Project work on Distributed Systems M

DISTRIBUTED DEPENDENCY INJECTION IN SERVERLESS SCENARIOS

Author:

Fabio Gentili

Abstract:

This report investigates the application of Distributed Dependency Injection (DDI) in serverless environments, focusing on platforms such as AWS Lambda and Azure Functions. This study adapts traditional dependency injection (DI) patterns to distributed systems, addressing the challenges of managing dependencies in a stateless and scalable cloud environment.

The proposed architecture involves an injector service that retrieves and manages dependencies via cloud-based NoSQL databases like DynamoDB and CosmosDB. The injector is implemented in multiple programming languages (Java, JavaScript, and Go), enabling performance evaluation across different runtimes. Stress tests simulate typical workloads, measuring latency, resource usage, and performance across both cloud platforms.

Results show that Azure Functions outperformed AWS Lambda in handling requests with fewer instances, demonstrating better scalability and resource efficiency. However, both platforms exhibited latency peaks during cold starts. The study concludes by recommending future work on on-premises solutions for greater control over the infrastructure, which public cloud providers do not offer.

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.

The most successful incarnation of serverless is the Function as a Service model. 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 horizontal scaling, containers’ lifecycle and external resources is automatically provided by the infrastructure. All the most famous public cloud vendors offer FaaS solutions on a pay per use basis (usually based on execution time).

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.

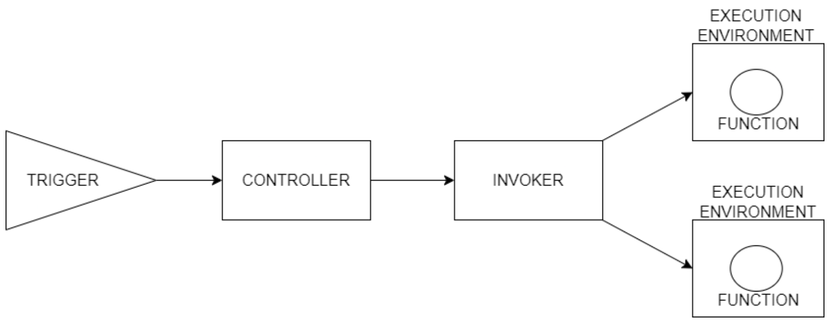

Figure 1: FaaS general architecture

As all other technologies, FaaS model presents many drawbacks as, for example:

- 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.
- Cold starts: given that we don’t have a server constantly running our code, the platform will need to execute some preliminary operations when a function must be instantiated. This increases the execution time.
- 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.

To overcome the drawbacks of adopting FaaS architecture, it is possible to follow some “best practices”, such as writing idempotent code to ensure that duplicate events are handled the same way and minimizing the complexity of the dependencies needed by the functions. Especially for this last point, it could be useful to exploit a famous pattern belonging to the object-oriented world: the dependency injection pattern. The aim of this work is to study how to transpose this pattern to a distributed and heterogeneous environment such as FaaS systems. So, after proposing an initial implementable architecture for distributed dependency injection, we will set up some test functions on the main public cloud vendors’ platforms to evaluate the performance of our solution.

2 BACKGROUND

2.1 Dependency injection:

Dependency injection can be defined 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. The main components involved in dependency injection mechanism are three:

- Client: the component that needs dependency.
- Service (dependency provider): the component that is needed.
- Injector: the component that creates the service and injects it into the client.

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 performance.

An initial architecture for distributed dependency injection can be represented as follows:

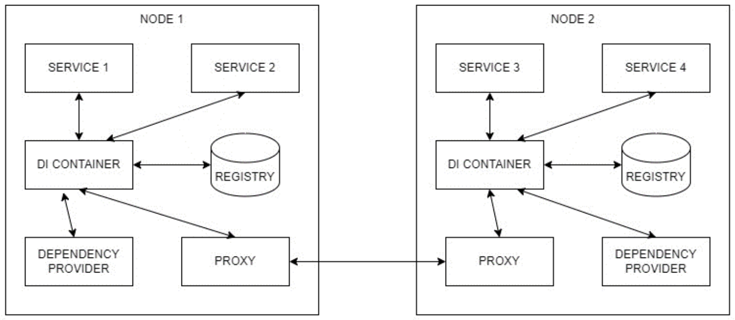

Figure 2: Distributed dependency injection architecture

When a service needs a dependency to be injected, it sends a request to the injector through its API. The injector will retrieve the dependency from the local registry if it is present, otherwise it will use a proxy to communicate with injectors on other nodes.

2.2 AWS Lambda and Azure Functions:

AWS Lambda is the FaaS solution offered by Amazon Web Services. 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.
- 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.

It is possible to tell Lambda the entry point to a 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.

Azure Functions is the respective solution offered by Microsoft. It presents the same characteristics as AWS Lambda, but they differ in the available scalability plans, in the limits of execution time and in the types of function's triggers. Furthermore, both platforms offer their own ecosystem of additional services that can be easily integrated in our functions, such as NoSQL databases (DynamoDB for AWS, CosmosDB for Azure) and monitoring systems (CloudWatch for AWS, ApplicationInsight for Azure). We created a free trial account on both platforms to implement our solution, keeping in mind that AWS Lambda's plan presents the limit of a maximum of ten instances per function running at the same time.

3 METHODOLOGY

3.1 Implementation:

Given that both AWS Lambda and Azure Function don't allow to run containers in the same runtime environment of the functions, it isn't possible to implement the architecture that we presented in the previous section. Instead, to maintain our solution as simple as possible, we decided to implement the injector as a class that provides methods to register and retrieve services from a NoSQL database (DynamoDB for AWS and CosmosDB for Azure). A service is represented as a tuple composed of *id*, *ServiceName* and *ServiceAddress*. We chose to implement the SDK in three different programming languages (for instance Javascript, Java and Go) in order to evaluate the performance with three different runtimes. We used the singleton pattern to implement the injector, so that a new connection to the databases is created only the first time a new instance of the function is created. We implemented also a simple hello-world Go function that we used as dependency provider to be injected in other functions. We initialized the databases with one entry representing this function.

3.2 Testing:

Both AWS and Azure offer tools to automatically collect a set of standard metrics during the functions execution, such as duration and number of invocations. To collect information about service retrievals duration and service binding duration, we had to proceed programmatically by logging messages into the monitoring systems.

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 could test how the scalability is managed either in a standard situation as in a critical one. 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.

4 RESULTS

4.1 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.

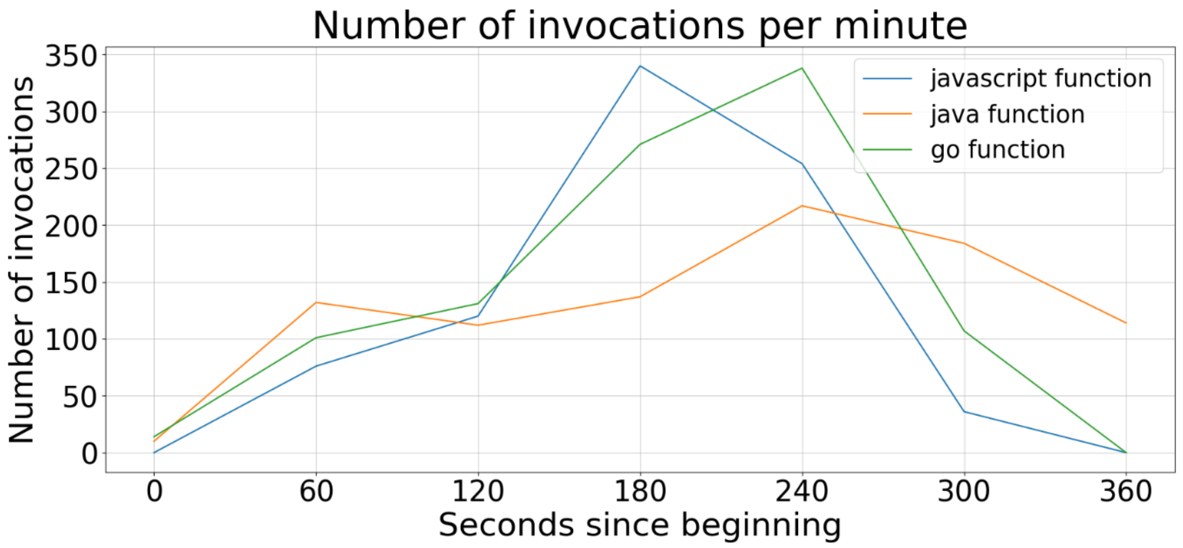

Figure 3: AWS – Number of invocations per minute

The invocation pattern is similar to all the functions: we have an initial phase in which the requests are constant and a peak phase. The differences between the functions in Figure 3 are caused by the different times in which we executed the stresser, as well as by delays when sending requests.

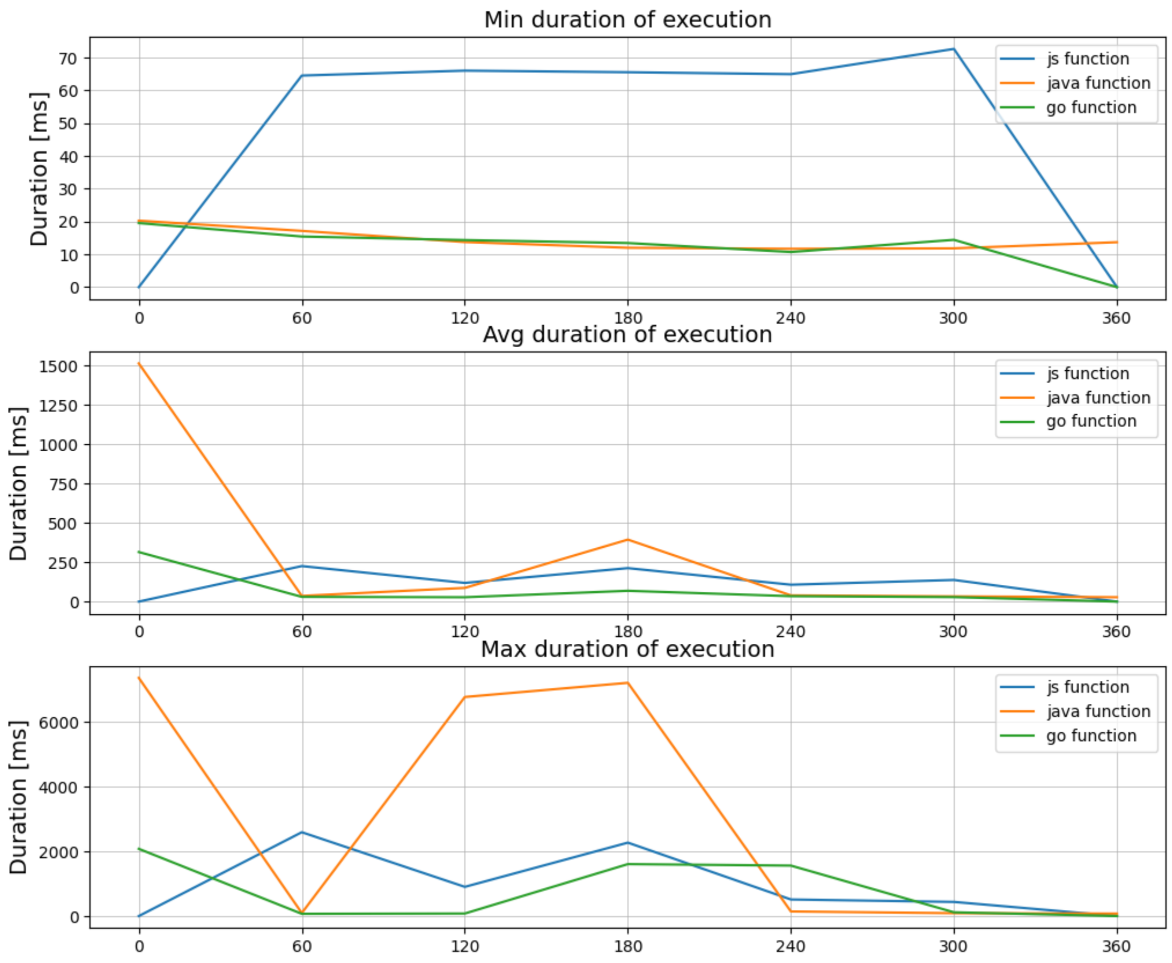

Figure 4: AWS – min/avg/max duration of execution per minute

As we can see in Figure 4, we obtained better performance with Go language in all the duration metrics. Cold starts in java are the slowest among the functions, while javascript function has the worst performance once the workload is settled.

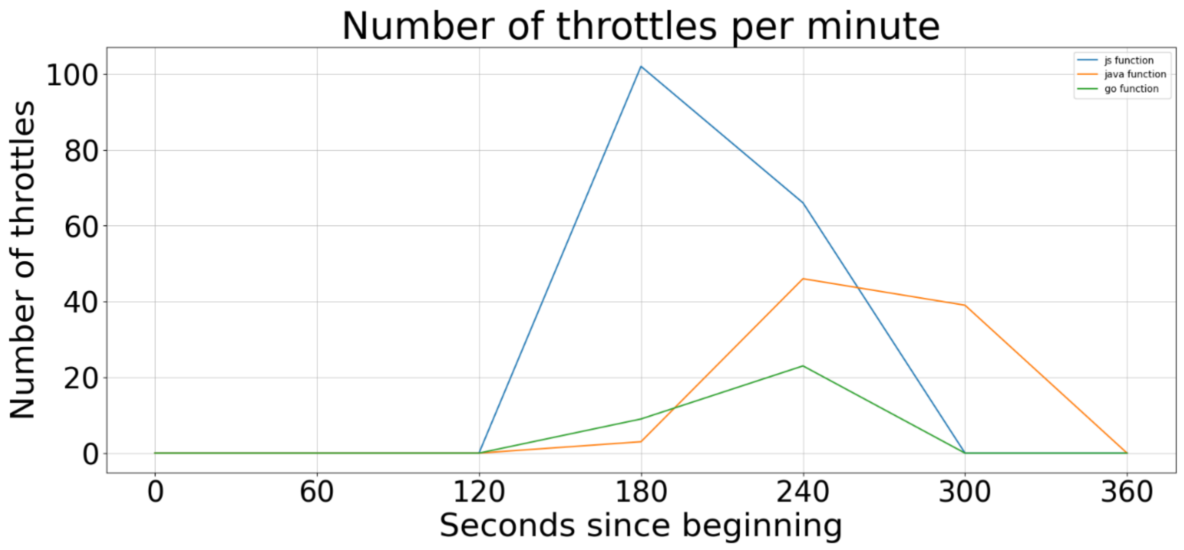

Figure 5: AWS – number of throttles per minute

In Figure 5 we can see that the Go function had the smallest number of throttles. This means that it could serve more requests with the same number of instances created compared to others. So, it is the fastest among them.

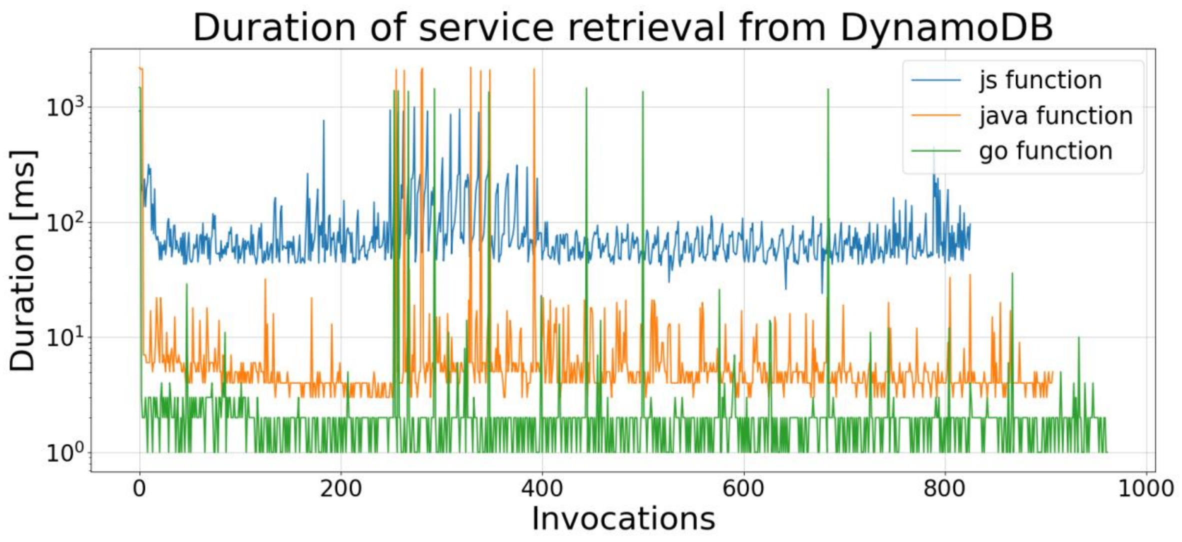

Figure 6: AWS – Duration of service retrieval from DynamoDB

In Figure 6, the peaks correspond to the creation of new connections to the database, so they represent the requests that caused the creation of a new instance of the function. Javascript SDK for DynamoDB is the fastest to create new connections to the database, but it is the slowest in reading. Go SDK clearly had the best results in service retrieval.

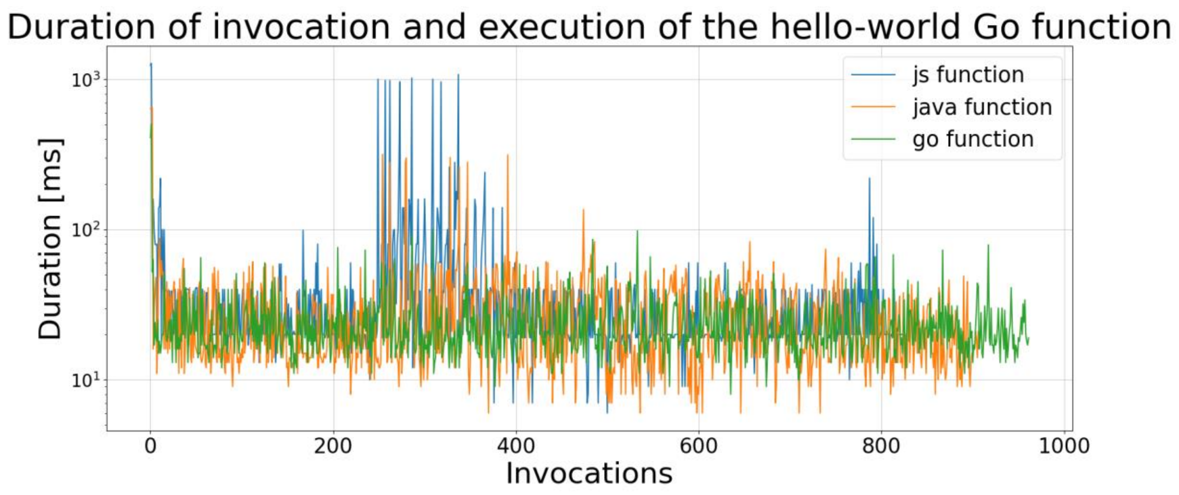

Figure 7: AWS – duration of invocation and execution of the hello-world Go function

As we can see in Figure 7, the Go function had the lowest peaks, but the Java function had faster bindings in most of the invocations

4.2 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 didn't track the throttles because there isn't a limit in the concurrent execution of functions.

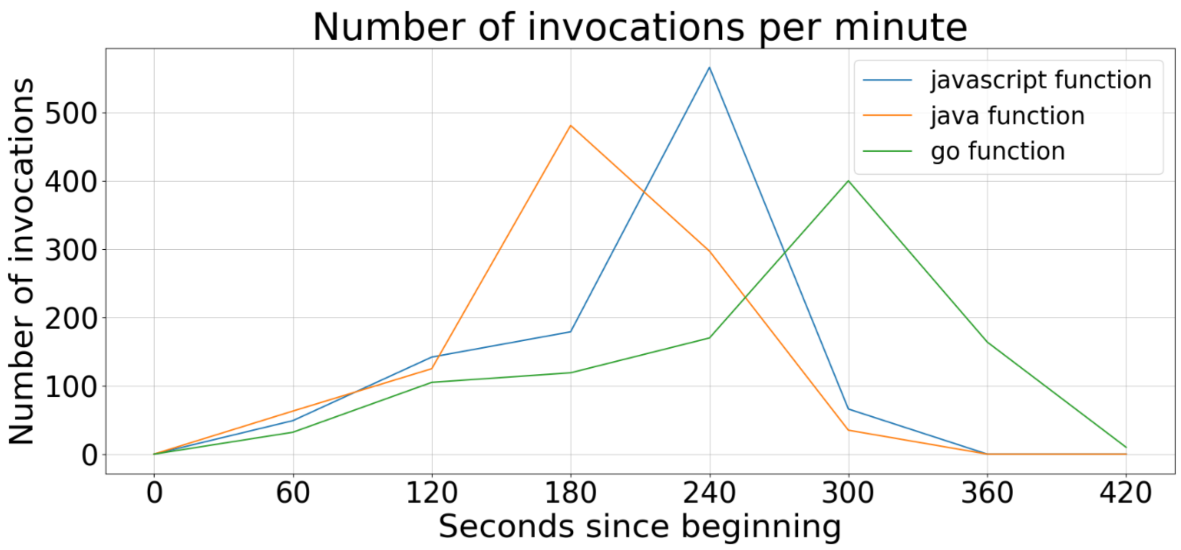

Figure 8: Azure – number of invocations per minute

All the considerations made for the previous case are also valid for Figure 8.

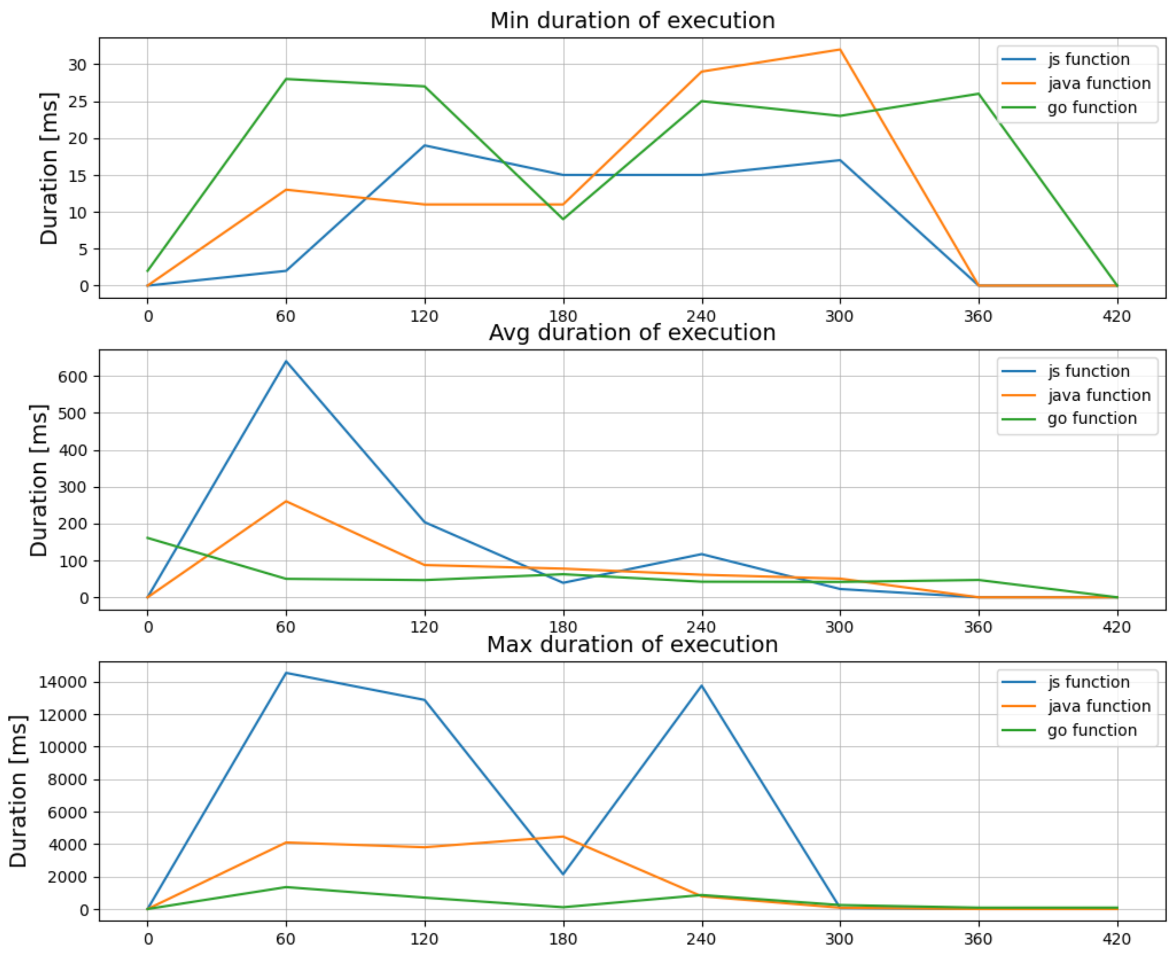

Figure 9: Azure - min/avg/max duration of execution per minute

As we can see in Figure 9, the Go function has the best performance except from the minimum duration of execution. Despite the fact that javascript slow starts are bigger compared to the ones on AWS Lambda, the average durations of the functions are smaller than the ones on AWS.

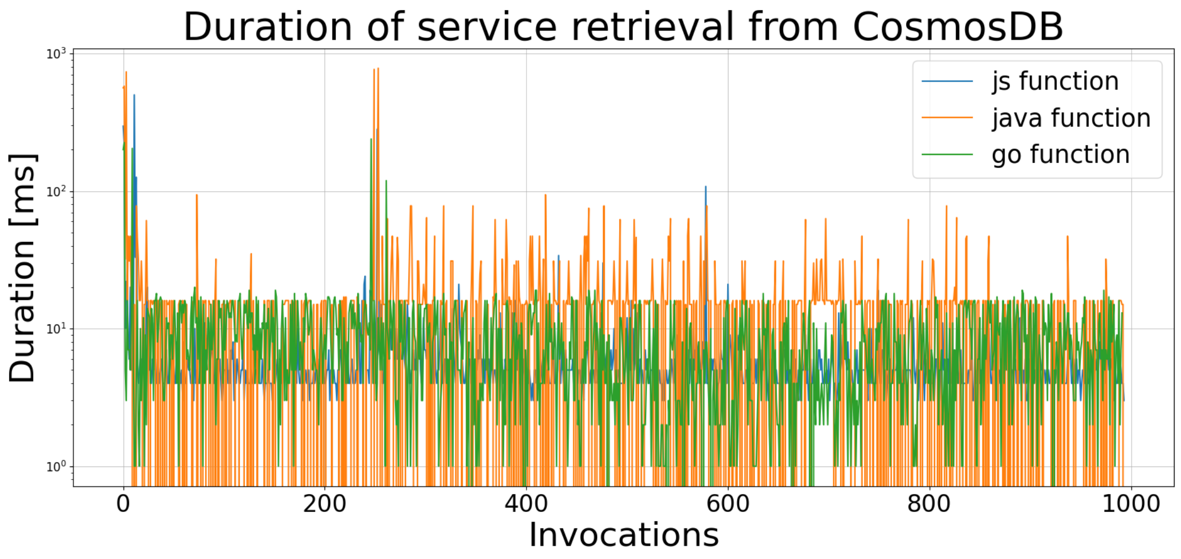

Figure 10: Azure - Duration of service retrieval from CosmosDB

As we can see in Figure 10, CosmosDB offers smaller times to create new connections compared to DynamoDB, while the read times are similar. Almost half of the retrievals made by the Java function took less than 1 ms. The Go SDK had the lowest peaks. In this graphic we can also notice that less than ten instances of every function were created to satisfy all the incoming requests, leading to smaller resource usage.

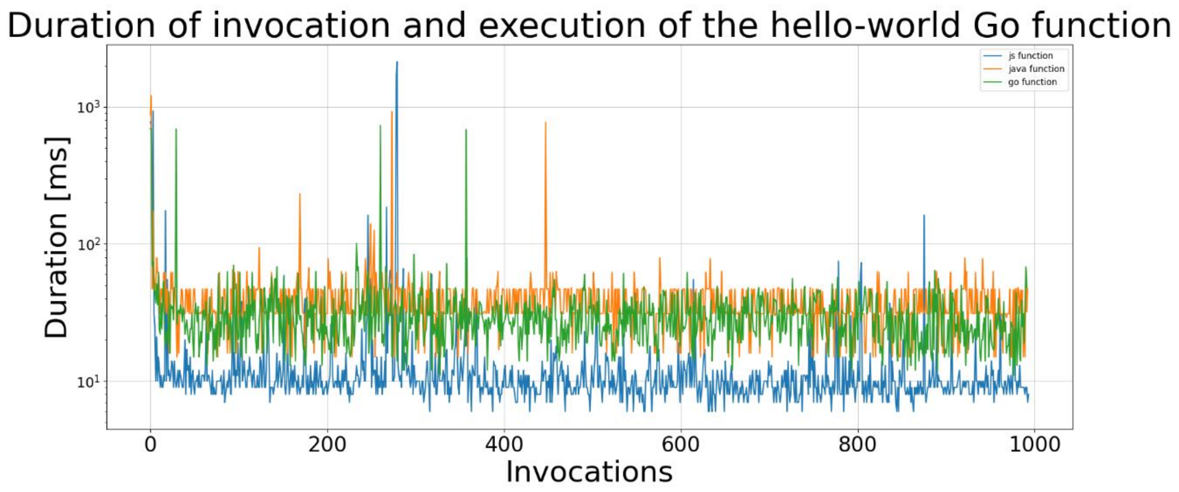

Figure 11: Azure – duration of invocation and execution of the hello-world Go function

In Figure 11 we can notice that the Javascript function performed better than others in binding duration. The Go function still had the lowest peaks.

5 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, CosmosDB's SDKs performed better both in creating new connections and reading the data compared to the DynamoDB's ones. Service binding duration is similar on both the platforms, with slightly bigger peaks on Azure caused by cold starts.

6 FURTHER RESEARCH

Given that public vendors' FaaS solutions don't offer the possibility to modify the underlying infrastructure in order to upgrade and optimize the distributed dependency injection architecture, we will switch to an on-premises infrastructure to improve our solution and to experiment services that aren't part of AWS and Azure ecosystems.

7 REFERENCES

[1] Red Hat, *What is serverless?*, 2022, https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless

[2] M. Roberts, *Serverless Architectures*, 2018, https://martinfowler.com/articles/serverless.html

[3] M. Roberts, *Learning Lambda*, 2017, https://blog.symphonia.io/posts/2017-11-14_learning-lambda-part-8

[4] AWS, *Best practices for working with AWS Lambda functions*, https://docs.aws.amazon.com/lambda/latest/dg/best-practices.html

[5] M. Seemann, *Dependency injection in .NET*, 2011

[6] N. Hodges, *Dependency Injection: Everything You Need to Know*, 2024, https://builtin.com/articles/dependency-injection

[7] AWS Lambda Documentation, https://docs.aws.amazon.com/lambda/

[8] Azure Functions documentation, https://learn.microsoft.com/en-us/azure/azure-functions/

[9] DynamoDB code examples for the SDK for JavaScript (v3), https://github.com/awsdocs/aws-doc-sdk-examples/blob/main/javascriptv3/example_code/dynamodb/README.md

[10] Working with Items in DynamoDB, https://docs.aws.amazon.com/sdk-for-java/v1/developer-guide/examples-dynamodb-items.html

[11] T. Hans, *Azure Functions with Go*, 2021, https://www.thorsten-hans.com/azure-functions-with-go/

[12] Azure Cosmos DB client library for JavaScript, 2024, https://learn.microsoft.com/en-us/javascript/api/overview/azure/cosmos-readme?view=azure-node-latest

[13] Azure Cosmos DB Client Library for Java, 2024, https://learn.microsoft.com/en-us/java/api/overview/azure/cosmos-readme?view=azure-java-stable

[14] Azure Cosmos DB Go examples, 2024, https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/samples-go