

INTRODUCTION

In recent years, Function as a Service (FaaS) has emerged as a pivotal paradigm in cloud computing, enabling organizations to develop and deploy applications more efficiently. This growth is fueled by the increasing need for scalable, cost-effective, and event-driven architectures that simplify application development while reducing infrastructure management overhead. As businesses continue to shift towards cloud-native solutions, FaaS is gaining prominence as an essential component of modern software engineering. Furthermore, FaaS systems are perfectly fitted for important scenarios such as Internet of Things (IoT) and Big Data, given their event-driven nature and scalability capacity. This makes them a first choice in many real-world use cases.

Of course, all these benefits come with some drawbacks, such as cold starts and difficulty in testing the functions. This last point is due mainly to the management of the dependencies inside the functions' code. In fact, functions usually need external services to perform their business logic and, so, to be useful. Minimizing the complexity of the dependencies management and writing loosely coupled code is crucial for FaaS as for traditional software engineering. For this reason, the concept of dependency injection from object-oriented programming can come in handy even in this case.

The objective of this work is to propose different architectures and implementations of dependency injection principle in FaaS scenarios, evaluating the performance and discussing the benefits and drawbacks of each case. In particular, the initial experiments will be conducted both on the main public cloud providers' FaaS platforms (AWS Lambda and Azure Functions) and on an on-premises infrastructure. In this case, the injector will be implemented as a library that can be used directly in the functions' code. The discussion about the results will show how public clouds don't allow the users to modify their infrastructure, so it is impossible to implement optimizations at infrastructure level. Subsequent experiments will be conducted only on the custom infrastructure, and they will focus on infrastructure optimizations, mainly switching the injection logic from the functions to the invokers, in order to make it transparent to the developers. Both the cases of dynamic and static injection will be taken into account, and performance differences between them will be discussed, focusing on benefits and drawbacks of each choice.

So, the work structure is organized in the following way:

- The initial chapter is dedicated to the description of the concepts at the basis of this work. In particular, it is described what serverless computing and dependency injection are, and how dependency injection can work in a FaaS scenario.
- The second chapter focuses on the description of the platforms used (AWS Lambda, Azure Functions and custom infrastructure) and their functioning, as well as the technologies involved.
- Afterwards, each chapter is dedicated to a particular implementation of the injection mechanism and to its evaluation. Three main architectures have been identified: the first one consists of an injector library that performs service retrieval from the registry and dependency resolution inside the functions' code. In the second one the injector is shifted from the functions to their invokers, with caching and connection pooling optimizations. The third architecture takes into account the possibility of directly invoking the dependency providers when they are local to the invoker, without passing through the trigger during the invocation.

In conclusion, we will discuss the proposed solutions and their implications in terms of trade-off between performance and infrastructure maintenance.

CHAPTER 1

BACKGROUND

1.1 Cloud computing:

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.

1.1.1 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 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.

1.1.2 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-premises 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.

1.1.3 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).

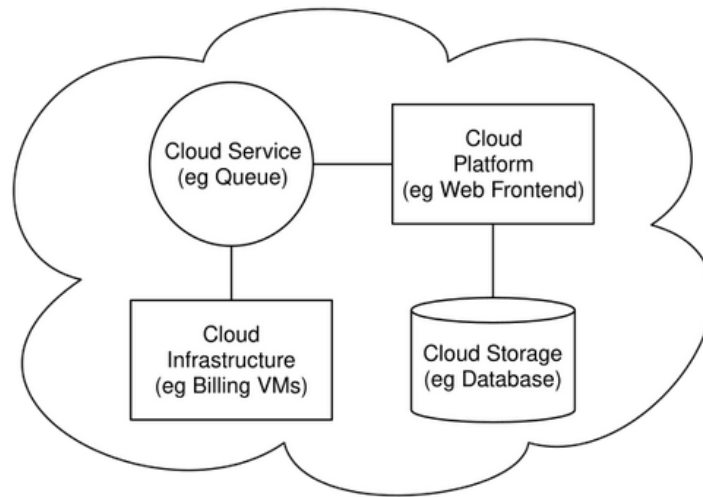


Figure 1.1: The four main components of cloud computing systems.

1.1.4 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 over 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-premises 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 of 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.

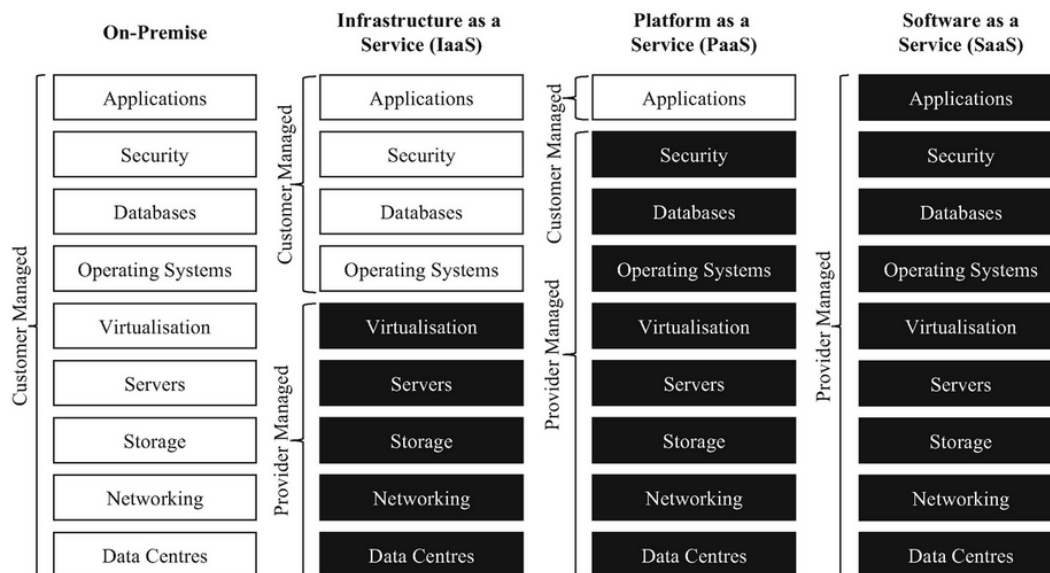


Figure 1.2: Main types of service models

1.1.5 Deployment 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.

1.2 Serverless computing:

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.

1.2.1 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 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.

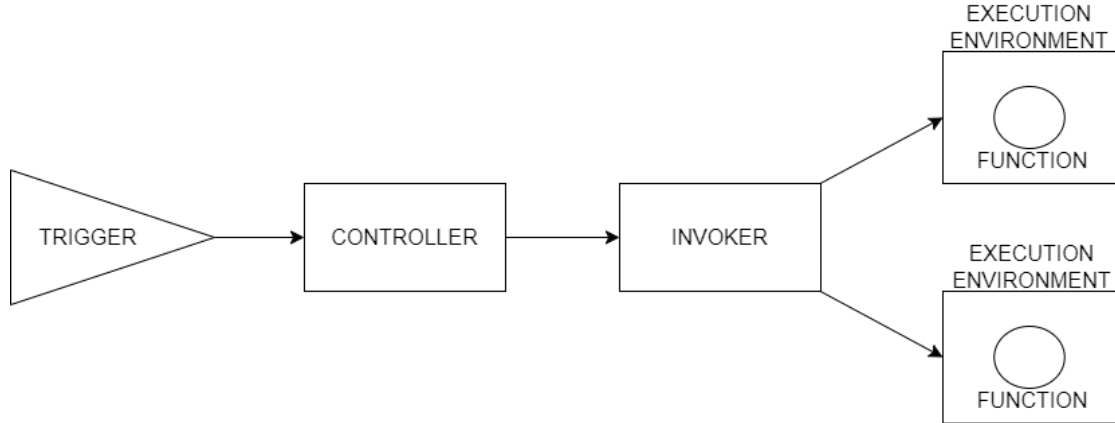


Figure 1.3: FaaS general architecture

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.

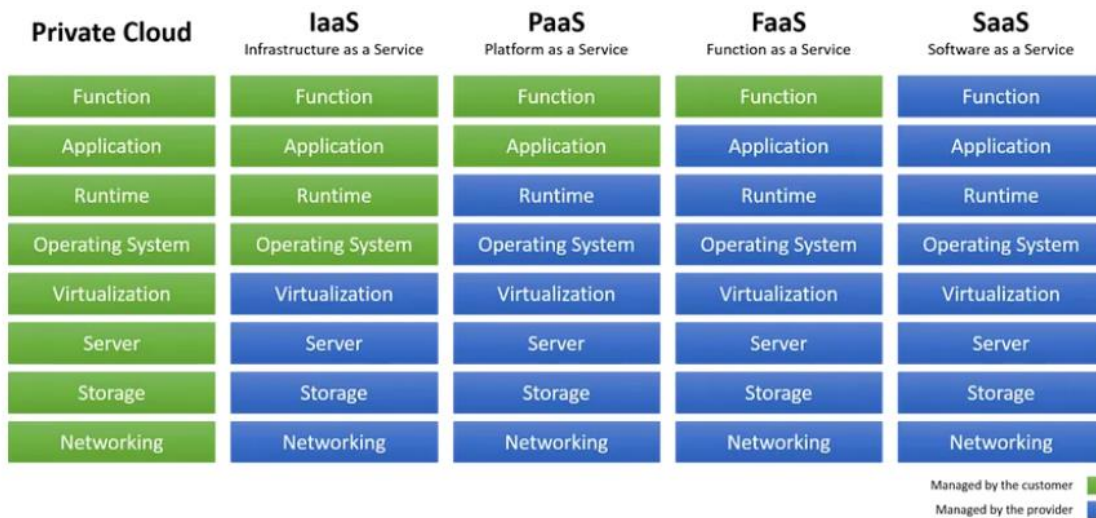


Figure 1.4: FaaS service model

1.2.2 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.

1.2.3 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 the 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.

1.2.4 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.

1.3 Dependency injection:

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:

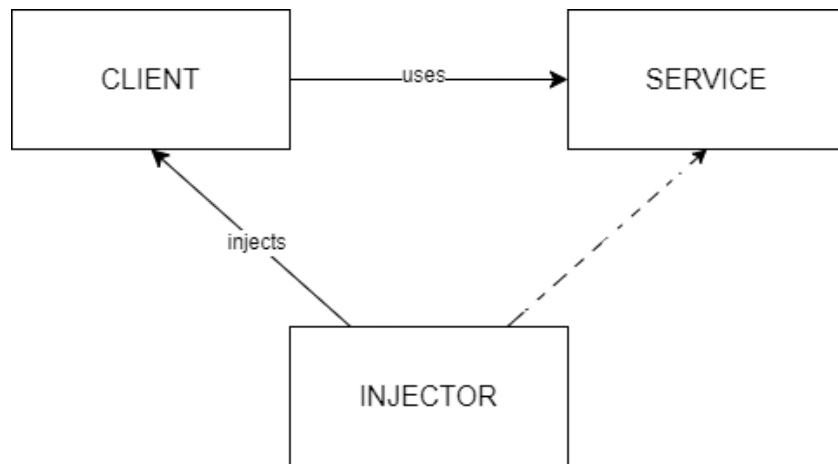


Figure 1.5: the main components involved in dependency injection

- Client: the class that needs dependency. Usually, annotations are used to indicate which dependencies are needed and at 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 follows:

- 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 performance.

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

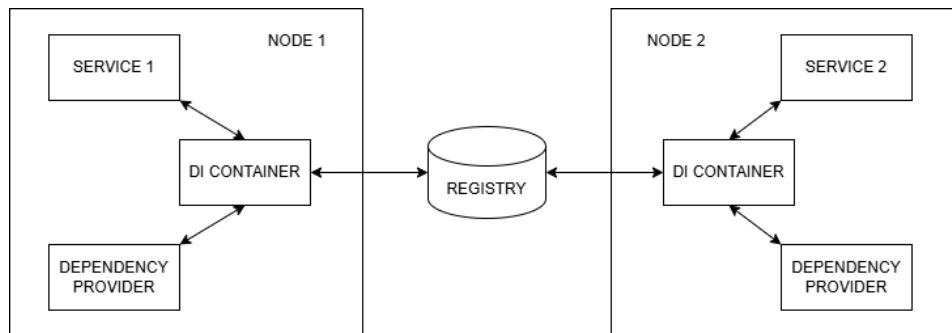


Figure 1.6: Distributed dependency injection architecture

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 retrieval 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.

1.3.1 Dependency injection in FaaS systems:

The previous architecture can be applied to every distributed system in general. To specify it for FaaS scenarios, we need to make some considerations. First of all, as we already know, functions are ephemeral, and their lifecycle and scaling management are totally demanded to the provider. So, we don't know on which node they will be executed and at which moment they will be activated. Nevertheless, the DI container should be active and running before the functions are executed, and it should be easily accessible by all the function instances. So, at least one DI container must be up and running on every node. With respect to the registry, it should be accessible to all the function instances too. Furthermore, every instance of the same function should have access to the same set of registered dependencies. So, it could be a good idea to have a single centralized instance of the registry.

According to these considerations, a possible architecture for dependency injection in FaaS systems can be represented as follows:

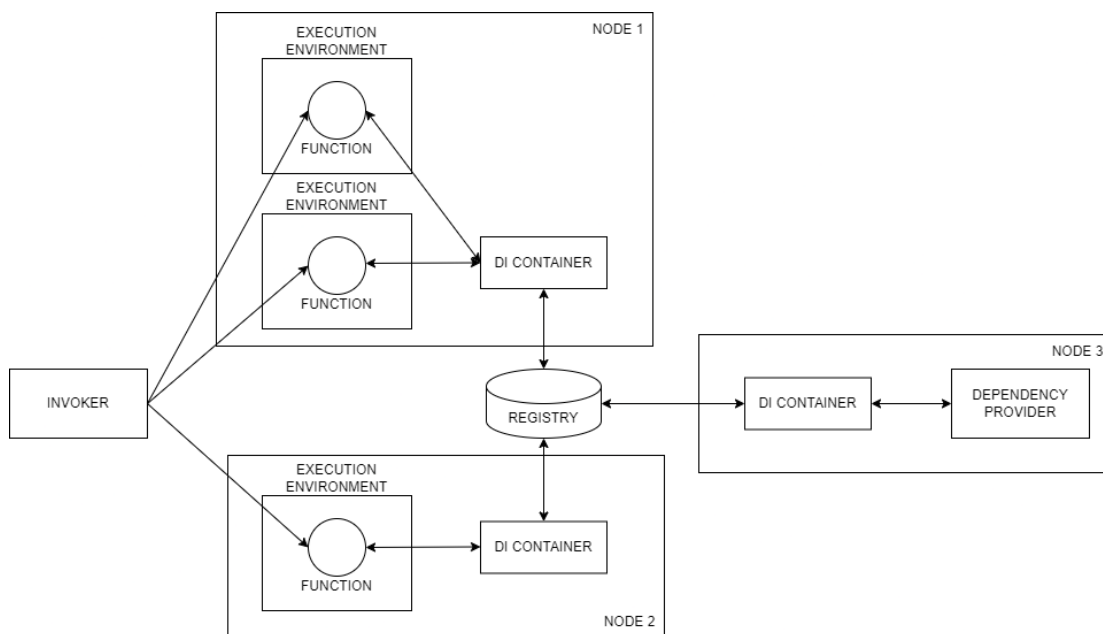


Figure 1.7: General architecture of dependency injection in FaaS systems

CHAPTER 2

PLATFORMS AND TECHNOLOGIES

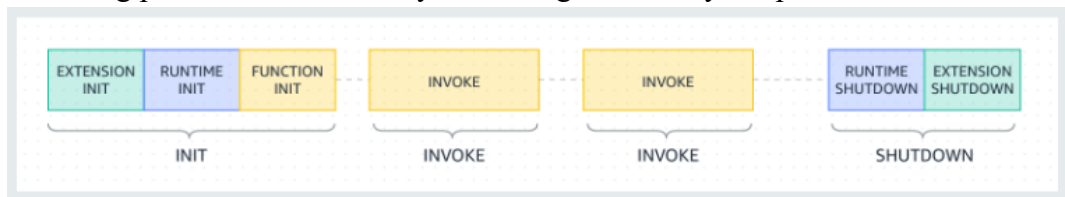
2.1 AWS Lambda and Azure Functions:

AWS Lambda

AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS) that allows users to run code without provisioning or managing servers. With AWS Lambda, it is possible to 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 their execution, 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.

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.

2.2 On-premises infrastructure:

The custom infrastructure implements the general architecture of a standard FaaS system presented in paragraph 1.2.1: it includes an HTTP trigger and many invokers, each associated with a certain function. The components are connected and can communicate through NATS, a message-oriented middleware.

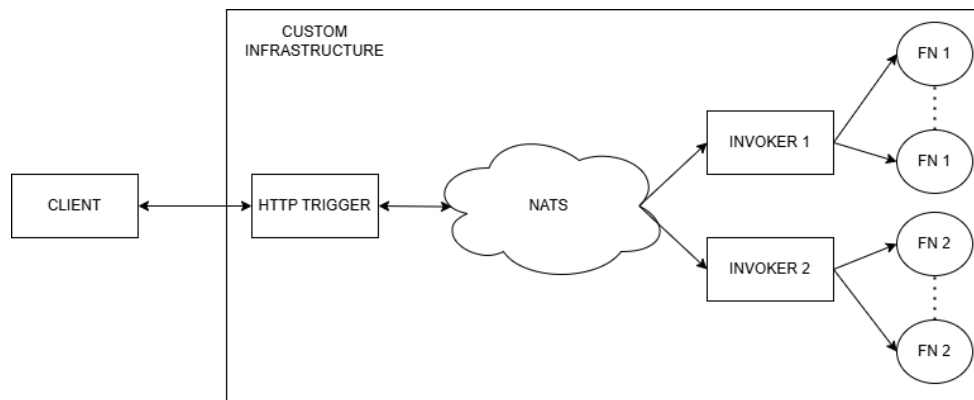


Figure 2.1: Custom infrastructure architecture

2.2.1 NATS:

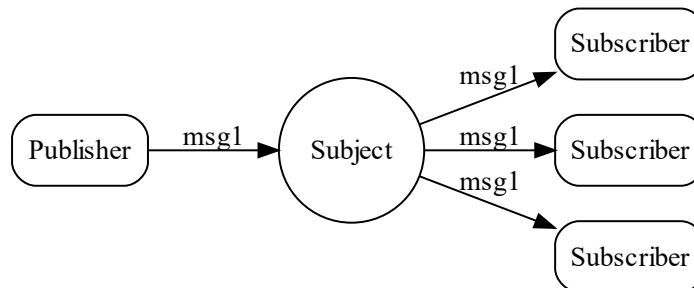
NATS is a high performance, open-source MOM designed to facilitate real-time communication between distributed systems, applications and devices. The NATS services are provided by one or more NATS server processes that are configured to interconnect with each other and provide a NATS service infrastructure. The NATS service infrastructure can scale from a single NATS server process running on an end device all the way to a public global super-cluster of many clusters spanning all major cloud providers and all regions of the world.

NATS is a subject-based messaging system that allows clients to publish and listen for messages on named communication channels called `Subjects`. At its simplest, a subject is just a string of characters that form a name the publisher and subscriber can use to find each other. More commonly subject hierarchies are used to scope messages into semantic namespaces. Through subject-based addressing, NATS provides location transparency across a cloud of routed NATS servers and a default many-to-many (M:N) communication pattern.

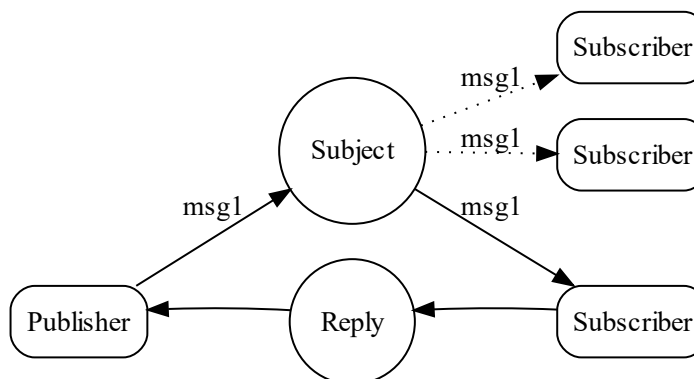
- Subject subscriptions are automatically propagated within the server cloud.
- Messages will be automatically routed to all interested subscribers, independent of location.
- Messages with no subscribers to their subject are automatically discarded.

The foundational functionality of NATS system is implemented in Core NATS. It operates in a publish-subscribe model using subject-based addressing. Starting from this base model, Core NATS offers three ways to exchange messages:

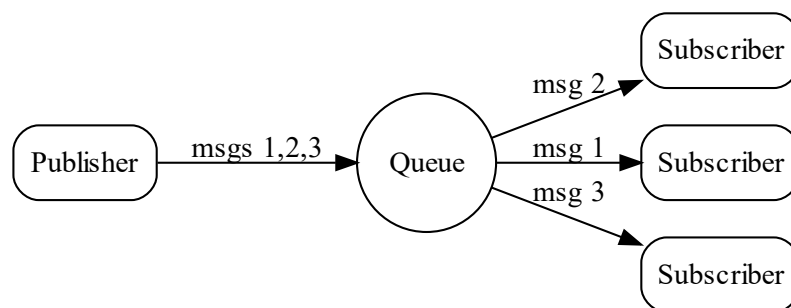
- Traditional publish-subscribe message distribution model: A publisher sends a message on a subject and any active subscriber listening on that subject receives the message.



- Request-reply: A request is published on a given subject using a reply subject. Responders listen on that subject and send responses to the reply subject. Reply subjects are called "inbox". These are unique subjects that are dynamically directed back to the requester, regardless of the location of either party.



- Queue groups: NATS provides an additional feature named "queue", which allows subscribers to register themselves as part of a queue. Subscribers that are part of a queue form the "queue group". In this way, when a message is published on a subject, only a randomly chosen subscriber of that subject receives the message.



Core NATS offers only a best-effort quality of service, in particular an at-most-once semantics for the delivery of messages. Anyway, it is possible to enable the built-in persistence engine, called JetStream, to reach higher levels of service quality.

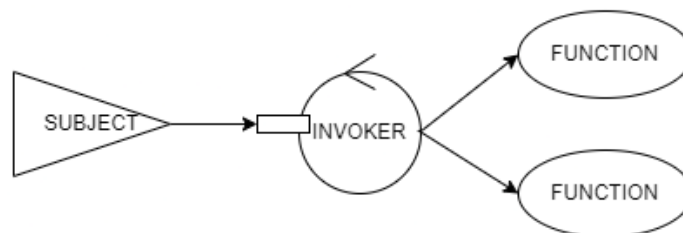
2.2.2 HTTP trigger:

It is the component that exposes the functions as HTTP URLs. In this way, a client can execute a function by sending an HTTP request to its specific URL. The task of the trigger is to associate the URL hit by the request to the right invoker. To do this, it is configured with a set of couples “address : topic” to bind the URLs to specific NATS subjects, in order to make trigger and invokers communicate.

In particular, the trigger is implemented in Rust with the `actix-web` crate. It exposes a REST API to trigger the execution of specific functions to be specified in the path and allows the clients to insert JSON formatted messages in the request’s body to pass as parameters to the functions. The functions can be invoked both synchronously and asynchronously, exploiting the NATS API to simply publish a message or wait for a response on a dedicated topic.

2.2.3 Invoker:

The invoker is the component designed for the association of the functions with specific subjects, which are the triggers of our infrastructure. So, an invoker is basically a NATS client that subscribes to a specific subject in a group queue and waits for messages (events) to be published on it. For every message, it runs the associated function in a separated process (`exec`), passing the message as input and publishing the output to a target output topic.



The invoker is implemented in Rust and uses the client SDK provided by NATS to connect to a NATS server and consume messages. It also uses the `tokio` library to manage concurrency in an asynchronous way. In fact, we don't want to block the invoker process during the execution of each instance of the function, but to let it continuously read new messages from the subject. So, the invoker basically acts as a control loop that delegates the messages to new tasks which will execute the function code synchronously in other processes. The code structure can be summarized as follows (pseudo-code):

src/invoker.rs

```
/* get environment variables (NATS server URL, subject
name, command name and arguments, etc...) */
let nc = nats::connect(&nats_server)?;
let sub = nc.queue_subscribe(&trigger_topic, &group)?;
for msg in sub.messages{
    tokio::spawn( async {
        /* execute function code and publish result on
        the output topic */
    });
}
```

In this way, it is possible to replicate the behavior of traditional FaaS platforms. We decided not to introduce a scaler component because evaluating the scalability of the system is not the main objective of this work. So, we will manually run a static number of invokers before executing our functions. The maximum number of instances of the same function running at the same time will be limited to the maximum number of executable parallel `tokio` tasks. The functions can be run directly from source code or by starting Docker containers containing them. In the first case, there are no cold starts, given that the execution environment is the same as the one of the invokers, and it is already set up.

CHAPTER 3

FIRST ARCHITECTURE: FUNCTION LEVEL INJECTION

3.1 Introduction:

The simplest way to implement dependency injection in a FaaS system is to implement the injector as an SDK to use inside the functions' code. The injector should provide two main methods: one for the registration of a service in the registry and one for the retrieval of a service. The registry can be implemented in many ways, but, basically, it should provide persistent memorization of the registered dependencies. Every record of the registry represents a service as a tuple of couples “key: value” that provide all the information needed to identify and invoke those services. So, every record should contain at least the service identifier and its address.

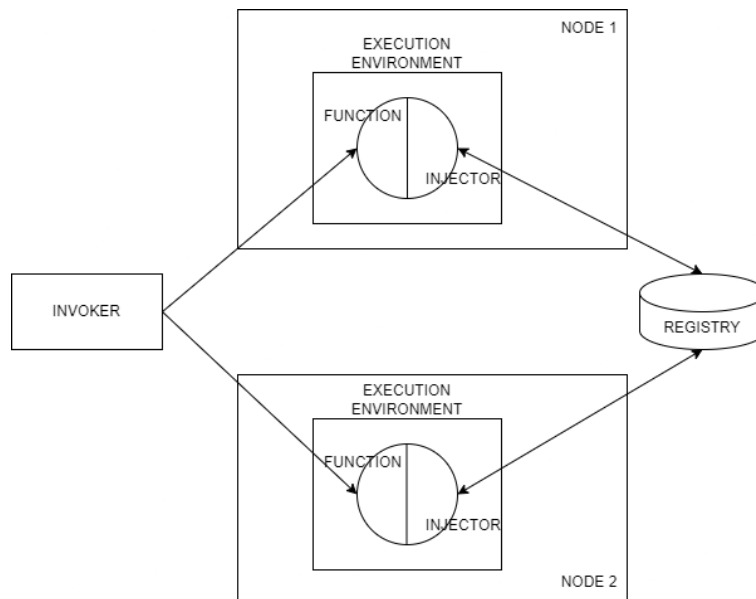


Figure 3.1: Function level dependency injection architecture

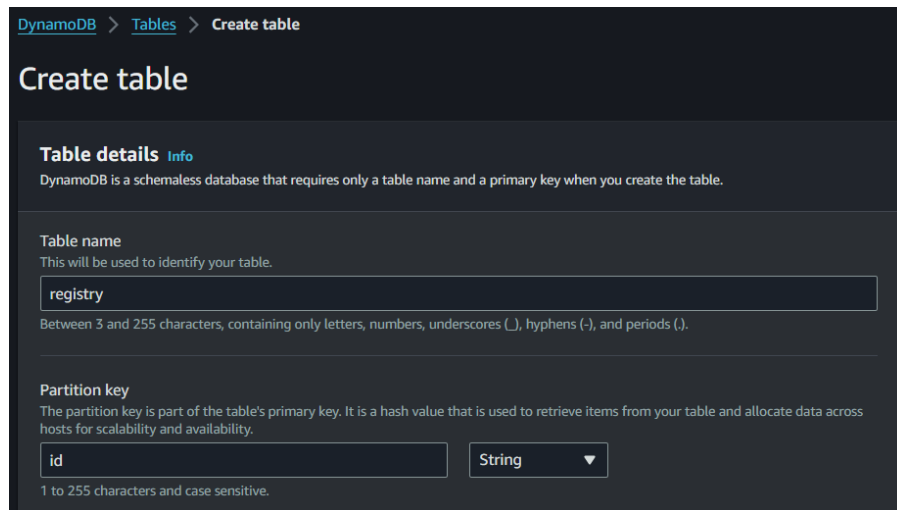
In the following paragraphs we will show how to implement this architecture on the presented platforms. The injector SDK is implemented in three different programming languages in order to evaluate the performance with different execution environments.

3.2 First architecture on AWS Lambda and Azure Functions:

Both AWS and Azure provide a rich ecosystem of services well integrated with their FaaS offers. In particular, we are interested in the NoSQL databases (DynamoDB and CosmosDB) to use as registries and in the logging systems (Cloud Watch and Application Insights) to collect data for the performance evaluation.

AWS Lambda

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:



The screenshot shows the AWS Management Console 'Create table' page for DynamoDB. The breadcrumb navigation at the top reads 'DynamoDB > Tables > Create table'. The main heading is 'Create table'. Below this, there is a 'Table details' section with an 'Info' link. A note states: 'DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.' The 'Table name' field is labeled 'Table name' and has a description 'This will be used to identify your table.' The input field contains the text 'registry'. Below the field, a note specifies: 'Between 3 and 255 characters, containing only letters, numbers, underscores (_), hyphens (-), and periods (.)'.

The 'Partition key' section is labeled 'Partition key' and has a description: '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.' The input field contains the text 'id'. To the right of the field is a dropdown menu currently set to 'String'. Below the field, a note specifies: '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. 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 on 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 from the AWS portal or command line interface (CLI):

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.

```
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 permissions for 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.

```
{
  "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 specifying 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.

```
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
}
```

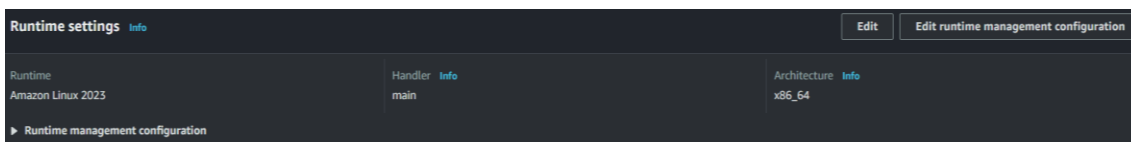
To deploy the function to Lambda, we need to pack it in a zip file. Each runtime has its own rules on which should be the structure of the zip file in order to execute the code. In the case of Go functions, we need to compile the module containing the handler in a file called “bootstrap”:

```
$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.



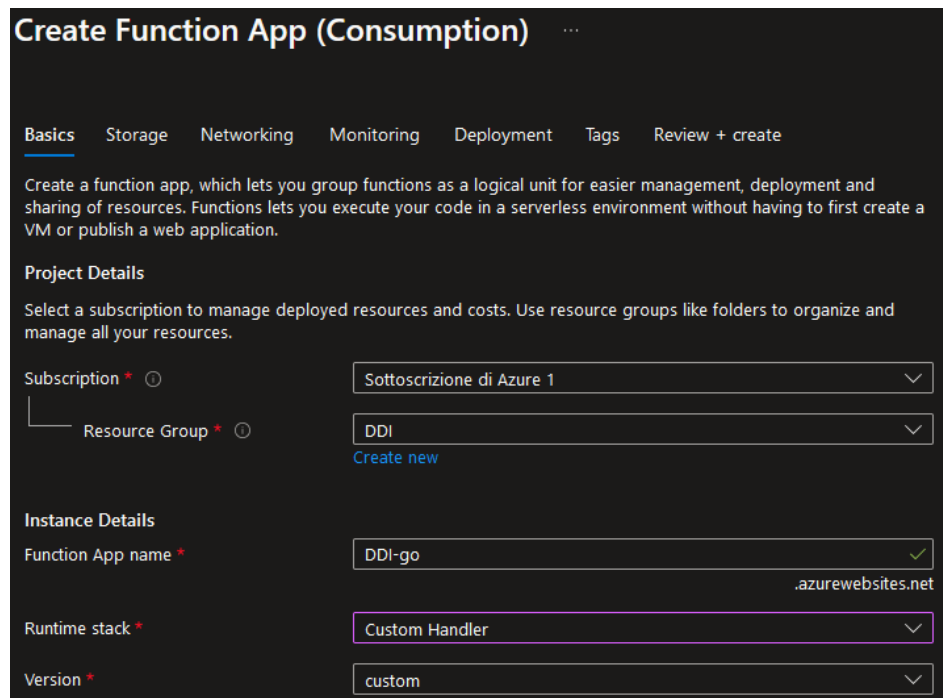
Azure Functions:

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.

Once the table is created, we can create a container inside it with the id parameter as partition key, 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) ...

Basics Storage Networking Monitoring Deployment Tags Review + create

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 ▼

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.

```
{
  "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.

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.

```
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.

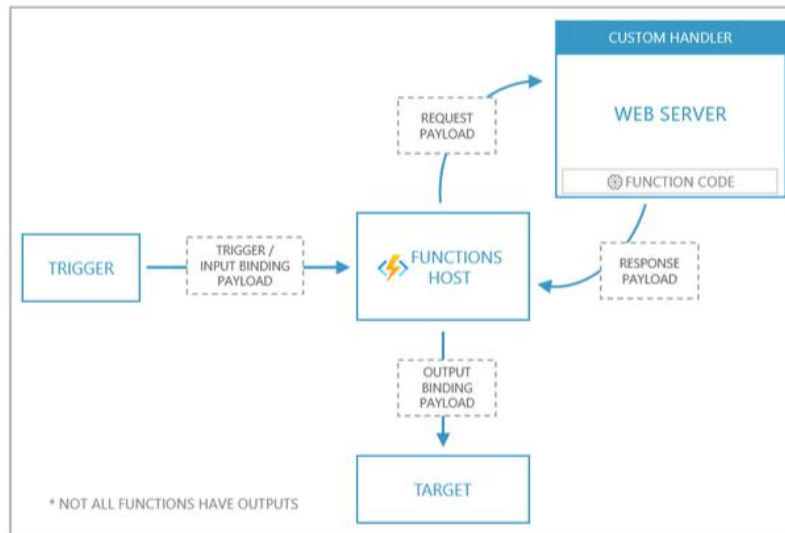


Figure 3.2: Azure Functions custom handler workflow

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.

```

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())
}

```

We can easily deploy the project to the Azure app using the Visual Studio Code extension.

3.3 First architecture on custom infrastructure:

Before implementing the first architecture on the custom infrastructure, we need to make some considerations. First of all, the infrastructure doesn't comprehend a scaling component. This means that it is possible to run a static number of invokers, and this will remain the same during all the execution. As we already said, the concurrency is obtained at the level of the single invoker thanks to the `tokio` library. So, we can't evaluate the scalability of the infrastructure, but it is not a problem since it isn't the objective of this thesis. The same thing can be said for the cold starts, given that they are not present unless we decide to invoke functions inside Docker containers. We introduced a configurable limitation in the invoker's code to reduce the maximum number of parallel runnable `tokio`'s tasks, in order to simulate a more realistic situation.

Furthermore, the infrastructure doesn't offer integrated services to achieve persistent storage. So, it is necessary to introduce an external database to realize the registry for the injector. The choice fell on MongoDB, a popular NoSQL database designed to handle large volumes of unstructured, semi-structured, or structured data efficiently. The reason is that it offers SDK for many programming languages, great performance in READS/WRITEs and the possibility to have flexible schemas for stored data. Moreover, we don't have strict consistency constraints, given that the WRITEs are few and done in a preliminary phase.

We decided to deploy the infrastructure as follows:

- On the first node, we run the NATS server, the MongoDB instance and the HTTP trigger.
- On the second node, we run the invoker related to the dependency providers.
- On the third node, we run the invoker related to the example functions.

In first instance, we use a simple hello-world function as dependency provider, like we did on AWS and Azure, in order to compare the results. Then, we will introduce more realistic dependency providers. Furthermore, we implemented again the injector in three different programming languages (Java, Javascript and Go) to evaluate any difference and compare the results with the ones obtained on the public clouds. Starting from these premises, we will show the steps needed to implement the function level dependency injection. We will concentrate on the Go function, given that the procedure is the same for all SDKs. The proposed solution's architecture will be:

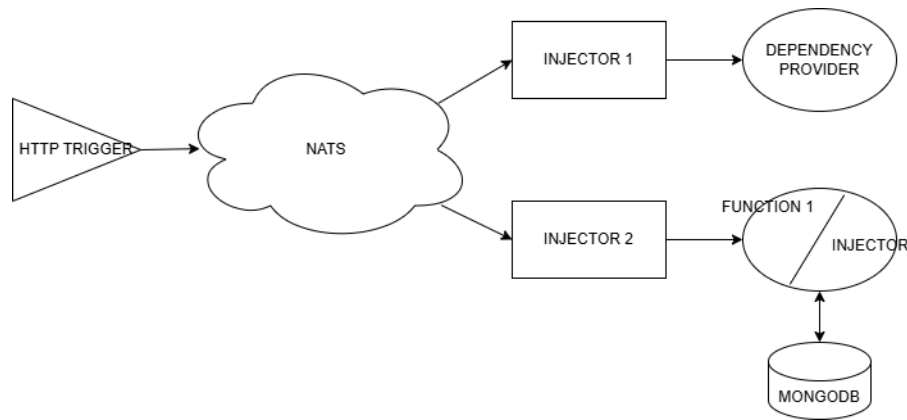


Figure 3.3: First architecture on custom infrastructure

Registry

First of all, it is necessary to install MongoDB on the first node and insert the entries related to the dependency providers. The entries are composed of id, service name and address. The address represents the URL to which perform a HTTP POST request in order to invoke the related function through the trigger. So, the URL is in the form `http://<trigger_IP>:<trigger_port>/function/<function_id>`. To install MongoDB version 8, run the following commands:

```

curl -fsSL https://www.mongodb.org/static/pgp/server-8.0.asc | sudo gpg -o /usr/share/keyrings/mongodb-server-8.0.gpg --dearmor

echo "deb [ arch=amd64,arm64 signed-by=/usr/share/keyrings/mongodb-server-8.0.gpg]https://repo.mongodb.org/apt/ubuntu noble/mongodb-org/8.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-8.0.list

sudo apt-get update

sudo apt-get install -y mongodb-org
  
```

Once it is installed, it is possible to interact with the `mongod` service through the dedicated shell by running the command `mongosh`. In this way, we can create a new

database containing our collection. To initialize the table, it is possible to use the command `db.collection.insertOne(...)`, passing a JSON formatted string containing the data.

By default, MongoDB launches with `bindIp` set to `127.0.0.1`, which binds to the localhost network interface. This means that the `mongod` can only accept connections from clients that are running on the same machine. To modify this behavior, it is possible to change the `bindIp` option in the MongoDB configuration file to `0.0.0.0`. In this way, `mongod` will accept connections from any node.

NATS

It is possible to install NATS server version 2.0 by running the following commands:

```
wget https://github.com/nats-io/nats-server/releases/
download/v2.0.0/nats-server-v2.0.0-linux-amd64.tar.gz

tar -xvzf nats-server-v2.0.0-linux-amd64.tar.gz

sudo mv nats-server-v2.0.0-linux-amd64/nats-server/
usr/local/bin/
```

At this point, run `nats-server` to start the server on default port 4222.

Injector

The injector is implemented as a simple Go class that provides an API to register and retrieve services from the registry using the client provided by MongoDB.

```

func (i *Injector) RegisterService(id, name, address string) {
    service := bson.D{
        {Key: "id", Value: id},
        {Key: "ServiceName", Value: name},
        {Key: "ServiceAddress", Value: address},
    }
    _, err := i.collection.InsertOne(context.TODO(), service)
    if err != nil {
        log.Fatal(err)
    }
}

func (i *Injector) GetServiceById(id string) Service {
    start := time.Now()
    var service Service
    err := i.collection.FindOne(context.TODO(), bson.D{{Key: "id", Value: id}}).Decode(&service)
    if err != nil {
        log.Fatal(err)
    }
    end := time.Now()
    i.logger.Infof("Read from MongoDB table executed in %v ms", end.Sub(start).Milliseconds())
    return service
}

```

Example function

The example function creates a new instance of the injector, gets the dependency passing its identifier and invokes the function using the address retrieved from the registry.

```

func handler() {
    i := injector.NewInjector()
    helloService := i.GetServiceById("hello")

    address := helloService.ServiceAddress
    message := map[string]string{"message": "world"}
    result, err := invokeFunction(address, message)
    if err != nil {
        log.Fatal(err)
    }
    os.Stdout.WriteString("hello function result: " + result)
}

```

The related invoker runs this function through an `exec`. So, to pass the output to it, the function must write the result on the standard output. The `invokeFunction` method hides the HTTP request to the trigger.

HTTP Trigger

The trigger exposes an endpoint to convert the HTTP request to messages published on NATS topics. Furthermore, it expects to receive as body of the requests a JSON formatted object structured as follows:

```
{ "message": "<json_formatted_message>" }
```

The content of the message is published to the NATS topic. This mechanism enables function-to-function communication through JSON-formatted messages. The invokers will pass these messages as `exec` argument to the functions.

```
#[post("/function/{fun}")]
async fn sync_invoke(
    routes: web::Data<DashMap<String, String>>,
    req: web::Json<PublishRequest>,
    fun: web::Path<String>,
) -> impl Responder {
    let function_name: String = fun.into_inner();
    let message: Message = query(function_name.clone(), routes, req).await.unwrap();
    info!("handling request to function: {}", function_name);
    HttpResponse::Ok().body(message.data)
}
```

Hello function

The dependency provider is a simple hello-world function that reads an argument and prints it to standard output.

```
func main() {
    // Check if an argument is provided
    if len(os.Args) < 2 {
        fmt.Println("Usage: go run main.go <your_argument>")
        return
    }
    // Read the first argument (os.Args[1])
    arg := os.Args[1]
    hello(arg)
}

func hello(arg string) {
    os.Stdout.WriteString("hello " + arg)
}
```

Invokers

As we said, we need to run an instance of the invoker for every function. So, we run one invoker for the example-function and one for the service provider. There still is the possibility to start more instances of the same invoker to load balance the messages received on the trigger topic through NATS queue groups.

The invokers are the same for every function, but they can be configured through environment variables.

```
//Get ENV VAR
let command: String = env::var("COMMAND").unwrap_or("node ../function/ACL-js-function/hello".to_string());
let trigger_topic: String = env::var("TRIGGER").unwrap_or("acl".to_string());
let output_topic: String = env::var("OUTPUT").unwrap_or("output".to_string());
let nats_server: String = env::var("NATSSERVER").unwrap_or("192.168.17.118:4222".to_string());
let group: String = env::var("GROUP").unwrap_or("default".to_string());
let max_instances: usize = env::var("MAX_INSTANCES").unwrap_or("10".to_string()).parse::<usize>().unwrap();
```

The `COMMAND` variable determines which command will be run inside the `exec`. For example, if the related function is written in Javascript, `COMMAND` will be:

```
"node <path_to_function>"
```

The `TRIGGER` variable determines the NATS subject to which the invoker will subscribe and wait for messages. `OUTPUT` contains the default NATS subject to which send the result of the function in case it was not specified in the trigger message. `NATSSERVER` contains the IP address of the NATS server instance. `GROUP` variable is used when more invokers subscribe to the same subject in order to load balance the incoming messages. `MAX_INSTANCES` can be used to define the maximum number of `tokio` tasks that can run at the same time.

3.4 Performance evaluation:

Once we set up the environment as described in the previous paragraphs both on public clouds and custom infrastructure, it is possible to trigger the execution of example functions by sending an HTTP request to their URLs. In the case of AWS Lambda and Azure Functions, the URLs are provided directly by the platforms. On the custom infrastructure, the URLs are structured in the following way:

```
http://<trigger_IP>:<trigger_port>/function/<function_id>
```

To test the performance of the distributed dependency injection architecture on the different platforms, we prepared 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 critical one, and which are the access times to the registries.

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.

```

private static void stressTestLambda() throws InterruptedException {
    ScheduledExecutorService scheduler = Executors.newScheduledThreadPool( corePoolSize: 1);
    // Preliminary phase with constant load
    runPreliminaryPhase(scheduler, INITIAL_REQUESTS_PER_SECOND, PRELIMINARY_PHASE_DURATION_SECONDS);
    // Incremental phase to peak
    for (int currentRequestsPerSecond = INITIAL_REQUESTS_PER_SECOND;
        currentRequestsPerSecond <= PEAK_REQUESTS_PER_SECOND;
        currentRequestsPerSecond += INCREMENT) {
        runLoadPhase(scheduler, currentRequestsPerSecond, STEP_DURATION_SECONDS);
    }
    // Decremental phase back to initial
    for (int currentRequestsPerSecond = PEAK_REQUESTS_PER_SECOND;
        currentRequestsPerSecond > INITIAL_REQUESTS_PER_SECOND;
        currentRequestsPerSecond -= DECREMENT) {
        runLoadPhase(scheduler, currentRequestsPerSecond, STEP_DURATION_SECONDS);
    }
    // final phase with constant load
    runPreliminaryPhase(scheduler, INITIAL_REQUESTS_PER_SECOND, durationSeconds: PRELIMINARY_PHASE_DURATION_SECONDS/2)
}

```

Given that in the custom infrastructure there are no cold starts, no horizontal scaling and the concurrency is managed with the `tokio` library, the capacity of serving an incremental workload depends on the capacity of spawning more parallel `tokio` tasks. These tasks are light weight, non-blocking units of execution, similar to threads, but managed by the `tokio` runtime. So, they are very efficient and can be created easily in a large number. To simulate a more realistic situation of a standard FaaS system, we decided to limit the maximum number of tasks that can be spawned. In this way, the effect of the increasing workload is more evident. Furthermore, we added a delay of 500 milliseconds to the dependency provider execution to slow down the creation of tasks.

The parameters taken into consideration to evaluate the incidence of the injection over the execution are basically three: the duration of the service retrieval from the registry, the duration of the dependency provider binding (combination of invocation and execution times) and the total execution time (composed by invocation, injection and service binding times).

3.4.1 Service retrieval time:

It represents the time needed to access the registry and perform a READ operation. Basically, it depends on the particular technology used to implement the registry and on the management of the connections to it.

AWS Lambda and Azure Functions

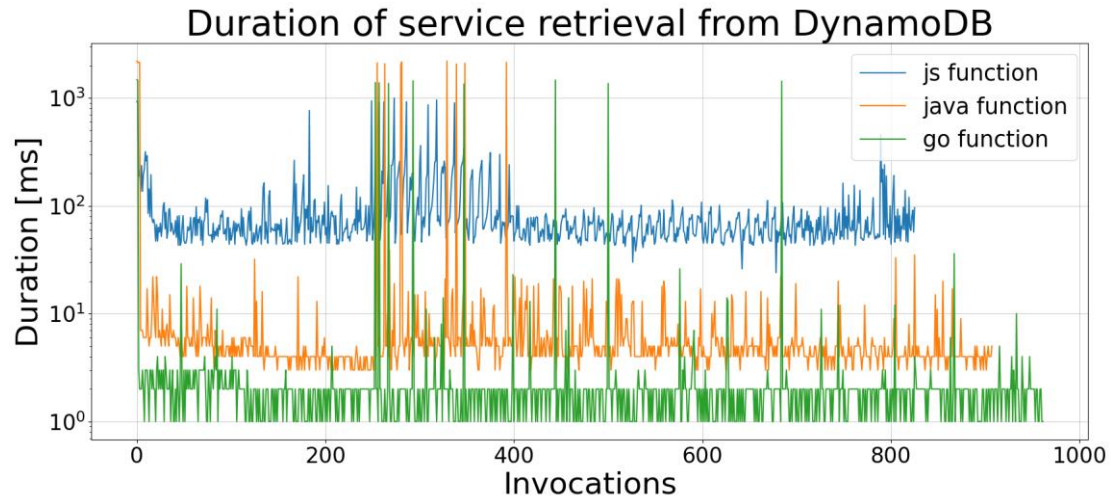


Figure 3.4: Duration of service retrieval from DynamoDB – AWS Lambda

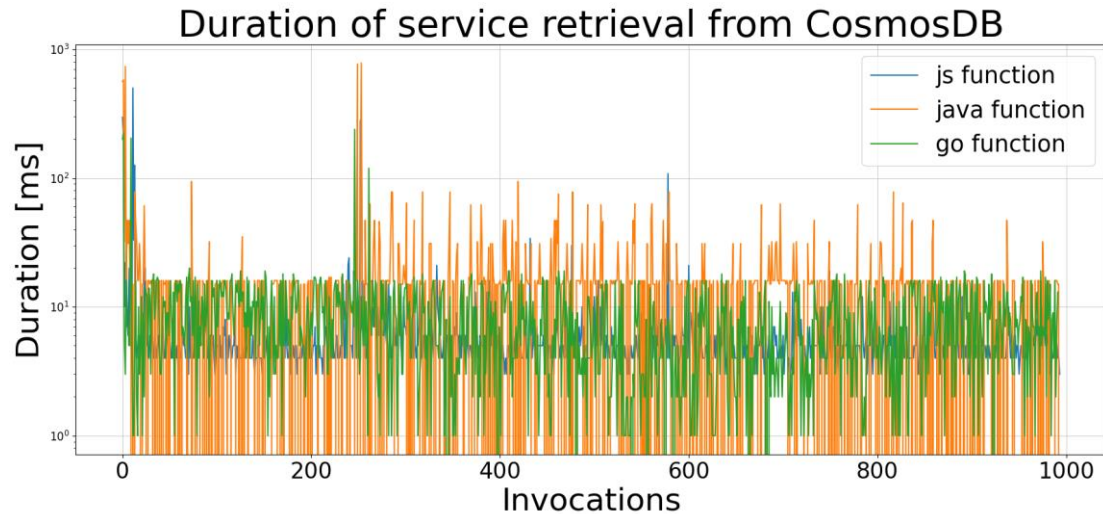


Figure 3.5: Duration of service retrieval from CosmosDB – Azure Functions

As we can see, the retrieval durations are almost constant for every SDK, except for some peaks. These peaks correspond to the creation of new connections to the databases and, so, to the creation of a new instance of the functions by the FaaS systems. With regard to DynamoDB, in figure 3.4 it is possible to notice that the peaks exceed the second in all the SDKs, but the Go function performed better in general, with average durations between 1 and 5 ms. In figure 3.5, the Java function registered the lowest durations. In fact, in almost all the invocations, they were under 1 ms.

Furthermore, it is possible to notice that the second figure shows less peaks. The reason is that our example function performed better on Azure Functions, so less instances of it were needed to serve all the incoming requests.

Custom infrastructure

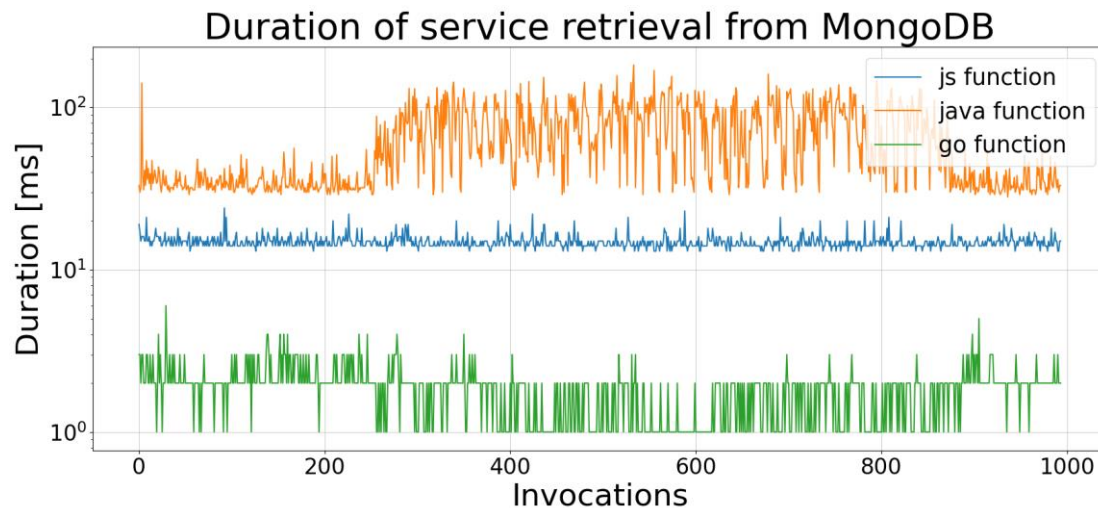


Figure 3.6: Duration of service retrieval from MongoDB – Custom infrastructure

In this case, a new connection to MongoDB is created every time a function is invoked, given that their lifetimes are limited to only one execution. Nevertheless, no peaks were registered and only the Java SDK performance was affected by the increasing workload. Even in this case, the Go SDK performed better, with almost all durations below 5 ms (comparable to DynamoDB).

3.4.2 Service binding time:

The service binding time represents the time needed to invoke the dependency provider by sending an HTTP request to its URL plus the time needed for its execution and the reception of its result. Under the hood, once the HTTP triggers have received the request, they dispatch it through the communication system of their platform to reach the associated invoker, that will finally invoke the function and send back its result. So, the binding time depends on many factors besides dependency provider's execution time.

AWS Lambda and Azure Functions

Duration of invocation and execution of the hello-world Go function

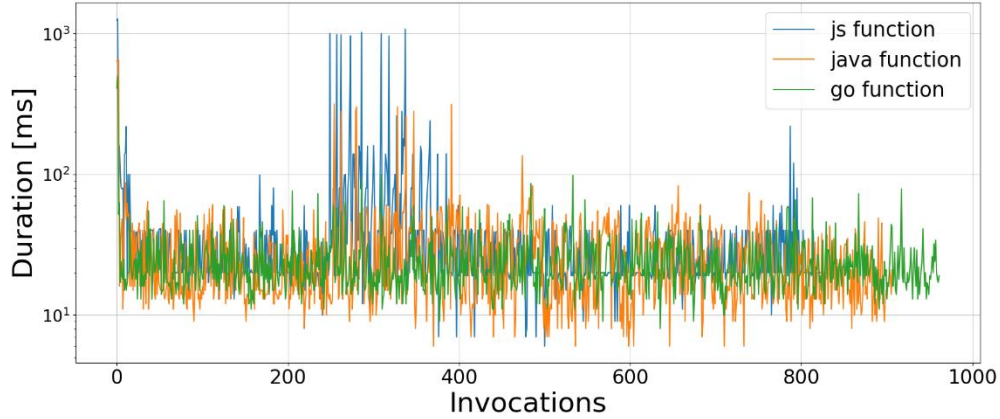


Figure 3.7: Service binding duration – AWS Lambda

Duration of invocation and execution of the hello-world Go function

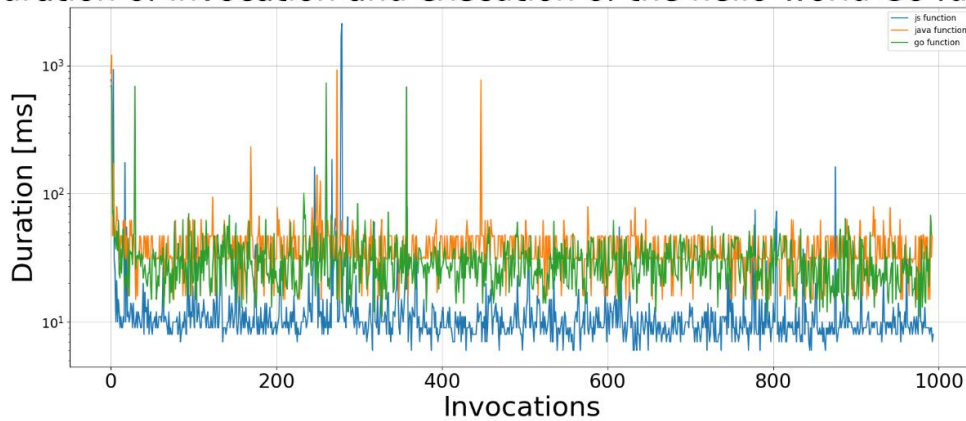


Figure 3.8: Service binding duration – Azure Functions

Both on AWS and Azure, the binding time is affected only by the cold starts of the dependency provider. On Azure, the Go SDK performed slightly better than others, while on AWS the durations present less differences.

Custom infrastructure:

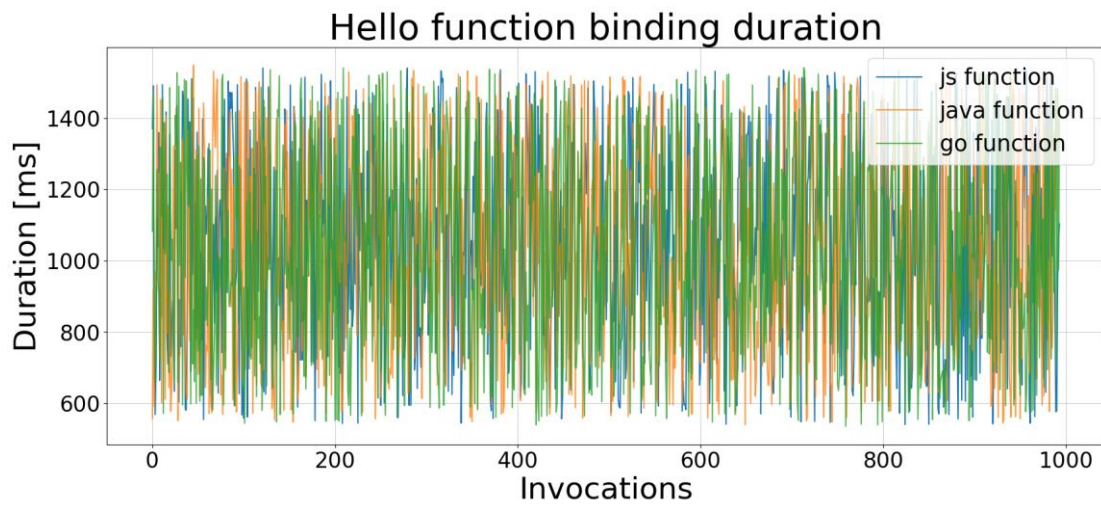


Figure 3.9: Service binding duration – Custom infrastructure

The binding duration is unaffected by the increasing workload, but its variability is high. Given that the execution time of the dependency provider is almost static around 500 ms, the variability is caused by the trigger capability and NATS communications.

3.4.3 Total execution time:

It represents the time between the reception of the request to the example function and the response. So, it includes all the previous parameters plus the time needed to invoke and execute the example function.

AWS Lambda and Azure Functions

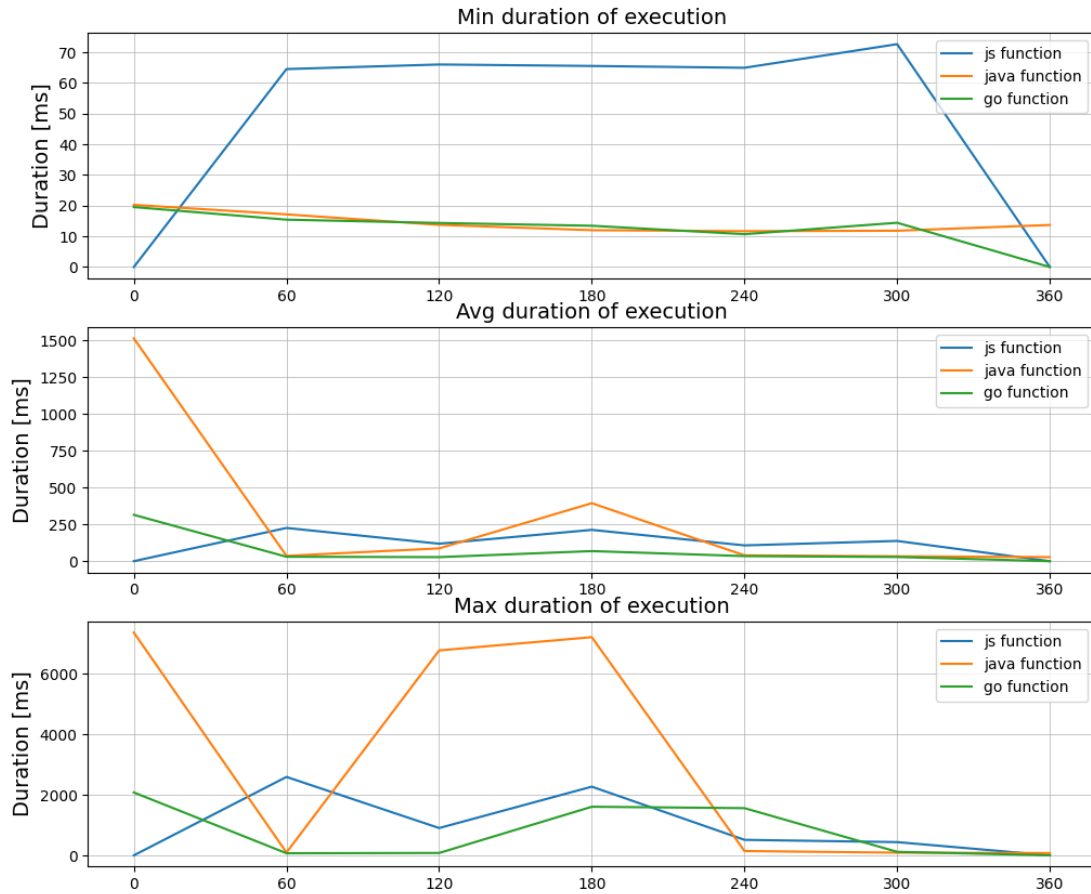


Figure 3.10: min/avg/max duration of execution per minute – AWS Lambda

As we can see in Figure 3.10, we obtained better performance with Go language in all duration metrics. Cold starts in Java are the slowest among the functions, while Javascript function has the worst performance once the workload is settled.

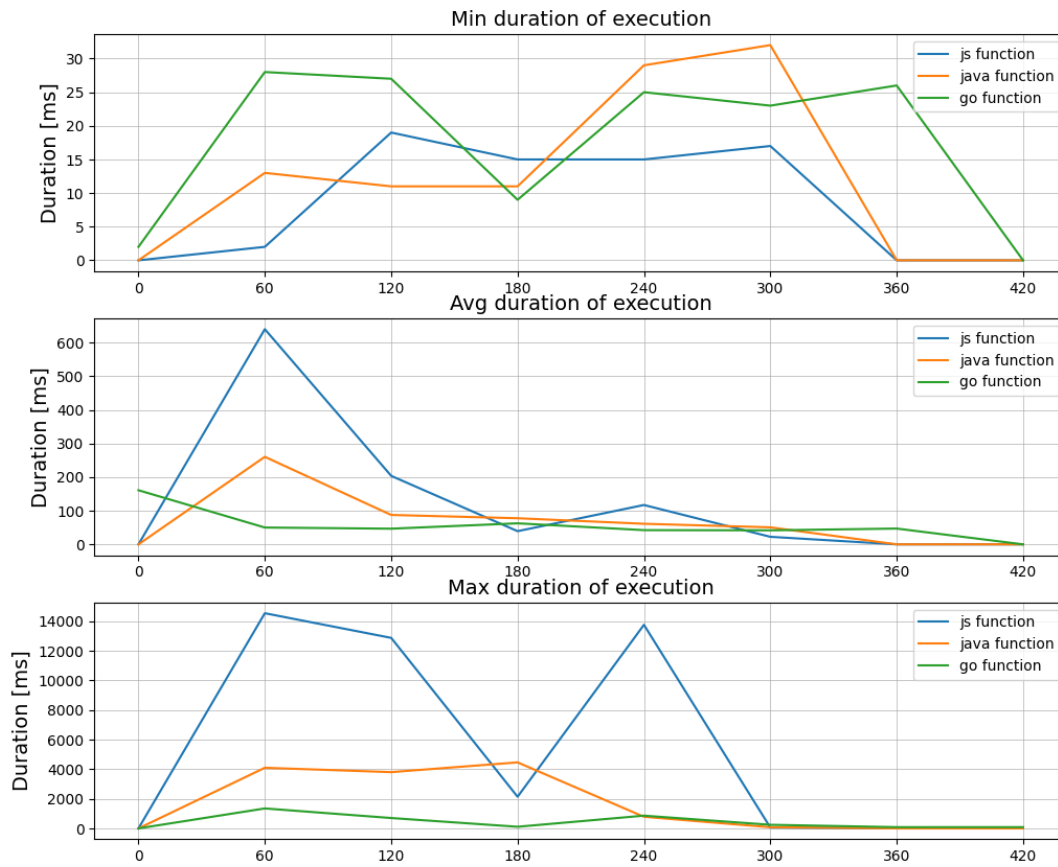


Figura 3.11: min/avg/max duration of execution per minute – Azure Functions

As we can see in Figure 3.11, the Go function has the best performance except for 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.

Custom infrastructure

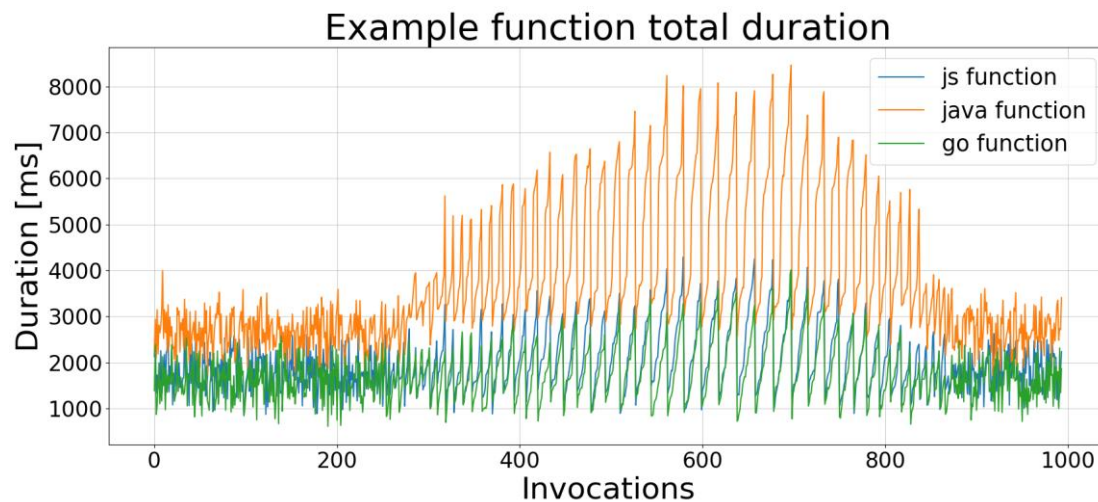


Figure 3.12: Total duration of execution per request – Custom infrastructure

As we can see in figure 3.12, the execution times follow the increase of the workload. Given that the retrievals from MongoDB and the binding times didn't follow the same trend, we can affirm that the trigger represents the bottleneck of the infrastructure.

We replicated the same test switching the dependency provider with two different functions that represent more realistic use case: one that performs authorization based on access control lists and one that performs a logging operation to a file. We obtained results similar to the ones presented above, with the only difference being that the dependency provider duration is a bit higher.

3.5 Final considerations on the first architecture:

Implementing dependency injection at the function level is simple and straightforward, and it is easily replicable on different platforms. The problem adopting this solution is mainly one: the injector suffers from all the disadvantages of functions, including cold starts and ephemeral state. In fact, the injector lives in the same lifespan of the functions, so it can't take advantage of all the mechanisms that require a durable execution environment, such as caching and connection pooling to manage in a more efficient way the retrievals from the registry. To achieve this possibility, it is necessary to move the injector to another durable component of the FaaS infrastructure: the invoker.

In the rest of this work, we will discuss how to enhance the performance of the first presented solution, and at which cost. Given that public clouds' platforms don't allow users to modify the underlying infrastructure yet, we will focus on the on-premises infrastructure in order to add an injection support at the level of the invoker.