1- ON-PREMISES INFRASTRUCTURE

1.1 Introduction:

As we already said, to improve and optimize the architecture implemented on AWS Lambda and Azure Functions, we need to switch to an on-premises infrastructure in order to have full control on the underlying components that allow to run serverless applications. The starting infrastructure we have available is basically composed of two components: message-oriented middleware (MOM) and invoker. So, its architecture can be represented as follows:

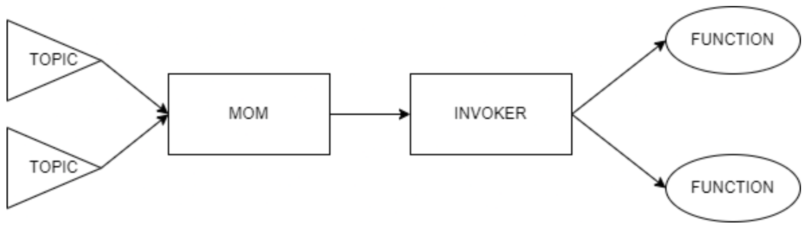


Figure 1.1: On-premises infrastructure architecture

The idea behind the infrastructure is to use the topics as triggers for our functions. In this way, we can associate a specific invoker to make it execute the correspondent function in response to the publication of a message on that topic. In the following paragraphs we will discuss about the technologies used to implement the infrastructure.

1.2 NATS:

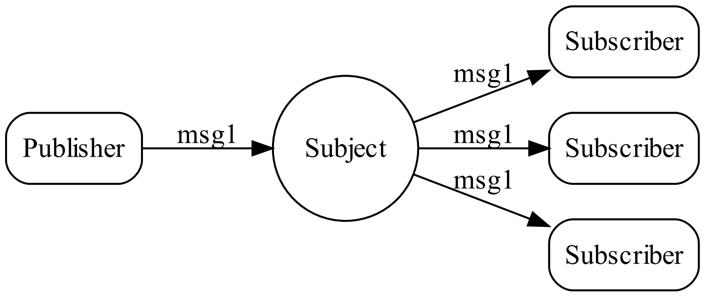
NATS is a high performance, open-source MOM designed to facilitate real-time communication between distributed systems, applications and devices. The NATS services are provided by one or more NATS server processes that are configured to interconnect with each other and provide a NATS service infrastructure. 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 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.

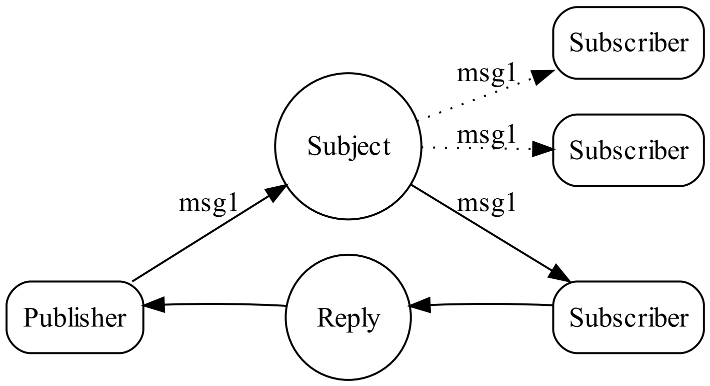
- Subject subscriptions are automatically propagated within the server cloud.
- Messages will be automatically routed to all interested subscribers, independent of location.
- Messages with no subscribers to their subject are automatically discarded.

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:

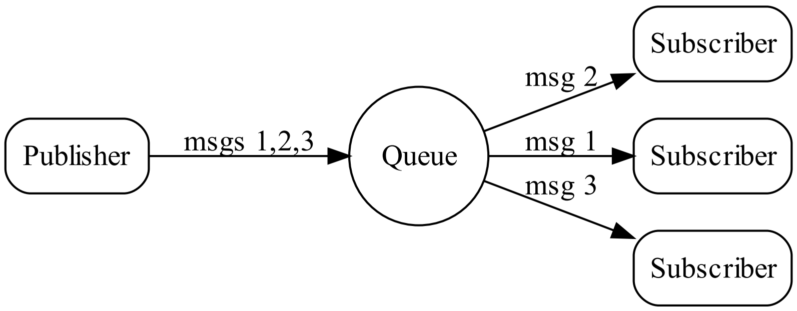
- Traditional publish-subscribe message distribution model: A publisher sends a message on a subject and any active subscriber listening on that subject receives the message.



- 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.



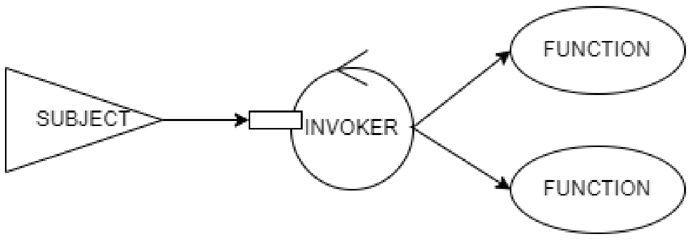
- Queue groups: NATS provides an additional feature named "queue", which allows subscribers to register themselves as part of a queue. Subscribers that are part of a queue form the "queue group". In this way, when a message is published on a subject, only a randomly chosen subscriber of that subject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

1.3 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, 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 deliver 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. In fact, if we keep in mind the traditional architecture of a FaaS platform, it is possible to notice that we have just used NATS as controller and its subjects as triggers, and our invoker acts as a standard invoker that associates to incoming events the execution of a certain function. Starting from this infrastructure, we will concentrate on how to implement dependency injection and use it in our functions.

2- FIRST ARCHITECTURE

2.1 Introduction:

As we have already done for the preliminary tests on public clouds, we want to implement our first dependency injection architecture on the new available infrastructure. Basically, the previous solution consisted of four elements on top of the FaaS platform:

- Client: the function that needs dependencies to be injected.
- Dependency provider: the dependency that should be injected.
- Injector: SDK that provides an API to perform CRUD operations (registration and retrieval of services) on a registry.
- Registry: storage for the dependency declarations.

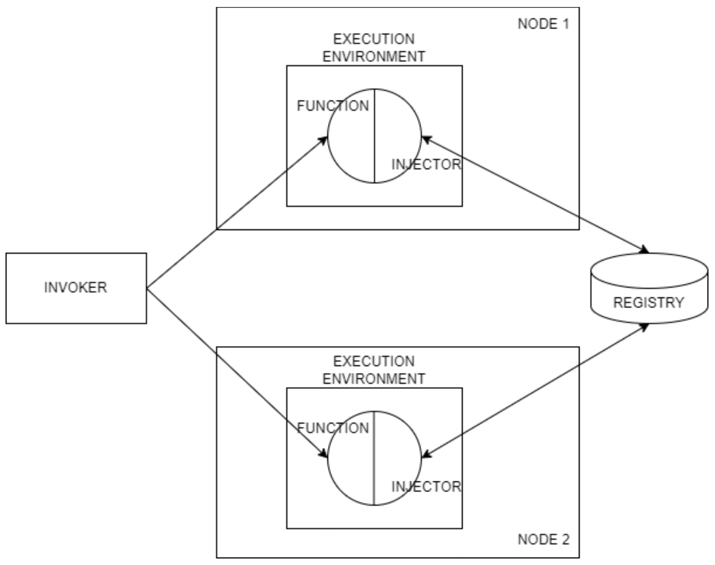


Figure 2.1: first architecture on public clouds.

For the previous tests, we decided to implement the Injector SDK in three different languages and to use the already integrated NoSQL database services as registries. We also created a simple “hello world!” Go function to use as dependency provider and we exploited the existing monitoring and logging services to collect performance parameters.

In this case, we don't have any kind of integrated feature available, but we need to build the resources that we need starting from the infrastructure presented in the previous section. Furthermore, we will use more significant dependency providers in order to replicate a more realistic use case. In particular, we will inject two functions that implement access control and logging features. In this chapter, we will discuss about the technologies used to implement the first architecture and how we put together all the components that we need. We will finally evaluate the performance of such a solution and try to optimize it.

2.2 Architecture:

Given all the consideration made in chapter 1, and keeping in mind Figure 2.1, we need to make some considerations before we start. As we already said, the invoker acts as an infinite loop that waits for messages to be published on a specific topic and creates a new process that executes the code of the associated function every time a message arrives. So, we will need to run an invoker for every function in our architecture, with the possibility to run more instances of the same invoker in order to load balance the upcoming messages. In fact, thanks to the "queue groups" feature of NATS, the balancing is offered for free by the MOM, since that the invokers of a same function will subscribe to the same group.

To enable communication between functions it is possible to use the messages published on NATS. So, it is possible to uniform the message payloads in order to make them serializable and deserializable for all the invokers. We decided to format the payloads as JSON objects, so that they can be easily managed with all the programming languages. In fact, we will implement the Injector SDK with the same languages used on public clouds (Java, Javascript and Go).

With regard to the registry, we decided to implement it using MongoDB, a popular NoSQL database designed to handle large volumes of unstructured, semi-structured, or structured data efficiently. In this way, we can represent dependencies as a tuple composed of three elements: identifier, name and topic. In fact, in this case the triggers are the topics our functions subscribe to, through their invokers, not the web URLs.

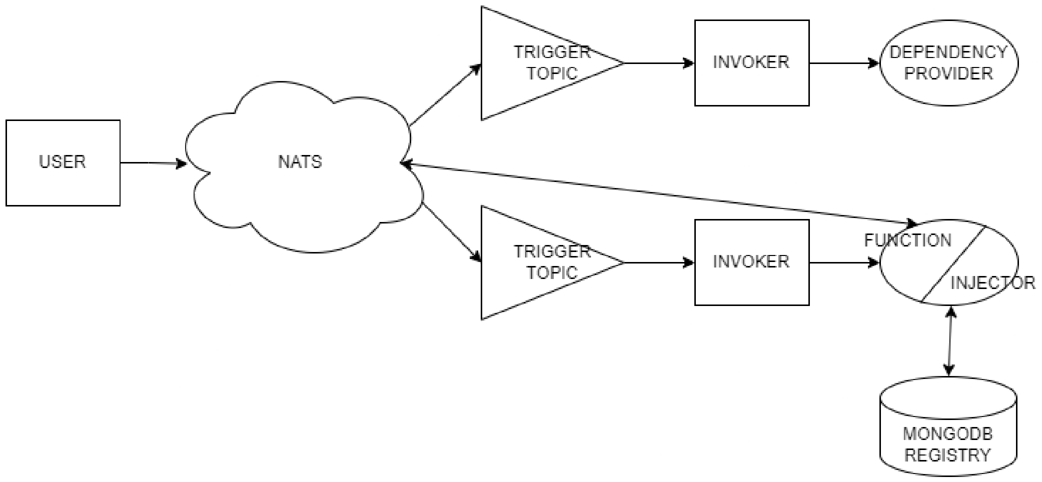


Figure 2.2: first architecture on the new infrastructure

In this way, we can replicate the same behavior of the tests conducted on the public clouds, in order to evaluate the performance differences and make further improvements.