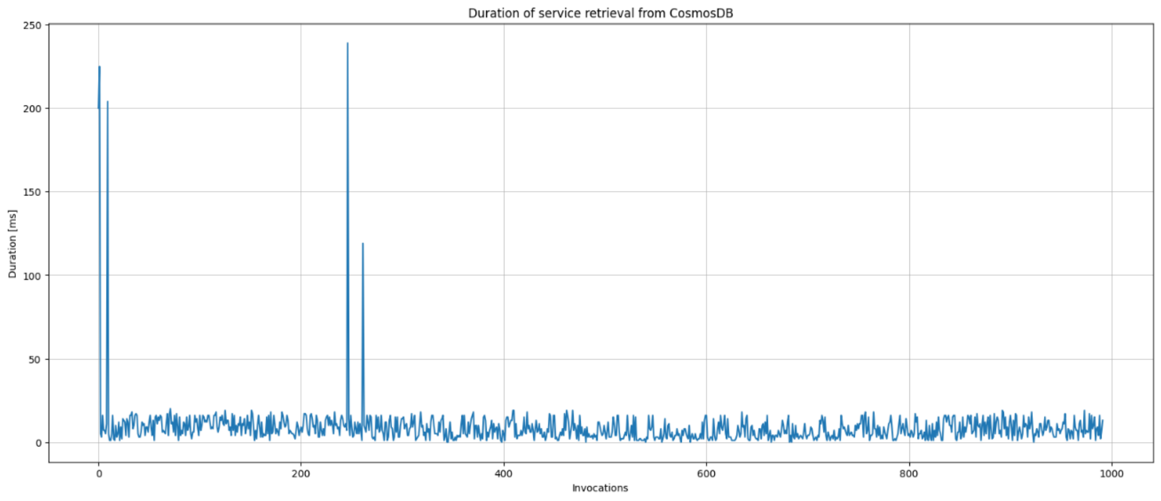


- Duration:



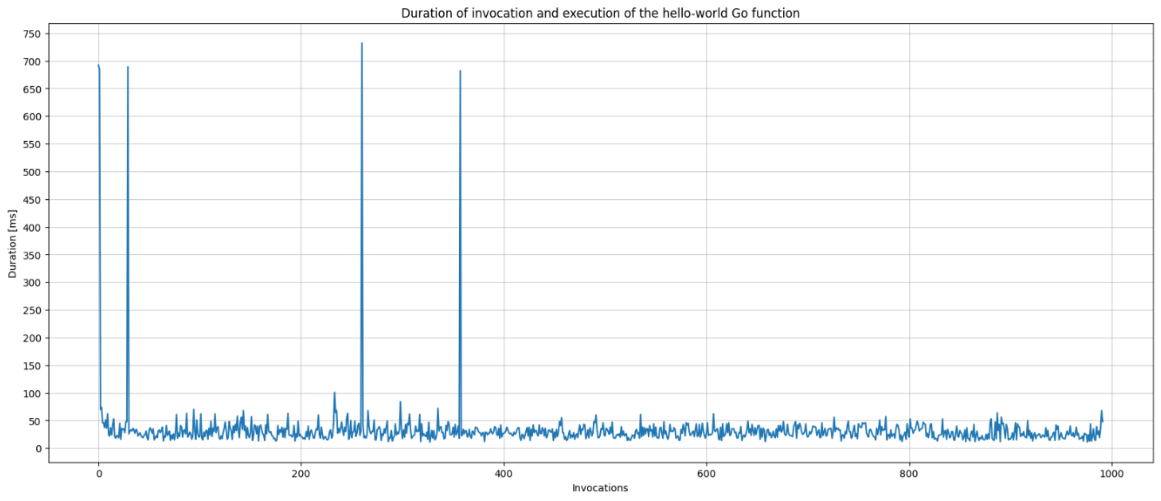
Even on Azure, Go functions have the fastest time of initialization and minimums between 2 ms and 28 ms.

- CosmosDB access duration:



Here we have the fastest creation of connections to the database, while the standard accesses range between 0 ms and 20 ms.

- Service binding duration:



3.3.8 Conclusions:

From the tests we have just presented, it is evident that our architecture performs better on Azure functions, both in terms of average latency and resource usage. In fact, while on AWS Lambda ten instances per function weren't enough to serve all the requests, on Azure it only took five instances per function to avoid throttles. It is also evident that the autoscaling algorithms work in different ways because we can notice that, while on AWS Lambda only two instances of the functions are created at the beginning of the execution and the others when the request peak is near, on Azure the third instance is created almost immediately. Furthermore, the CosmosDB's SDK performed better both in creating new connections and reading the data compared to the DynamoDB's one. Service binding duration is similar on both the platforms, with slightly bigger peaks on Azure caused by cold starts.

References:

https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf

T. Erl et al., “Cloud computing : concepts, technology, & architecture”, Prentice Hall, 2013.

https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless

https://martinfowler.com/articles/serverless.html

https://blog.symphonia.io/posts/2017-11-14_learning-lambda-part-8

https://docs.aws.amazon.com/lambda/latest/dg/best-practices.html

M. Seemann, “Dependency injection in .NET”

https://builtin.com/articles/dependency-injection

https://docs.aws.amazon.com/lambda/

https://learn.microsoft.com/en-us/azure/azure-functions/