ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

TESI DI LAUREA MAGISTRALE

IN

INFRASTRUCTURES FOR CLOUD COMPUTING AND BIG DATA M

PERFORMANCE EVALUATION OF FUNCTION COMPOSITION IN MIDDLEWARES SUPPORTING FAAS FOR SERVERLESS COMPUTING

Candidato

Corrado Proietti

Relatore

Chiar.mo Prof. Antonio Corradi

Correlatore

Dott. Andrea Sabbioni

Anno Accademico 2021-2022

Sessione IV

Summary

Abstract

The present experimental thesis entitled Performance Evaluation of Function Composition in Middlewares supporting FaaS for Serverless computing is the result of a first phase of research and literature analysis in the field of FaaS architectures and of a second experimental phase, in which qualitative conclusions were drawn about the adoption of an innovative architectural model.

The concept of Serverless Computing is a new and exciting aspect of cloud computing that involves the deployment of small pieces of software applications and services as serverless functions.

Serverless computing architecture enables the cloud provider to fully manage the execution of a server's code, eliminating the need for customers to develop and deploy the traditional underlying infrastructure required for running applications and programs.

Even though big tech companies are extensively utilizing serverless computing in their products and investing billions on this novel but affirmed technology, it is affected by various problems still considered an open field in research.

In fact, by definition, FaaS architectures are geographically dislocated and consequently subject to event propagation delays that can significantly degrade the overall system performance. What is generally done, is to reduce as much as possible cumulative delays especially if attributable to the infrastructure itself that could determine a greater or lesser competitiveness on the market.

The background idea, which becomes the leit motiv throughout this work, is to develop and assess the performance, and thus the validity, of a Message-Oriented Middleware-centric serverless platform architecture promising to enable advanced analytics capabilities and better overall performance, without renouncing the essential characteristic of scalability in the context of distributed systems. Experiments in emulated conditions show that applying the MOM coordination co-locality principle improves the end-to-end delay and data processing performance.

Introduction

The advent of the Internet and the growth of the World Wide Web at the tail end of the 1980s solidified the crisis of the centralized model in substitution of more decentralized models. The prospect of being able to take use of computer connections on a worldwide scale has made it possible for a multitude of companies to provide an ever-expanding selection of services that can be accessed and utilized through the Internet. The enormous increases in the amount of data that is transferred over the network has, however, been accompanied by the need for companies to lowering the adoption step curve for distributed services to better match the emerging market needs.

There was a noticeable shift in preference towards solutions that offered faster time-to-market by implementing cloud adoption and making it more economically feasible. This trend was driven by the need to reduce development time and costs, while also delivering products and services more quickly and efficiently to meet customer demands. As a result, the very first solutions that would later come to be known as Cloud Computing started to take shape and refers to the act of making computer resources available to users in the form of services that may be accessed via the usage of the Internet.

The first chapter introduces cloud computing as a technology that has emerged in the last decades, starting from a historical treatment that has justified its rapid development and adoption both in the business and consumer sectors. Will be also argued the technological context which cloud computing has taken inspiration from for its development and point out the motivations behind the model. Among the multiple advantages in terms of pricing, dependability, and scalability cloud computing offers, there are also some disadvantages, especially in terms of performance that still represent an open challenge in the academic front.

A FaaS platform adheres to the so-called serverless computing model, hiding to the users the infrastructure and the management thereof, tasking the consumers only with the creation of the business logic functionality. Serverless computing enables developers to spend more time on the creation of application logic, programming,

and writing code in terms of functions and events. This frees the cloud provider from the responsibility of maintaining the complexity of the underlying infrastructure.

Subsequently, the second chapter, exposes that function chaining, from the performance perspective, represents one of the most relevant limitations of such technology with penalties and overall system responsiveness degradation. Moreover, starting from an in-deep literature about FaaS and the current state of function composition approaches, the chapter proceeds highlighting what are the open challenges on that.

Even though latencies can materialize not only from a bad user-defined chaining logic but also from inefficient infrastructural support, the fourth chapter will highlight the proposed solution to such problems implementing a MOM-based function coordination infrastructure exploiting the so-called reflective invocation interaction model.

For the aim of this thesis and in order to put this proposal into action, an investigation of the many options now available on the market for message-oriented middlewares that include message interception as a defining trait will be carried out. The purpose of this implementation, which can be found in chapter three, is to formalize a workflow proposal that will be helpful in the building of a testbed that is able to test the potential of this model.

Since FaaS architectures are inherently agnostic on the execution state, but more significantly on the location of the deployed functions, these can be deployed both in single-cloud contexts, with mostly low latency times, and in multi-cloud contexts, where times are significantly higher. Beginning with the suggestion given in the prior paragraph, chapter four will proceed to discuss the two deployments that have been suggested in order to recreate the aforementioned situations.

The fifth chapter will first discuss the many sorts of tests constructed, from the perspective of interacting actors and from the kind of workloads proposed, and then it will present the outcomes of the executions of these tests. This will allow for some definitive conclusions to be drawn about the work that was carried out. These findings will aid to highlight the benefits and drawbacks of the solution that has been offered in terms of its performance and the resources that are involved.

Ultimately, in the sixth and last chapter, we want to draw conclusions and considerations on the results obtained and gives inspiration for possible future developments.

1. Cloud Computing

The study of distributed systems is an active area of research for more than seventy years. It has also played an essential part in the field of computer science, since it was instrumental in the development of the Internet, which is actually essential to all aspects of contemporary life. In addition, distributed systems have continued to develop as a result of a variety of changes and resulted in the creation of new kinds of computer systems as well as the modification of previously established paradigms, ranging from client-server up to what is called cloud computing. The essential traits and model components, on the other hand, have stayed generally unchanged throughout time, with the current paradigm enhancing (or re-engineering) technologies from earlier paradigms.

Distributed systems have gone through a continuous evolution as a consequence of technical breakthroughs and the shifting roles they play in society. The effect of each transformation has ultimately been the establishment of a new paradigm. Each new distributed system paradigm, where the most prominent example of which is cloud computing, makes it possible for new forms of commercial value to be created, but it also ushers new research challenges that need to be addressed in order to realize and improve the operation.

Recalling the NIST definition of Cloud Computing [1]:

"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 cloud model is composed of five essential characteristics, three service models, and four deployment models.", it comes up that Cloud Computing relies on a set of interconnected resources territorially distributed which need to communicate to synchronize and coordinate themselves to fulfil their aim. Hence, a reliable networking architecture layer is a crucial component of Cloud framework because users interact with services disregarding the location the request comes from.

Cloud computing, in its broader definition, is indeed the delivery of computing resources over the net, adding up to easy management and access even by non-expert

users. When we store our files online in its place of our domicile computer, or utilize webmail or a social networking site, we are using a "cloud computing" service.

From an historical perspective Cloud Computing, intended as the mean of aggregating computational and storage resources to whoever wanted them, is not a recent idea.

In the 50s, mainframes represented a step toward what today is being called as Cloud Computing, in fact, users might access these huge room-sized computers through terminal in a shared fashion. One of the main drawbacks of such approach were that user had to reserve computational time, keeping calculus as short as possible, in order to free these resources for others.

Universities and companies rented out computation time on mainframe computers because this was one of the available ways to access computing resources as they were too expensive to be owned by individuals.

By the 60s the main underlying ideas of Cloud Computing started to come to light, conceptualization of computing resources as public utility and the possibility of a networks of computers that would allow people to access data from anywhere in the world.

Using virtualization software, it became possible to execute one or more operating systems simultaneously in an isolated environment. Complete (*virtual*) computers could be executed inside one physical hardware which in turn can run a completely different operating system. Virtualization played a key role in driving technology forward and was a significant contributor to the advancement of communication and information.

Virtualization, in fact, uses software to create an abstraction layer over computer hardware that allows the hardware resources of a computer, such as processors, memory, storage to be divided into several virtual computer or virtual machines (VM). Each VM runs its own operation system completely isolated with respect of other VMs even though it is running the same underlying hardware resources.

Such technology was and is still a valuable offer for mid-large sized company because enables more efficient utilization of physical computer hardware with a greater return of investment.

1.1. Model of Deployment

Cloud Computing has only become a mainstream reality and widely used term in the early 21st century with the support of major technology companies.

The idea of Cloud Computing started to emerge when, initially, companies had to expose their services to the web. Indeed, with growing expansion of the Internet and users, companies had to give the most seamlessly experience toward such services but containing the cost and thus contributing the expansion of the cloud computing architecture.

As it often happens in traditional software architectures, Cloud Computing could be categorized in different categories and since they can all provide different capabilities, the distinction usually falls in these three main types of cloud computing: Public Cloud, Private Cloud and Hybrid Cloud.

In simple terms, a Public Cloud is a vast array of readily available to the public compute and software resources such as networking, memory, central processing unit (CPU) and storage. These resources are globally hosted, distributed into data centers and fully managed by vendors ready to be rent and used to build IT infrastructures. In short, Public Clouds are cloud environments typically created from IT infrastructure not owned by the end user.

Traditional public clouds always ran off-premises meaning that the infrastructure is not owned by the final customer even though today public cloud providers have started offering cloud services on clients on-premises data centers. This has made location and ownership distinctions obsolete.

Even fee costs are not necessary characteristics of public clouds anymore, since some cloud providers (like the *Massachusetts Open Cloud*) allow tenants to use their clouds for free.

Accessing your resources in this type of cloud can be a simple task as using a web browser.

One of the great benefits of the public cloud is that the underlying hardware and logic is hosted, owned, and maintained by each of those vendors. This means that customers have no responsibility for buying or maintaining the physical components that make up their public cloud IT solutions.

The *pay-as-you-go* model used to charge for these resources makes them a more cost-effective solution than owning them as you only pay for what you consume. The ability to scale the size of your solution up to accommodate for the peaks and troughs in usage saves the customer money and gives huge flexibility.

Financially backed Service Level Agreements (SLAs) commit each vendor to a monthly uptime percentage and guarantee of security in line with standards. The public cloud vendors have invested, and continue to invest, tenths of billions of euros in their data centers so they can be provisioned with state-of-the-art fault tolerant power supplies, network paths, storage facilities and automated monitoring and maintenance systems to meet these SLAs.

Private clouds, instead, are owned and used by single private or organisations. They have traditionally been physically located at the business own data center using its own hardware.

However, a business may employ a third-party provider to host their private cloud on their kit. In that scenario, private cloud does have some similarities to public cloud in that the resources are in a remotely managed data center. However, this emerging trend about private clouds foresees that data centers are located off-premises, meaning they are not situated on the organization own property. However, although these providers will offer administrative services, they will only be able to offer a tiny percentage of the global services of a public cloud.

Basically, all clouds become private clouds when the underlying IT infrastructure is dedicated to a single customer with completely isolated access.

This new approach to building private clouds eliminates the need for organizations to abide by traditional location and ownership rules, as they are now able to access and use vendor-owned data centers from any location. This provides organizations with more flexibility and scalability when it comes to their cloud computing needs. If the private cloud is being hosted in your own data center, instead, then you can tailor your cloud computing approach to your own preferences and internal processes. Some of the more stringent security and compliance legislation insists on certain types of data and resources being kept inside your own security boundary - a self-hosted private cloud is the perfect tool to enforce these policies.

Ultimately, a hybrid cloud solution offers benefits from the best of both options and makes cloud bursting possible. In the example of extending your private cloud network, this means that if you are running out of compute capacity on premise, it can be supplied by the public cloud. This is a cost-effective way for businesses to increase compute capacity on demand while still utilising the already paid for on premise resources.

1.2. From Single to Multi Cloud

Single and multi-cloud are two different approaches to cloud computing that companies can consider when deciding how to manage their IT infrastructure. Single cloud refers to the use of one Cloud Service Provider (CSP) for all cloud-related needs, whereas multi-cloud refers to the use of two or more cloud service providers.

There are benefits to being multi-cloud, as it can provide more options and flexibility when it comes to cloud computing. For example, companies can take advantage of different CSPs strengths and capabilities, such as better high availability and disaster recovery, lower costs, and more diverse feature sets. Multi-cloud can also help companies reduce to a single CSP, which can be beneficial in case a CSP becomes a competitor to the company or experiences an outage or other issues. Another benefit is that by implementing a multi-cloud strategy, enterprises may guarantee uninterrupted service and reserve cloud services as a crucial component of their disaster recovery and business continuity strategies.

Furthermore, multi-cloud can help companies deal with capacity issues, as they can rely on another CSP for additional capacity and address missing features or products that are available on one CSP but not on another. This can be particularly important for regulatory or government reasons, as different CSPs may have different cloud regions or capabilities in specific areas.

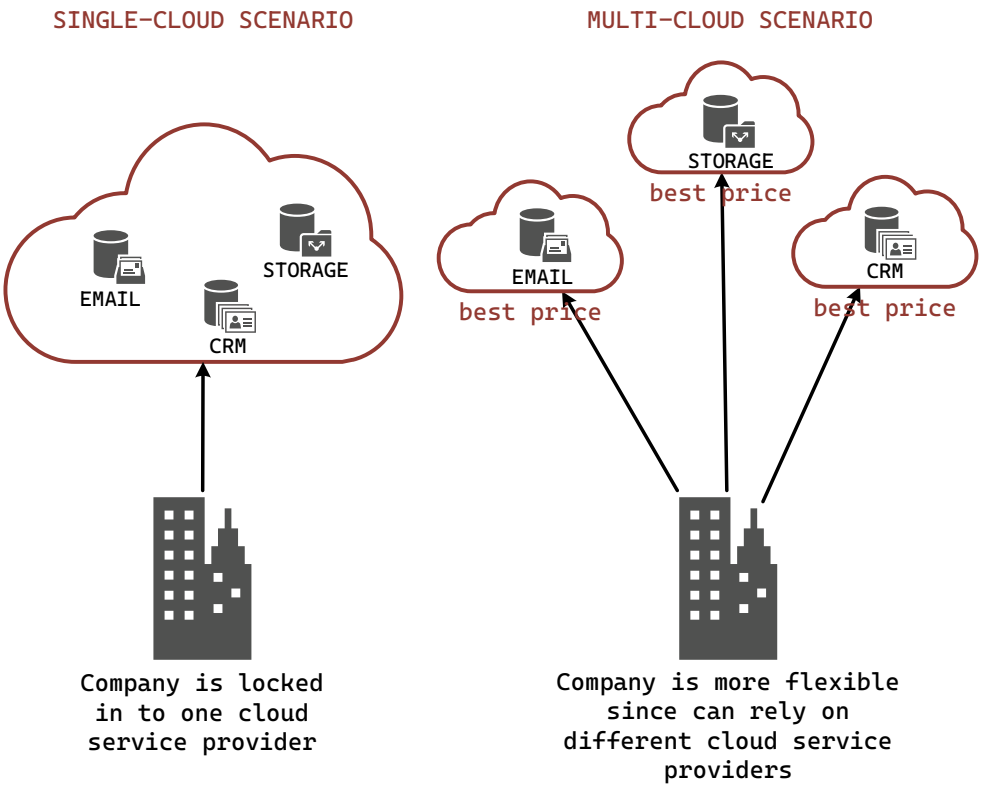

Figure 1 - Single-Cloud and Multi-Cloud Strategies

However, there are also downsides to multi-cloud that companies need to consider. One major issue is the potential for reduced performance, as moving data and computing between clouds can be slow and cumbersome. This is because CSPs are not designed to work together, and each CSP focuses on their own components, which can create challenges for interoperability.

Another drawback is that multi-cloud can increase costs and require companies to have a more extensive skill set, as they must understand how to use multiple CSPs. This can create additional management overhead and administrative complexity, as well as increased switching costs and limited access to high-value services.

Finally, multi-cloud can expose companies to increased risks and security concerns, as data is spread across multiple clouds instead of being consolidated in one place. This means companies must be vigilant about managing and securing their data across multiple CSPs, which can be a daunting task.

Overall, the decision to use single or multi-cloud depends on a company specific needs and priorities. While multi-cloud can provide benefits in terms of flexibility and reduced lock-in, it can also create challenges in terms of performance, costs, and security. As such, companies need to carefully consider their options and make an informed decision that aligns with their goals and objectives.

1.3. Models of Service

With the term *as-a-Service*, it often refers to a cloud computing service managed by an external provider on behalf of the user, which can effectively focus its efforts on business related strategy tasks. Each kind of cloud computing offers the possibility to delegate the management of an increasing number of components of the on-premises infrastructure.

On-premises IT infrastructure brings a high level of responsibility with respect of users and manager. In fact, when hardware and software are hosted on-site the upgrading, replacing, or managing of such components are company and its teams concerns. In this context, cloud computing, allows to delegate partially or totally to a third-party actor these duties.

As the user delegates a larger set of its IT infrastructures, three kinds of cloud computing emerge, each one with an increasing degree of management left to the user: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS).

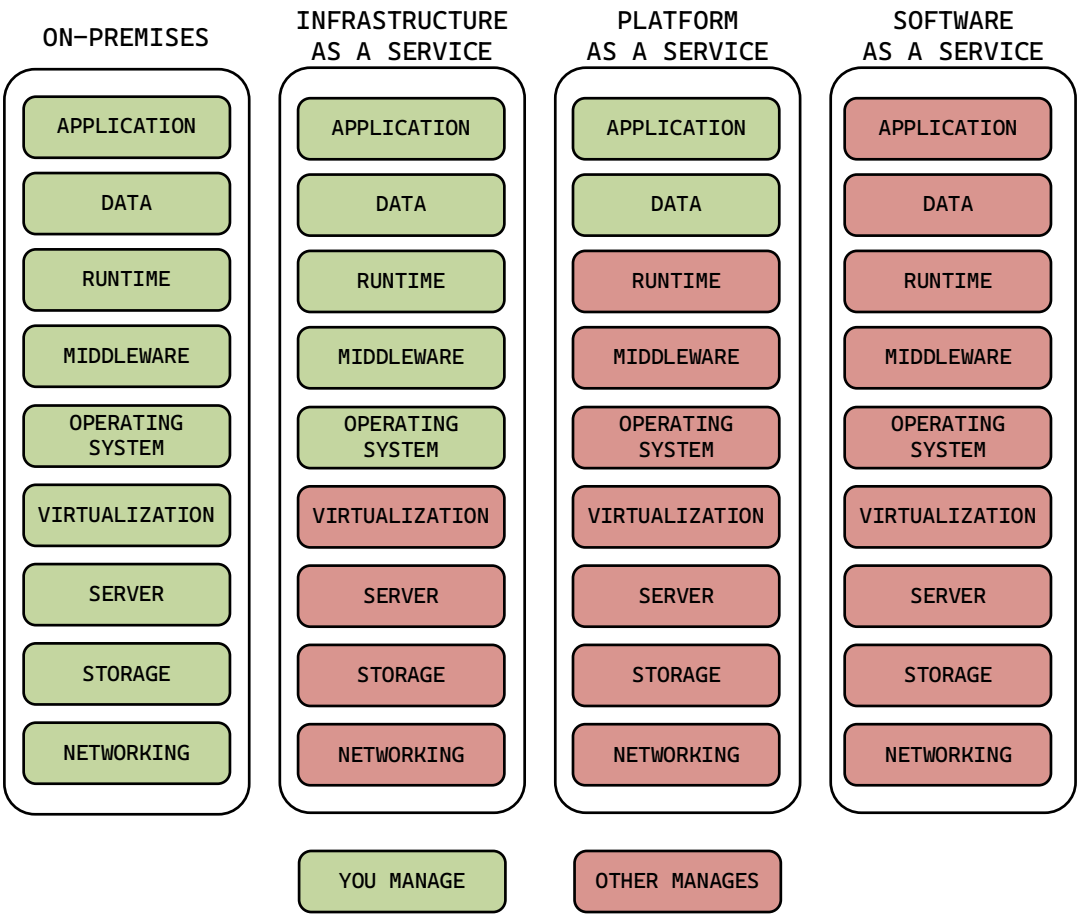

Figure 2 – Models of Service

In reference of Figure *3*, Infrastructure-as-a-Service, or IaaS, represents the first level of delegation relocating to the cloud the management of the on-premises IT infrastructure. It usually relies on a pay-as-you-go plan meaning that the users pay what resources, such as storage or virtualization, and how much time the user effectively consumes.

In short, user is responsible for the operating system and data, applications, middleware, and runtime, while the cloud provider grants access to and management of network, server, virtualization, and storage. There is no need to manage or update the on-premises datacenter, which is done by the provider. The user has complete control of the infrastructure through an Application Programming Interface[1] or a web dashboard.

[1] Abbreviated as API

This model offers considerable flexibility allowing the user to purchase only the components it needs, adding or deleting them when and how needed with the advantage of low fixed and no-maintenance costs.

It is a fast and flexible way to create and then retire test and development environments. You can use only the infrastructure you need to create the development environment, grow, or shrink it, stopping once you get what you want and paying only for the services you use. IaaS also has some drawbacks, such as potential issues related to service reliability, provider security and multitenant systems where the provider needs to share infrastructure resources with multiple customers. However, these issues can be avoided by choosing a reliable and qualified provider with a consolidated experience and reputation. Examples of IaaS from public cloud providers are Amazon AWS, Microsoft Azure, and Google Cloud.

Platform-as-a-Service, or PaaS, instead, lays another abstraction layer on top of the infrastructural layer adding integrated stacked solution or services.

Mainly targeted for developers and programmers, PaaS cloud services offers a platform where the users could develop, execute, and manage its applications without taking into account all the management aspects related with the platform itself.

The user is therefore focused on coding, building, and managing the application disregarding software updates or hardware maintenances.

Giving an environment where to create and distribute applications, providers let developers build modular and integrated applications reducing the time spent in coding.

At the time this work is being written, most popular PaaS providers are AWS Elastic Beanstalk, and Red Hat OpenShift.

Software-as-a-Service, or SaaS, known as cloud application services, is the most comprehensive cloud services seen up to now, and it is about distributed business-level applications accessed by users typically via web browser.

The provider takes care of software updates, bug fixes, and other general software maintenance tasks, while the user connects to the app through an API or dashboard. There is no software installation on individual machines and groups access to the program is more streamlined and reliable.

A typical SaaS service we are all familiar with is when you have an email account from a web service like Outlook or Gmail since logging into your account and using your mail from any device, you are actually using a SaaS cloud service. SaaS is an optimal option for small businesses that do not have enough staff or bandwidth to handle software installations and updates, or for applications that require minimal customization or are used only sporadically.

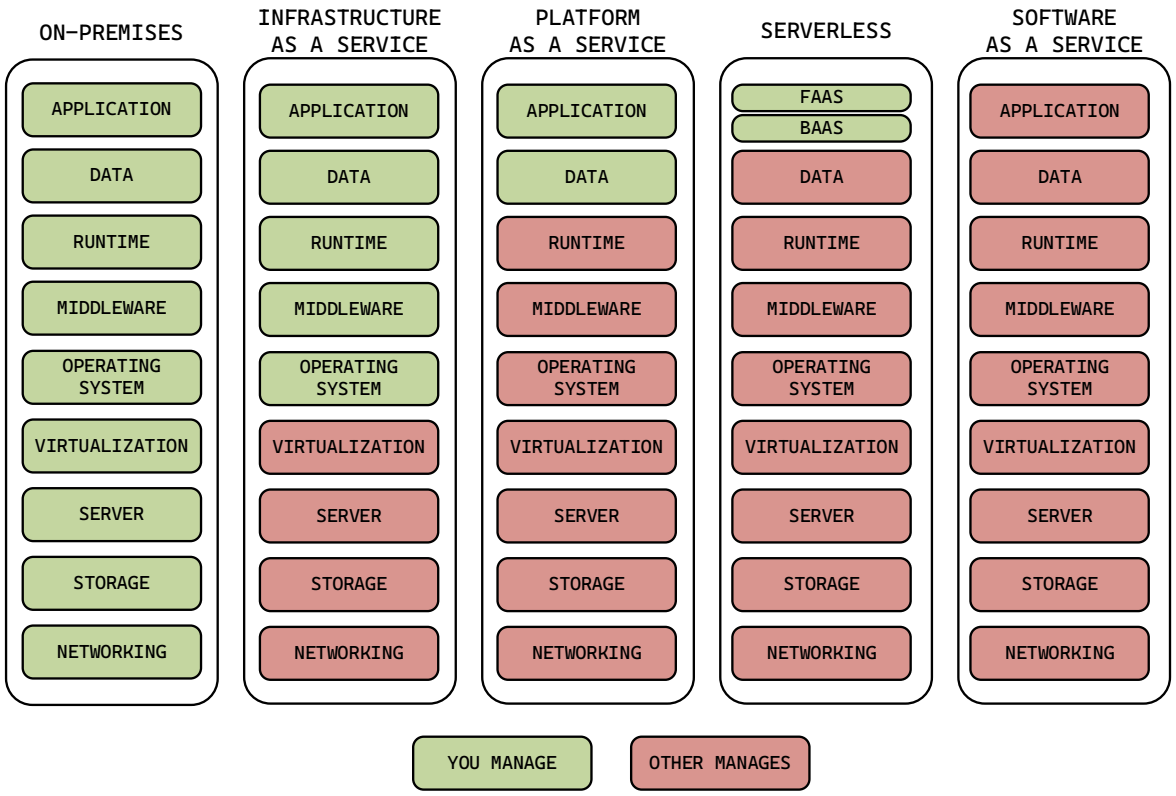

Figure 3 - FaaS Service

Function-as-a-Service, or FaaS, is a cloud computing model in which the key elements are the functions triggered by either external or internal events.

It sits between the PaaS and SaaS because it actually runs on dedicated middleware capable of managing the function lifecycle; basically a PaaS but not a complete business level, ready-to-use application as SaaS are.

Such functions are typically hosted in stateless container but since these containers cannot contain application state (either user, session or application state) it is usually stored outside the boundaries of the container itself via a cloud storage service.

Concisely, developers need to build and execute functions bundled as application packages, without maintain their own infrastructure resulting in applications made up of several functions.

However, when the requirement for fine-grained state sharing arises, the ephemeral nature of functions creates a number of problems that necessitate effective, scalable, and affordable storage solutions. At the same time, the opaqueness of functions makes it more challenging to implement efficient mechanisms for process coordination, such as broadcast [2], aggregation and shuffling, which are common communication primitives in distributed systems.

As a consequence of FaaS deployment criteria, the resulting programming model is inherently stateless as there is no guarantee that subsequent invocations of the same function will be executed in the same environment, or that the resources allocated for the function execution are not reclaimed after its termination. In this one-to-one mapping between functions and triggering events, FaaS platforms achieve finer-grained scalability: at any moment, the computational resources allocated to handle user requests match the ingress load.

FaaS solutions are generally available on leading public clouds whilst provision can be done on-premises, providing a net increase in capacity for enterprise IT departments involved in app development.

Although it is typically a cloud computing platform that uses cloud computing services, the model is expanding to include on-premises and hybrid deployments as well.

FaaS infrastructure is typically measured and used on demand by the service provider thus, being present when needed, it does not require server processes to run consistently in the background.

Due its execution model, FaaS functions boast minimal memory footprint and great scalability compared to traditional distributed applications.

There are architectural constraints in order to enable dynamic scalability, like time limits on the execution of a function. Function needs to be something that can start up and run quickly starting in milliseconds and process individual requests. As the demand increases and there are many simultaneous requests, the system creates

the number of copies of the function needed. As demand decreases, the application automatically deletes unnecessary copies.

Dynamic scalability is an advantage of the FaaS model as providers only charge for the resources used and not downtime while remaining a model perfect for high-volume transactions and sporadic workloads such as reporting, image processing, scheduled tasks, data processing in IoT environments and mobile or web apps. As for SaaS, FaaS providers let customers to interact with functions through APIs conceived as a form of contract, with documentation representing an agreement between the parties, simplifying the integration of new application components into existing architecture, promote systems integration.

In the next chapter what is going to be examined, after a short but comprehensive historical framing, are the reasons why serverless computing started to come up as emerging technologies, which is the typical FaaS deployment scenario and how Message-Oriented Middleware could exploit the potentiality of such technologies.

2. Function-as-a-Service

FaaS architectures, as can be guess from the previous chapter, is a type of technology that does not arise from a specific need, or as better to say, is born after a long process of improvement and optimization of existing architectures. The reasons that push the development of distributed technologies, as we will see, is not exclusively due to performance requirements, but also to economic and organizational needs.

Although virtualization technology dates back to the 60s, it only became widespread in the early 2000s. Hypervisors were developed decades ago to provide multiple users with simultaneous access to computers with active batch processing. However, over the following decades, the problem of having multiple users on a single computer was addressed by choosing solutions other than virtualization such as the development of Time-Sharing Operating Systems.

Around 90s, most companies used physical servers and IT stacks associated with a single vendor, which did not allow existing applications to run on hardware from other vendors (commonly known as *lock-in problem*). Companies then, began to upgrade their IT environments with less expensive equipment from various vendors. However, they were tied to poorly used physical hardware, since each server could only perform one task. Without virtualization, enterprises, would have required separate servers with each one with one OS and the required software.

That is when virtualization began to spread. Server usage has been made more efficient, in some cases abandoned, thus reducing the costs associated with purchasing, configuring, cooling, and maintaining.

Virtualization [3] brought huge improvements in business efficiency and costs prediction accuracy for various cloud application workload patterns.

Virtualization technologies, to allow the efficiency of hardware infrastructures and therefore the installation of multiple operating systems, require a software component called Hypervisor.

The Hypervisor, as shown in Figure *4*, is capable to virtually abstract the underlying hardware and present it to the different virtual machines. In this way, the operating system will see this abstraction as if the hardware were exclusively

dedicated to it when, instead, the different virtual machines compete for the acquisition of these resources.

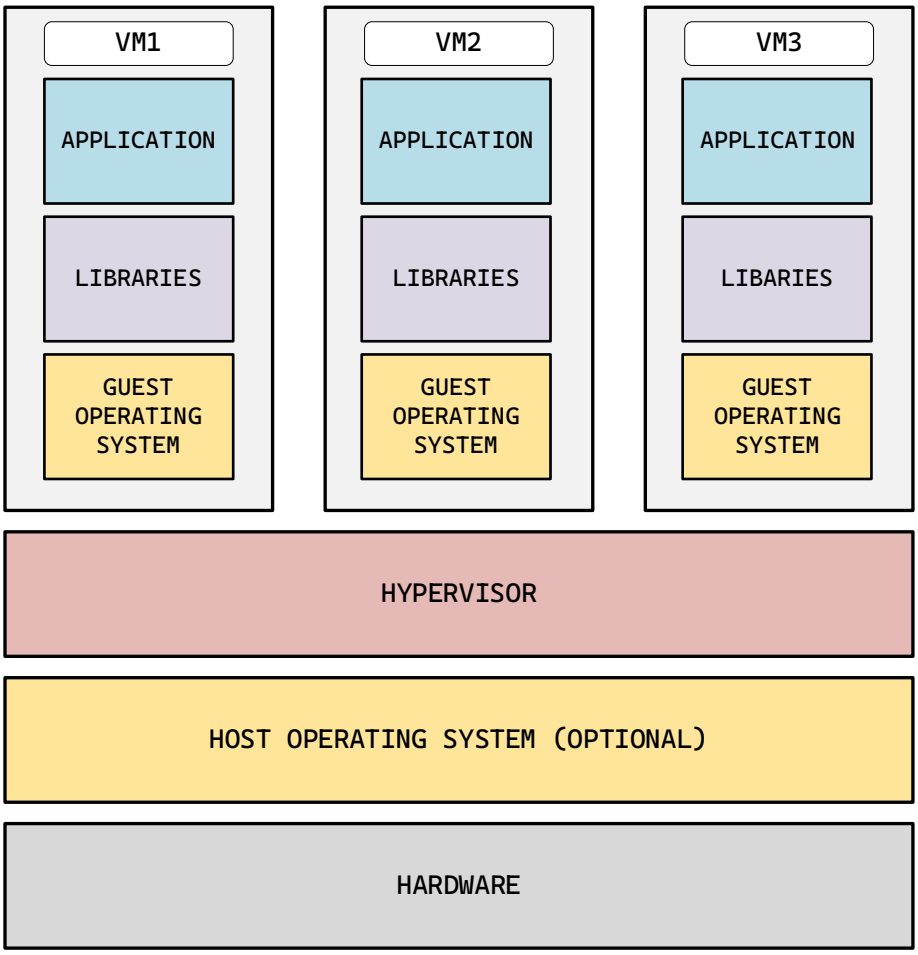

Figure 4 – Typical Virtualization Layers

In short, virtualization brought various kinds of benefits though:

- **Resource efficiency:** Before virtualization, each application server required its own dedicated physical resources (CPU, memory, ...) and each physical server would be underused. As an example, a single application may use 10-20% of processor capacity while virtualization could improve this percentage up to 70-80% sharing the resources among all applications. This enables maximum utilization of the physical hardware computing capacity.
- **Easier management:** Virtualizing computers with software-defined VMs simplifies policy management through software, enabling the creation of automated IT service workflows. With tools for automated deployment and

configuration, administrators can group VMs and apps as services using software templates, allowing for efficient and consistent setup without manual errors. Virtualization security policies can enforce security configurations based on the VM role and even improve resource efficiency by retiring unused VMs to conserve space and computing power.

- **Minimal downtime:** OS and application crashes can cause downtime and disrupt user productivity. Admins can run multiple redundant virtual machines alongside each other and failover between them when problems arise. Running multiple redundant physical servers is more expensive.
- **Faster provisioning:** Buying, installing, and configuring hardware for each application is time-consuming. Provided that the hardware is already in place, provisioning virtual machines to run all your applications is significantly faster. It can be even automated using management software and build it into existing workflows.

The broad applicability of virtualization has helped reduce vendor constraints and laid the foundation for cloud computing.

However, while virtual machines on clouds give a simple method to increase processing capacity on demand, a cloud provider must forecast how much will be required because starting a virtual machine takes time. Waiting until demand peaks may result in lost income from unfulfilled demands, while buying excessively in advance results in wasted capacity and in unreturned revenues producing expenditures.

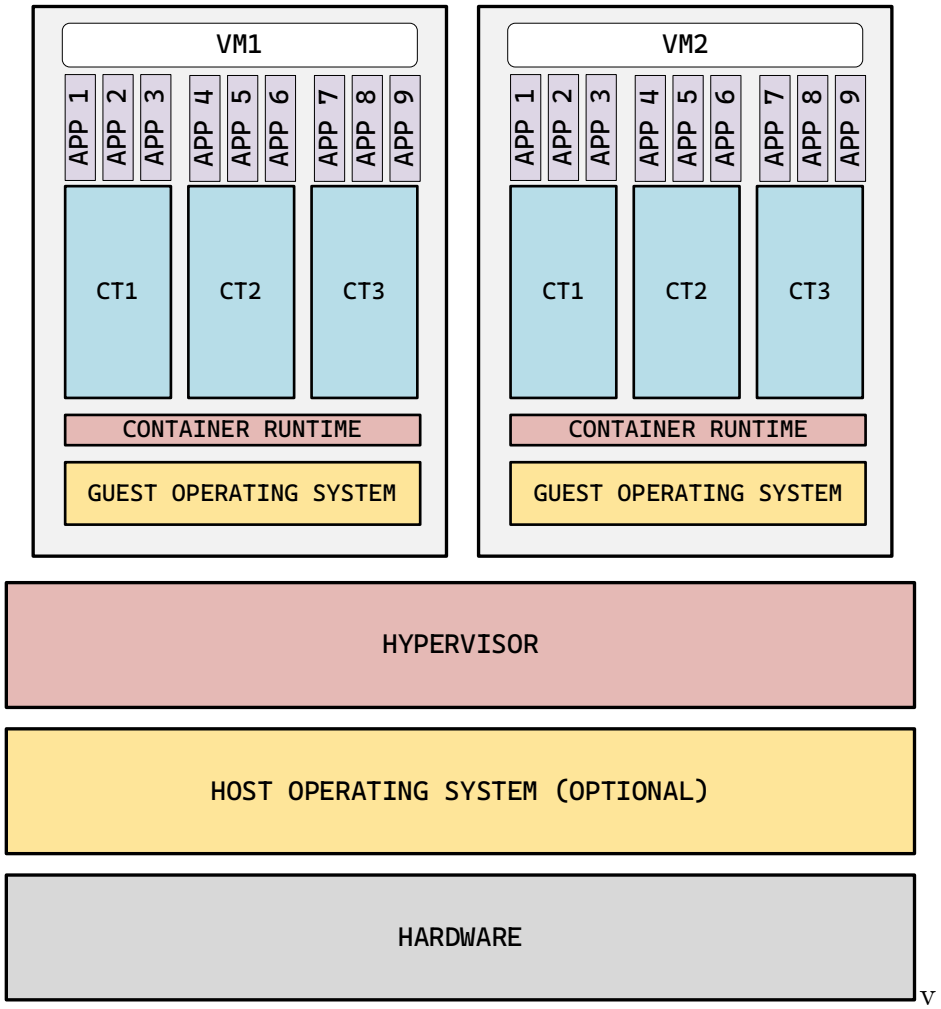

Figure 5 – Typical Containerization Layers

A cloud solution is specifically designed to address these issues and here where containerization comes in.

Containers are a lightweight solution that maintains all the benefits of virtual machines, such as the segregation of users and programs in their own stack on a shared hardware platform, while enhancing speed and better using server resources under fluctuating loads. As shown in the figure below, a container packs up only an application and any associated software, such as libraries, it needs to run without requiring any OS, but it uses the OS of the underlying host for those functions. By losing the OS, a container is much smaller in size, and it is quicker for an instance to be created.

As just said and as shown in Figure *5*, all containers which run on the same physical server use the underlying OS. A container runtime, namely the *containers*

hypervisor or *container runtime* maintains the isolation of each container processes from those of others, while sharing the OS runtime. Thus, it is sometimes said that containerization virtualizes the OS, while virtualization virtualize the hardware.

Given the size reduction of containers, thousands of these self-contained bits of code can run on a server. A bonus that emerges from this technology is that a container is portable from one server to another supporting that same OS allowing developers to create and test a containerized application on a confined environment and then move it to a cloud sever, when appropriate.

A container starts faster than a virtual machine and shares the Operating System with other containers, thus reducing deployment unit sizes and increasing application density per virtual machine.

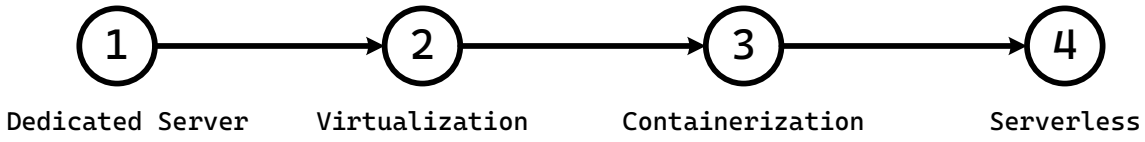

Figure 6 - Evolution from Virtualization to FaaS architectures

To summarize up, as depicted in Figure *6*, when was the case of dedicated servers deployments, different applications were distributed on different physical server; in consequence, servers were over dimensioned and used in an inefficient utilization rate.

With virtualization in use, instead, the application density on bare metals was increased. Even though this was a huge resources improvement, time by time, companies became aware of the fact that virtual machine images, the minimum deployment unit, were very large.

To obtain the same level of virtualization isolation and consolidation but pragmatically operate more than one application per virtual machine, and thus reducing their deployment size, containerization was the new emerging trend.

But a container still requests a share of CPU, memory, and storage. What was desirable were to keep services as much resource efficient as possible. Even if the provided service is hardly requested, it continues to consume resources during the

inter-time between incoming requests resulting in a waste of computational resources.

The *FaaS runtime* must ensure that services timeshare a host, in the sense that as long as the service is not being used, the resources bound to it are deallocated. This assumes that the FaaS model of deployment, follow a serverless architecture style.

Initially, FaaS and serverless indicated rather similar concepts. Over time, the meaning of serverless has expanded to include a broader set of architectural patterns and practices that extensively use common services in addition to custom business logic coded in FaaS. Serverless can be used by microservices and traditional apps, as long as they are containerized matching dynamic scalability and state management requirements of the infrastructure itself.

2.1. Serverless

We said that typically, when developers talk about serverless, they are referring to the FaaS model, two concepts often mistakenly interchanged. The serverless, from a broader point of view, is everything related to server-side logic, possibly with state maintenance, built through multiple separate services, running on an infrastructure invisible to eyes of the developer. As for the relationship between serverless and FaaS, it is more correct to say that the first includes the second.

Even with FaaS model, developers must write custom server-side code, but in this case that code, tipically, runs in containers that are fully managed by the cloud provider.

The term serverless is also used to refer to managed services such as databases and messaging systems, which do not require the intervention of a developer or administrator because they are, in fact, managed by a cloud provider or in general by a third party. The combination of FaaS services and common back-end services, such as database, messaging, and authentication, primarily connected through an event-driven cloud-native architecture, is what enables serverless developers to reap the greatest benefits.

The ability to *scale-to-zero* [4] instances is one of the critical differentiators of serverless platforms compared with container focused PaaS, or virtual machine focused IaaS services.

Scale-to-zero makes it possible to avoid being charged for always-on components, which eliminates the priciest cloud consumption behaviour. Cloud-native architecture and technologies are an approach to designing, building, and operating workloads built in the cloud and taking full advantage of the cloud computing model.

As stated by Cloud Native Computing Foundation [5]:

"Cloud-native technologies enable organizations to build and run scalable applications in modern, dynamic environments, such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.", these technologies enable weakly coupled systems that are resilient, manageable, and observable. Combined with reliable automation, they enable to make frequently and predictable high-impact changes with minimal fatigue.

Although servers are still used in this model, they are abstracted from application development.

In fact, the authors of the Serverless framework [6], a popular tool thanks to which you can build serverless applications capable of running on all leading-edge cloud providers, stated:

"Just like wireless internet has wires somewhere, serverless architectures still have servers somewhere. What 'serverless' really means is that, as a developer you do not have to think about those servers. You just focus on code. You do not have to actively manage scaling for your applications. You do not have to provision servers or pay for resources that go unused."

Routine tasks for provisioning, maintaining, and scaling server infrastructure, instead, are handled by a cloud service provider thus relegating developers to simply package code into containers for deployment.

After deployment, serverless apps respond to requests and automatically adapt to different scalability needs. The usage of serverless solutions offered by public cloud providers is typically measured on demand via an event-driven execution model, so serverless functions cost nothing when not in use.

In a serverless model, a cloud provider runs physical servers and dynamically allocates their resources on behalf of the user, who can deploy code directly to production.

Serverless computing products typically fall into two categories: Backend-as-a-Service (BaaS) and Function-as-a-Service (FaaS).

BaaS allows developers to outsource most aspects backend of a web or mobile application, so that they only have to write and manage business logic. Specifically, vendors usually provide services reachable remotely through simple APIs, which cover mechanisms such as database management, user registration and authentication systems, encryption, updates or push notifications. Being "*plug-and-play*", the integration of these services into a company systems is much simpler and, sometimes, more reliable than developing them internally.

BaaS services allow developers to tap into a variety of third-party services and applications. For example, a cloud provider can offer authentication services, additional encryption features, cloud-accessible databases, and highly reliable consumption data.

Instead, as far as concern FaaS and from the definition formulated by A. P. Rajan [7]:

"Serverless computing or Function-as-a-Service (FaaS) is defined as a software architecture where an application is decomposed into 'triggers' (events) and 'actions' (functions), and there is a platform that provides a seamless hosting and execution environment.", fundamentally emerge peculiar architecture components, such as the triggers, the invokers and the platform which breaks down whole application built on top of the FaaS architecture.

The *trigger*, usually consisting of an API Gateway backed by HTTP server, in the Function-as-a-Service model [8], is the component responsive to client external event solicitation.

If the trigger is implemented by an API Gateway, the client should call the HTTP endpoint to stimulate the trigger execution. Each endpoint corresponds to the activation of a function typically mapped from the HTTP request to a simplified format such as a JSON object, and such request can be easily used as the input parameter of the function.

Once the request has been processed, the function will execute its stateless logic via the usage of an additional component, the *invoker,* which prepares the proper execution environment, executes the user-defined function and return the result to the same API Gateway, which will turn this output back into an HTTP response to be forwarded to the caller. Additionally to this basic function, the API gateway can perform non-functional operations such as user authentication and input validation.

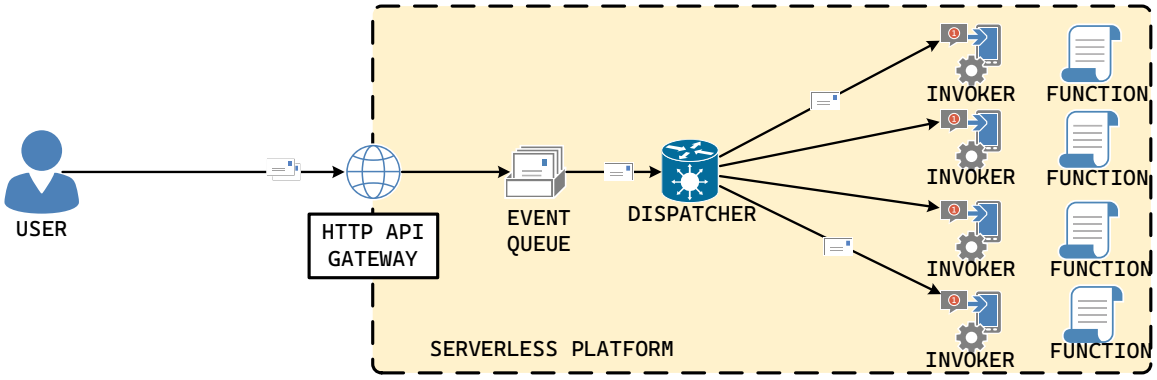

Figure 7 - HTTP API Gateway in a typical FaaS architecture

Even though FaaS architectures are suitable for asynchronous stateless applications and for usage scenarios that result in sudden and unpredictable spikes in demand, they lack a standard way how they manage some of the most design challenges as when facing distributed systems, such as:

- **Cold start:** The response time of a function, for which there is still no instance ready to respond, is one of the first limitations of serverless platforms and can range from a few milliseconds to several seconds.
- **Concurrency:** When choosing a vendor specific architecture, need to be considered what the number of requests and the total number of invocations processed per second will be; it is a good idea to check what are the degrees of competition, the limits imposed by the provider, and if these can be increased on request.
- **Autoscaling capabilities:** Not all platforms scale equally, as they take advantage of different automatic scaling mechanisms; using poorly optimized

autoscaling services can be a major bottleneck when the cluster receives a significant number of requests.

- **Runtime:** Almost not all languages are supported by a single provider, which may affect the selection of the platform to use.

Although research in the field of distributed systems is very active, especially in FaaS architectures, the resolution of the aforementioned problems is of actual interest and great scientific research. In fact, there is still no de facto solution that deals with the management and coordination, in the broad sense, of FaaS components.

The solution proposed in this thesis turns out to be, as a support platform, the introduction of an intermediate entity between the trigger and the invokers that plays the role of communication mediator called Message-Oriented Middleware (MOM) detailed in section 2.2 below.

2.2. Message Oriented Middleware

Message-Oriented Middleware (MOM) or Message Broker, [9] is a type middleware that enables the communication between different applications or systems by providing message passing for coordination among entities. The messages are typically sent asynchronously, so meaning that the sender and receiver do not need to be active at the same time in order to exchange messages.

MOM is typically used in distributed systems and in FaaS deployments where different components need to communicate with each other but may not be able to do so directly. For example, an application running on one server may need to send a message to an application running on another server, or a mobile device may need to send a message to a server. In these cases, MOM acts as an intermediary, allowing the different components to exchange messages without needing to know the specifics of how the other component is implemented.

They act as intermediaries between other applications, allowing senders to output messages without knowing where recipients are, whether they are active or not, or how many there are. This facilitates the decoupling of processes and services within systems.

MOM systems typically provide a variety of features such as message routing, message persistence, and message acknowledgement. Routing allows messages to be directed to specific recipients based on certain criteria, such as the content of the message or the sender identity. Persistence allows messages to be stored for later delivery if the recipient is not available at the time the message is sent. Acknowledgement allows the sender to know if the message has been successfully delivered.

MOM can be implemented in a variety of ways, such as using a message queue, a publish-subscribe model, or a request-response model. Each of these models has its own set of advantages and disadvantages, and the choice of which to use will depend on the specific requirements of the system.

These systems are widely used in enterprise environments, where they are used to integrate different systems and applications, and to build robust and scalable distributed systems.

Applications that exploit the power of a MOM create a distributed product that is compatible with various operating systems because messages are. In addition, MOM allows various software components to communicate or share data, synchronously or asynchronously with a store-and-forward capability [10], and in fact it is sometimes described as a link between front-end and back-end systems or as a software integration tool (Enterprise Application Integration).

Message brokers can address a wide range of business needs across industries and within different enterprise computing environments. They are useful whenever and wherever reliable inter-application communications and guaranteed message delivery are required.

To provide reliable message storage and guaranteed delivery, message brokers often rely on specific components, called message queue, that stores and sorts messages until the applications can process them.

A message-oriented architecture has a number of additional benefits in addition to requiring that the systems it is applied to operate in an actual and trustworthy synchronous or asynchronous mode:

- **Extensibility and adaptability:** The broker has the ability to handle numerous message types, manage multiple queues, and duplicate messages as

necessary. However, the most potent feature it offers system builders is its complete independence with respect to the consumer and producer processes, both of which may often be implemented in a wide range of programming languages. They can communicate with each other without any issues as long as they all adhere to the same protocol and message structure. There are many different types of producers, as well as customers. One queue of data can be consumed by many consumers in a manner similar to Round Robin, or they can perform various activities using the same input (i.e., all of them receive all the messages). This is a very useful feature since it enables, for example, real-time data replication to various data storage systems.

- **Scalability:** Any number of producers can fill any number of queues, and any number of consumers can either consume the data simultaneously or not. It makes horizontal scaling fairly simple because extra backend servers can be set up to transmit data to the broker as more HTTP requests come in. To enhance the rate at which communications are consumed by customers, the same thing can be done. The broker can also be clustered to do out replication and load balancing. However, the systems (such databases and file storage systems) where the producers and consumers would have to communicate data would also become overburdened. In order to truly give an architecture using a messaging middleware the ability to scale up completely, they should be able to scale as well.
- **Fault tolerancy and resiliency:** These architectures may be more resistant to hardware and software problems. To avoid any bugs that might cause the broker to fail before it could send the message, the broker can take care of replicating and storing the message once it has been pushed to it. The same is true for consumers: if they run into an issue, they can cut off their connection to the broker and simply cease processing messages until they are restarted or until another user steps in to finish the job. Consumption acknowledgements can be used on the consumer side to inform the broker when a message has been processed, allowing the broker to determine when to redeliver it in the event of a consumer failure.

- **Burst management:** On the Internet, traffic bursts frequently happen and may briefly overload some systems. Without altering the entire server code, the overall number of resources devoted to handling connections can be increased by relieving them of specific tasks that can wait. To serve as a "shock-absorber", message-oriented middleware is built to have a very high throughput capacity. The system builders have more latitude on the consumer side thanks to this "shock-absorbing" role because multiple consumers can be set up to consume the same set of messages (all receive all messages), though perhaps not at the same rate or even at the same time. By allowing sufficient time for messages to arrive before sending them to a database, it offers a fantastic approach to convert expensive one-shot processes (like database insertions) into batch actions. Additionally, even if one process adds data to a MongoDB instance while another adds it to a Hadoop cluster, for example, if one is slower than the other, they will still work as efficiently as possible without interfering with one another.
- **Process segregation:** Task separation is de facto enforced by message-oriented architectures. It is better to handle processes outside of the primary application server if they are not time-sensitive, important, or require replication. The activities are handled independently by focused tiny processes, while the broker serves as a scheduling and queuing mechanism.
- **Simplified approach for maintenance, administration, and delivery:** A message-oriented architecture is also simpler to administer in addition to providing process isolation and resilience. Tasks may be immediately diverted toward new consumers, which can be turned on and off without having an impact on the production application server. The broker works as a queue, continuing to receive and store messages from the producers while it waits for new consuming processes to be brought up, enabling the installation of new consumers while the old ones are being removed with no downtime.

A client of a MOM system can send messages to, and receive messages from, other clients of the messaging system. Each client connects to one or more servers that act as an intermediary in the sending and receiving of messages.

MOM platforms allow flexible cohesive systems to be created; a cohesive system is one that allows changes in one part of a system to occur without the need for changes in other parts of the system.

From the perspective of communication mechanism offered, two interaction models dominate message-oriented middleware environments:

- **Synchronous Communication:** The caller code must block and wait (halt processing) when a procedure, function, or method is called using the synchronous interaction paradigm. Once the called code has finished running and returned control, the caller code is free to resume processing. Systems rely on the return of control from the called systems while employing the synchronous interaction model because they lack processing control independence.

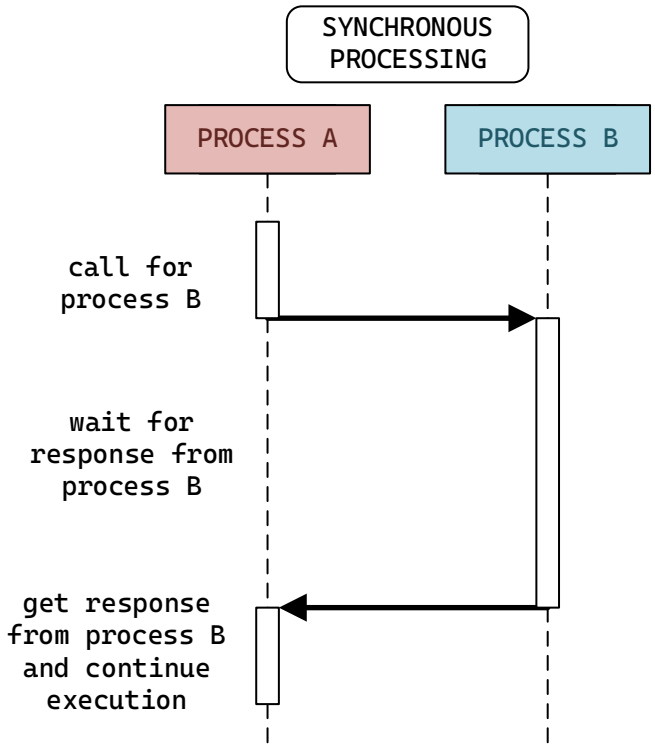

Figure 8 – Example of Synchronous Communication

- **Asynchronous Communication:** The caller can maintain processing control thanks to the asynchronous interaction mechanism. The calling code

does not have to block and await the returning code. This approach enables the caller to go on processing regardless of the called procedure, function , or method processing status. When there is asynchronous interaction, the called code might not run immediately. The exchange of requests must be handled by an intermediate in this interaction architecture, which is typically a message queue. The asynchronous paradigm allows for processing independence for all participants despite being more complicated than the synchronous model. Regardless of how the other participants are doing, participants can carry on processing.

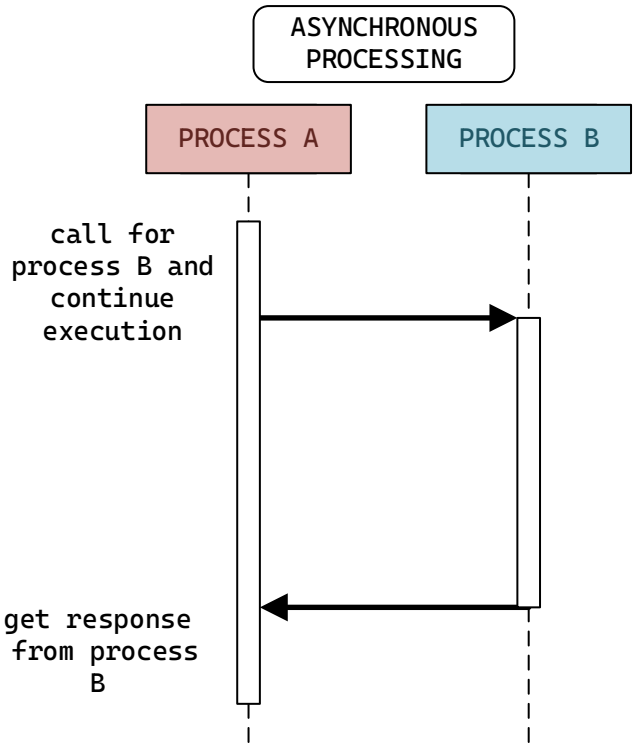

Figure 9 – Example of Asynchronous Communication

MOM-based distributed system deployments, and most importantly in FaaS deployments, offer a service-based approach to inter-process communication in an asynchronous fashion. MOM messaging is, in a sense, similar to the postal service. Messages are delivered to the post office; the postal service then takes responsibility for safe delivery of the message.

As a consequence of this, MOM decouple participants in a system granting the ability to link applications without having to adapt the source and target systems to each other, resulting in a highly cohesive, decoupled system deployment.

Furthermore, message loss through network or system failure is usually prevented by the adoption of a store and forward mechanism for message persistence. This capability of MOM introduces a high level of reliability into the distribution mechanism preventing loss of messages when parts of the system are unavailable or busy. The specific level-of-reliability is typically configurable, but MOM messaging systems can guarantee that a message will be delivered, and that it will be delivered to each intended recipient with different semantics.

These semantics may vary between different MOM implementation but as general reference they can be summarized as:

- **Exactly-Once:** For each message handed to the mechanism, that message is delivered once or not at all; in more casual terms it means that messages may be lost.
- **At-Least One:** For each message handed to the mechanism potentially multiple attempts are made at delivering it, such that at least one succeeds; again, in more casual terms this means that messages may be duplicated but not lost.
- **At-Most Once:** For each message handed to the mechanism exactly one delivery is made to the recipient; the message can neither be lost nor duplicated.

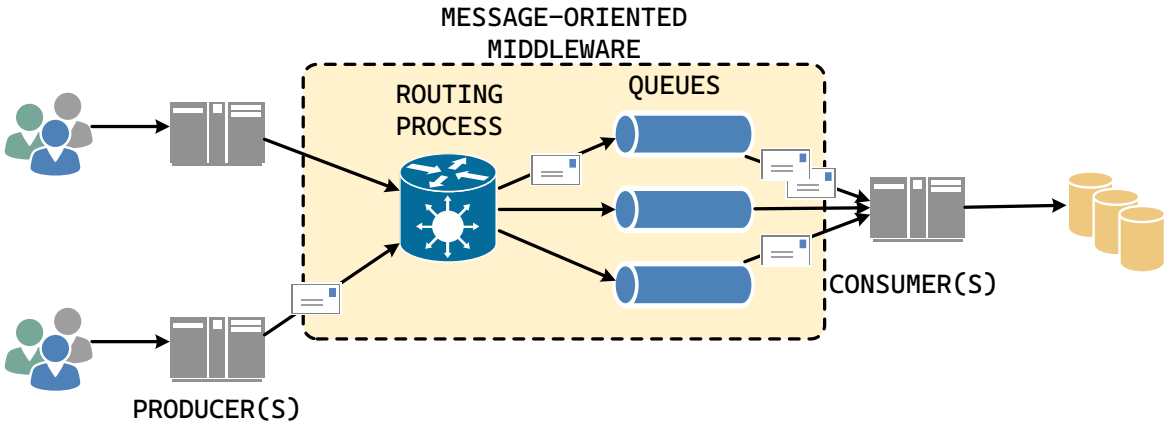

Figure 10 - Message Oriented Middleware High Level Architecture

At stated before, what emerges from Figure *10*, the message queue is a fundamental concept within MOM. Queues provide the ability to store messages on a MOM platform and represent the mean clients are able to send and receive messages.

Queues are central to the implementation of the asynchronous interaction model within MOM. A queue is a destination where messages may be sent to and received from; usually the messages contained within a queue are sorted in a particular order. The standard queue that can be found in a messaging system is the First-In First-Out queue; as the name suggests, the first message sent to the queue is the first message to be retrieved from the queue.

Typically, many attributes of a queue may be configured including the queue name, queue size, the save threshold of the queue, message-sorting algorithm, and so on.

Potentially each application interacting with the middleware may have its own or share a queue.

With respect to data exchange model, MOM specifically implements message delivery across software architecture scenarios in a point-to-point or publisher/subscriber messages dispatching models:

1. **Point-to-point messaging:** This is the message queue distribution model where there is a one-to-one correspondence between the message sender and recipient. Each message in the queue is only ever used once and is only ever forwarded to one recipient. When a message needs to be acted upon only once, point-to-point communications is appropriate. For example, Payroll processing and financial transaction are two examples of use cases that fit this messaging style. Both senders and recipients in these systems want the assurance that each payment will only be transmitted once.
2. **Publish/subscribe messaging:** The message producer publishes the messages to a topic in this "pub/sub" message distribution architecture, and several message consumers subscribe to the topics they want to receive messages from. All applications that have subscribed to a subject receive all

messages published in it. The link between the advertiser and their customers is one-to-many in a broadcast-style distribution strategy.

Now that we grasp why MOMs are considered crucial in FaaS deployments, the next step is about to understand why, despite the enormous financial and research efforts on such technologies they still suffer from some, we say still "insuperable", performances issue.

2.3. Approaches and Challenges in Function Composition

Writing functions, which are really just mappings from inputs to outputs, is the foundation of traditional programming. The programs are comprised of compositions using these function building blocks. One straightforward method for programming the cloud is to provide developers the ability to register functions in the cloud and then to construct programs out of those registered functions. This operation is known as function composition or chaining logic and foresees generally high delays and function propagation latencies. By decoupling complex problems into smaller ones, function chaining enables smarter management of complex tasks and processing pipeline capabilities. Even better, the possibility to compose functions together encourages their reusability, thus further reducing the development burden and hence the time to market. Unfortunately, current function chaining solutions exhibit some performance issues: response latencies can materialize, as stated in the introduction chaptyer, not only from a bad user-defined chaining logic but also from inefficient infrastructural support to function composition. Current FaaS architectures are indeed not optimized to handle bursts of short-lived functions, an inherent property of this increasingly popular approach, that can amplify the overhead in the function invocation path. In addition, no current production-ready - but in academic research [11] - platform adopts function co-locality optimizations that are otherwise widely employed in more traditional data processing platforms. This optimization does not only benefit single-host FaaS deployments but also multi-host scenarios where one knows in advance that two or more functions belonging to the same function chain can be co-located in the same host. In fact, this is not an

uncommon situation, and the multi-host FaaS scheduler can be tasked to handle the placement.

Before getting into what has been done up to now, it is important to consider the question of why serverless function composition is a relevant issue. The serverless functions concept is rather young and lacks adequate coordination mechanisms between functions. Currently, it is difficult and requires quite some effort to orchestrate a large set of serverless functions to create a complex application. Furthermore, serverless *"lags behind the state-of-the-art when it comes to function composition"* [12].

Serverless (and microservice) architectures are not arbitrarily derived from or created upon conventional application architectures since they use a less centralized and more distributed approach, hence requiring a cloud application architecture redesign.

Furthermore, serverless functions are not necessarily conceived with the idea of complex compositions in mind, however, is it widely spread that such complex compositions are envisioned in the form of workflows.

Amazon created their own workflow composition service that allows developers to chain AWS Lambda functions together to create more complex behaviours called AWS Step Functions [13]. This service is limited to the AWS platform and comes with a pricing scheme, but it demonstrates the recognized need for serverless composition solutions.

Due to this growing complexity and novel architecture composition approach, it is important to look at what composition models are suitable for serverless architectures and what are *"ways to express compositions of functions, and hybrid-cloud deployment"* even with the tools built so that could support the creation of compositions and their maintenance. [14].

FaaS architectures, mainly composed by ephemeral containers, are by design non-discoverable units that must be opened deliberately to the host machine network. In other words, we cannot explicitly address the container with an IP address or endpoint. Consider the potential security difficulties this might pose.

Thus, they offer handles for communicating with the function or entry points that can be triggered but lack direct network addressability.

This means that, if a developer has multiple functions that must be composed together to form a pipeline, rather than triggering each other internally and directly, the developer will have to hack around it by either triggering it via an HTTP endpoint if the provider permits it, or other external queueing systems they provide. In each of these instances, it is difficult to avoid additional latencies.

This makes FaaS particularly inefficient for distributed computing applications that rely on extremely fine-grained communication between functions.

In general, two general models for serverless composition patterns invocation can be identified [15]:

1. **Reflective Invocation:** In this kind of composition a third-party entity - the *Coordination Function*- encapsulate the logic of function chaining and it is in charge of invoking each function, waits for the result, forwarding it to the next function in the chain, until all the chain process is completed.

   To begin, we can think of reflective composition as a list of actions, which reflectively invokes each action in turn, moving the data from one stage of the pipeline to the next while depending on the presence of an external coordinator. This is just a first attempt at conceptualizing reflective composition. In this model, the controller, acts as an external scheduler in the sense that it is responsible for coordinating the timing of events and the flow of data. Since it makes use of an external reflective scheduler, the composition could make calling the underlying operations much more expensive. This is a trade-off for using the scheduler. As a matter of fact, the scheduler has to be kept active for at least the same amount of time that each of the sequenced operations needs to be kept active, as seen in Figure 11. There are economic repercussions that stem from a scheduler action being active simultaneously with its constituent parts. Every proactive activity depletes a common resource and, as a result, must have an associated cost. It would be preferable to steer clear of this overhead. This need is referred to as *the double billing constraint*, and as we will see in a moment, it is one of the issues that cannot be avoided when working with the function composition model.

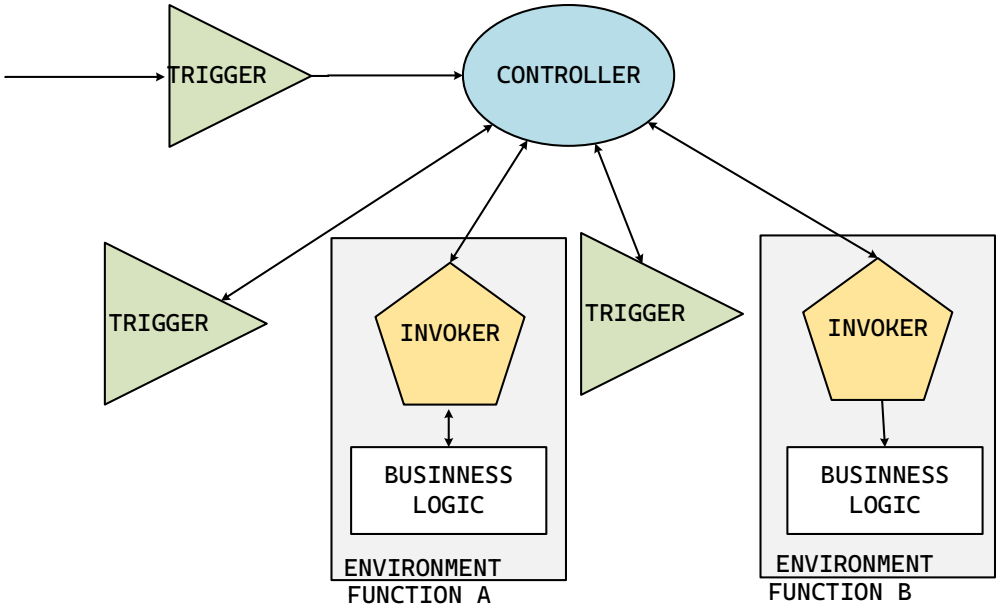

Figure 11 - Reflective Invocation

2. **Continuous Passing:** Composition pattern expects that invoked function is capable to locate, name and execute the next function piping the output straight into the input of the subsequent function.

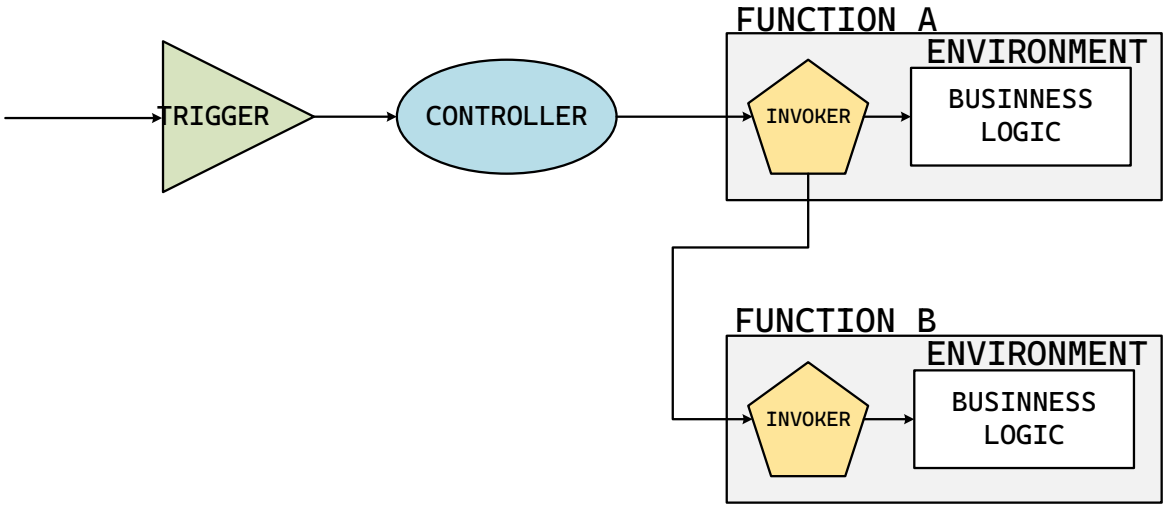

Figure 12 - Continuous Passing at Infrastructural Layer

Both patterns could be supported either at the business level (*Business Layer*) or as an infrastructure-level capability (*Infrastructural Layer*).

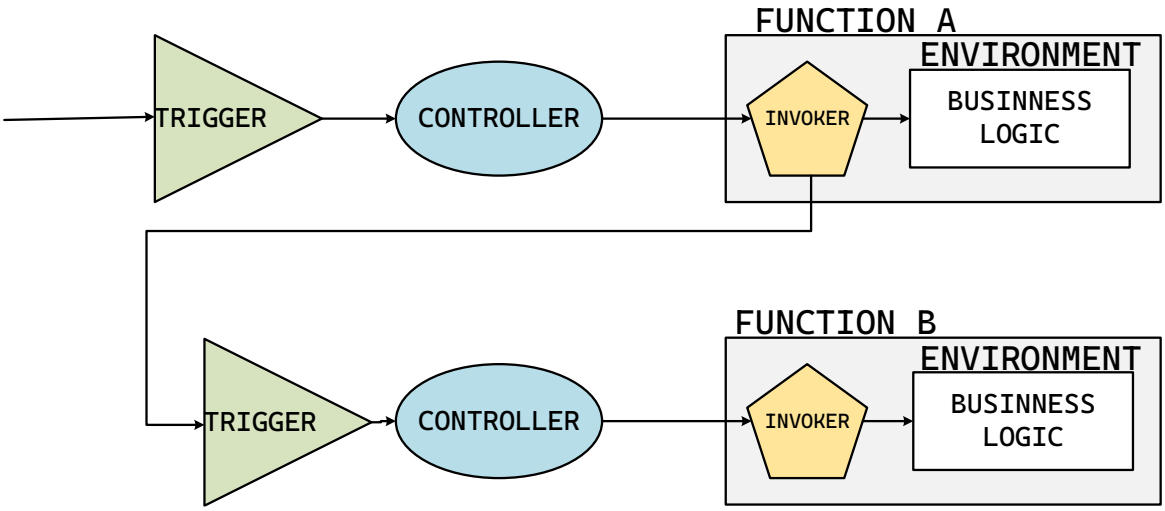

Figure 13 - Continuous Passing at Business Layer

At the *business level*, the infrastructure is based on an external application or service which will orchestrate several functions into a workflow. The service (e.g., AWS Step Function) exposes an API which when called will forward the request to the corresponding function which would be the next step in the workflow.

Functions must have at least one trigger that is externally available through the public API and may be executed by other. In this case the developer is responsible for developing the appropriate invocation logic and protocols for the composition and exchange of messages amongst functions in this instance.

The business logic is restricted to a predetermined composition strategy, which restricts the reusability and modularity of functions. However, this method gives higher expressiveness and dynamic possibilities.

Moreover, studies demonstrated an extension of the business level composition with the client-focussed composition [16]. This entails moving the composition from the cloud to the client application. For example, a smartphone application that handles the orchestration of many serverless functions itself instead of relying on a microservice or *AWS Step Functions*-like service handling the orchestration. Obviously, this moves the complexity from the cloud to the client, which might or might not be desired. Think about the security implications it has.

At the *infrastructure level, instead,* is up to the infrastructure the chaining logic of the workflow. What is obtained is that the business logic is totally decoupled from

the composition managed by the FaaS platform providing a neat separation among policy and mechanisms leading to better performance exploiting some optimizations such as function co-location, caching and optimized communication protocols.

From a high-level perspective, what the major literatures agree at is that function composition consists of at least three competing and inescapable constraints [12] depending on the function composition implemented:

- **Invocations could be double-billed:** When using reflective invocation composition, having a scheduler active at the same time as its components has economic ramifications. Any active action consumes the shared resource - the coordinator- and the resource associated with the function invocation could cause the *double-billing*, as shown in Figure 14;
- **Functions may not be treated as black box:** Trying to mitigate the double-billing problem inlining the functions code inside the external coordinator, will violate the black box principle because it would require reading the functions source code invalidating the principle. This is even worse when functions are written in different languages, because in this case inlining would be not supported at all.[2]
- **A composition of functions could not be a function:** This property does not hold in case of continuous passing composition because functions conversion with respect of asynchronous and synchronous function invocation. In fact, in order to avoid the double-billed issue it would be advisable to suspend the invoker such as it sends function invocations in a fire-and-forget fashion. This could lead a change of the invoked functions from a dictionary-based to a `Future<dictionary-based>` return type causing incompatibilities in their composition.

Understood the requirements, the FaaS architecture is appealing to have and seize on the constraints that would be acceptable in certain situations and on the ones that are not; that is the biggest challenge when dealing with distributed architectures as FaaS.

[2] Known as Polyglot Constraint

Despite the relative novelty of FaaS platforms, many solutions, both commercial and academic proofs-of-concepts, address the problem of function chaining. On the commercial front, Microsoft Azure Cloud has recently introduced Azure Durable Functions as an extension of Azure Functions. This extension enables the customer to define stateful workflows by writing special orchestration functions, whose state is managed by the cloud platform. Amazon with his AWS Step Function follows a different approach which involve the intervention of a particular external component, a finite state machine, controlling the execution of AWS Lambda functions composition. AWS Step Function allows to define a series of checkpoints in the chain, used to enable some mechanism of fault tolerance such as error handling and retry logic.

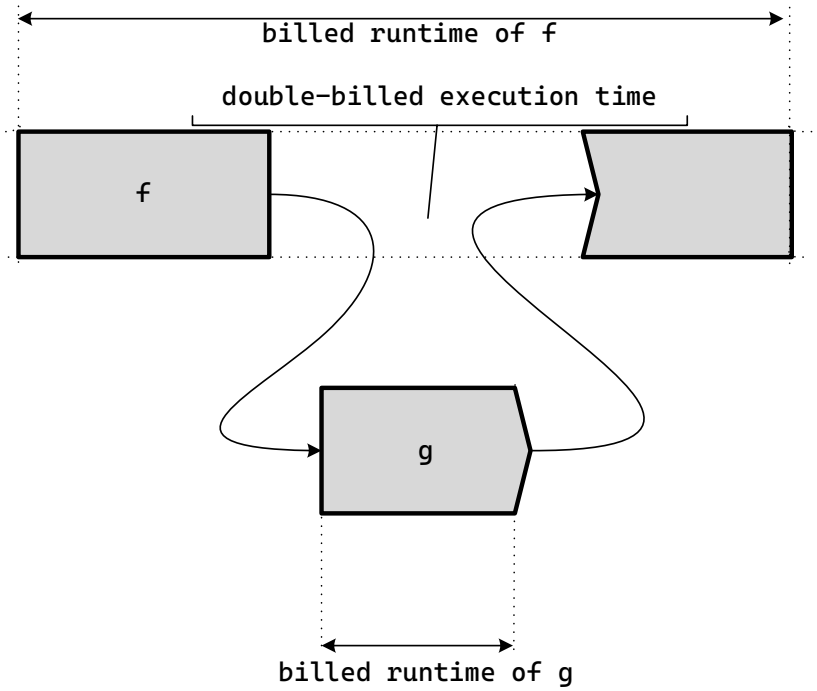

Figure 14 - The Double-Billing problem

A taxonomy [12] of potential function composition techniques in serverless systems, based on the formal characteristics they conform to, has been developed in the academic world.

All those solutions enable the execution of complicated processes on public clouds, but they also call for the creation of a third component that controls function triggering and event forwarding.

3. Thesis Project

As mentioned in the previous chapter, FaaS architectures suffer from efficiency problems with respect to communication between functions when the *controller* is involved in each function invocation in the *reflective invocation* composition architecture. The proposal made in this thesis is about in performance evaluation through some adopted optimizations when implementing function composition at architectural level through the usage of a MOM and it is about to enables the execution of advanced function composition without leveraging on any external component but only through the MOM capability as function composition supporting platform.

Furthermore, in this scenario what is going to be evaluated is the scalability of the system, in particular, whether it maintains such property through the usage of functions written in two different programming languages, JavaScript and Rust respectively, and in presence of multiple invokers.

To accomplish this, taking advantage of MOM-specific capabilities of internal information gathering and message interception, events are generated triggering the execution of the next function.

One of the most significant novelties introduced by analysing messages passing through the MOM and instrumenting the middleware is to generate events when a particular condition is met. This process can extend the number of operations executed natively at the infrastructural layer by the serverless platform such as the activation of a function every time a certain number of messages, for example, are passed through a specific topic.

In particular, two components, has been integrated inside the MOM and they are defined as *Queue Coordinator* and *Queue Monitor.* The *Queue Monitor* performs constant monitoring of the state of the overall MOM as well as of the individual queues. This monitoring includes tracking the number of clients subscribing, the number of messages that have been processed, and the number of messages that are waiting in a queue.

In terms of data and statistics that can be gathered via *Queue Coordinator,* the usage of interceptor has a broad capability in terms of kind of committed operation.

In fact, it is feasible to use an interceptor to not only inspect the status of the queue and the metadata associated to messages, but also the content of individual messages. On the other hand, since the messages processing occur synchronously in the broker, it is predictable that this will have some effect on the computing resources and performance of the MOM [17].

The interceptor, instead, is a specific component that subscribes to one or more queues of the MOM and collects statistics over the message received. The interceptor follows the general design dictated by the interceptor design pattern [18], a version of the Chain of Responsibility pattern from the Gang of Four (GoF) [19]. This pattern improves a system adaptability and extensibility allowing functionality to be readily added to the system in order to dynamically modify its behaviour. This seamless integration of functionality may be typically achieved without the need to halt and recompile the system, enabling its introduction during runtime.

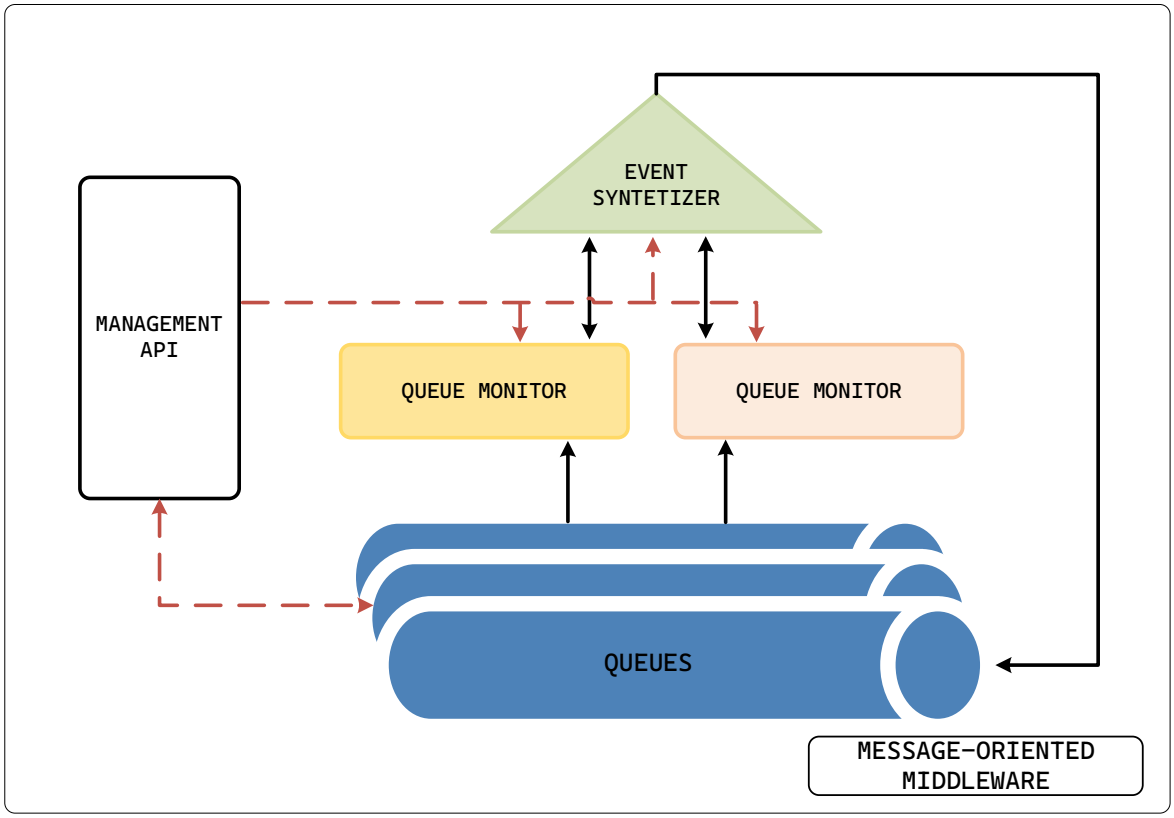

Figure 15 - MOM Centric Architecture

The metadata and information thus obtained by these two components are sent to the *Event Synthesizer*, which then produces events according to the process

specification set by the user. After that, events are directed into certain queues in accordance with the criteria that have been established in the workflow design, ultimately leading to the execution of a function being triggered by invokers subscribing to that particular topic. The reckoning is that, since the event source is in close proximity of the event generation workflow, it is expected to result in higher performance than the current state-of-the-art solutions, which are dependent on processes that are not part of the serverless architecture but external.

3.1. High Level Components and Interaction Schema Workflow

Herein, two setups will be shown as proof-of-concept designated to evaluate the effectiveness of the proposed solution. The first one, in Figure *16*, represents what is the current widely adopted solution in function composition, where external coordinators collect the data and generate events based of their internal state.

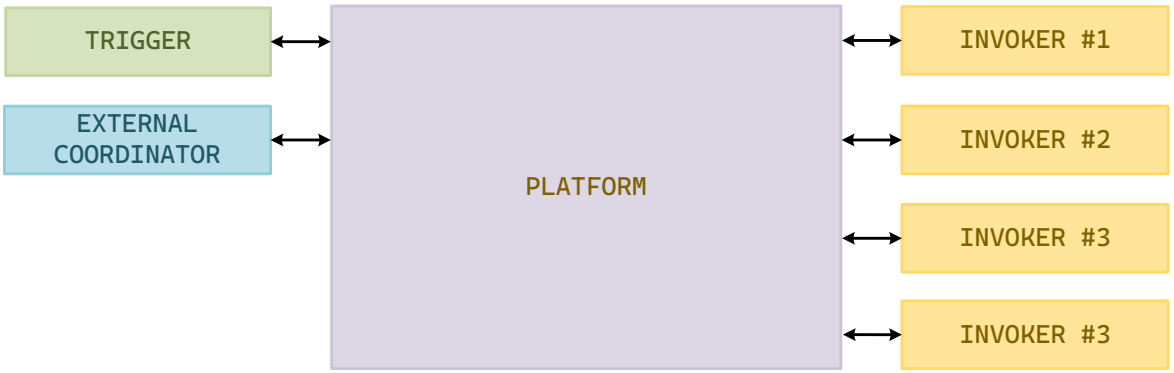

Figure 16 - Function composition through the usage of an External Coordinator component

The general interaction schema foresees that, upon user interaction via the *trigger* which receives external events from heterogeneous sources via potentially different protocols, converts them to local events for the FaaS platform. The events generated by the trigger are then managed by a *coordinator* that, based on configuration parameters provided by the user, forwards them to the proper queue in the MOM.

Function represents the business logic piece of code loaded in FaaS, which executes when specific events occur.

The *invoker*, instead, receive those events by mean of the middleware when particular and configured conditions are met, inject the business code deployed by the end-user and execute the business-logic function.

The function is always executed inside proper execution environment conceived containing all needed dependencies, system libraries and environmental variables.

All the computations performed by invokers, seen as a whole, follows the general structural workflow outlined by the *MapReduce* framework, so that each invoker belongs to either at the map phase or at the reduce phase.

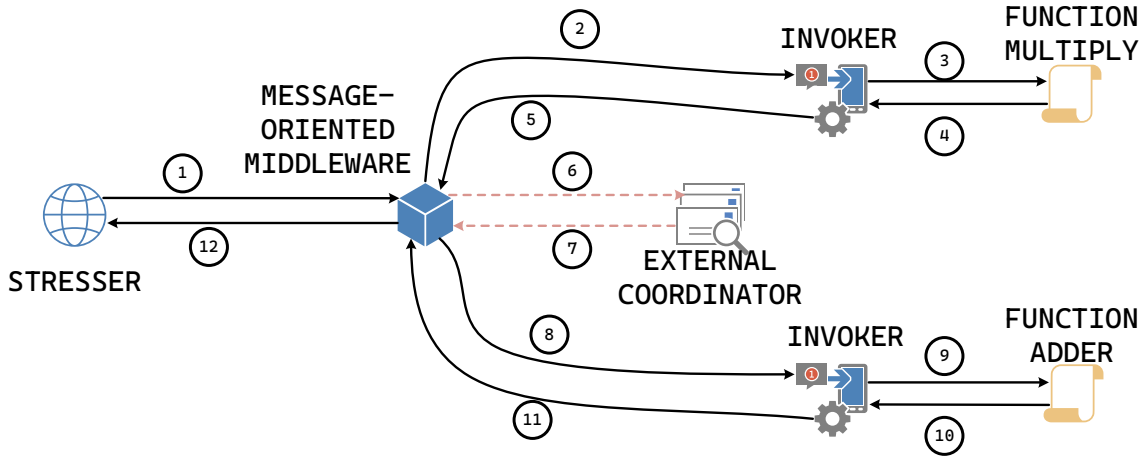

Figure 17 - Logical Interaction Schema. In contrast for what happens when the external coordination process is used, steps 6 and 7 are integrated inside the MOM as embedded component.

Starting from the very beginning, and in reference to Figure *17*, in the *input phase* (1), the stresser generates different traffic workloads toward the MOM, which will forward them the to the proper queue based on the topic destination labelled on the message header.

The invoker(s) subscribed at that topic will read (2) and extract the message from the queue, inspect it, and before it performs the function invocation (3) it sets the proper execution environment. When the synchronous function invocation ends (4), the invoker replies to the result back, publishing a brand-new message, to a preconfigured queue (5). With this operation, these invoker(s) conclude(s) the *map phase*.

Whenever a configurable number of messages is received in such queue (8), another invoker(s) read them in a whole, prepare the invocation environment, execute(s) the reducing function (9), and write(s) the invocation result back in a final queue (10-11). This concludes the *reducing phase*.

Between the map and the reducing phase, a *coordination phase* is performed (6-7). The configurable number of messages sent to the reducing phase depends upon the coordinator configuration. The role of the coordinator is, in fact, to fire the reducing phase whenever some condition, here implemented as a simple modulo counting, is met.

From an implementation point of view, the *QueueCoordinator, Queue Monitor* and the *EventSynthetizer* have been collapsed inside the *coordinator* component because, as infrastructural plugin, it demonstrates enough capability to produce relevant results during the stress test anyway.

Depending on the tested architecture, the coordination phase would happen inside or outside the broker.

In the latter architecture the *coordinator* is deployed outside the middleware, as illustrated in Figure 16 , conversely, primer one, the *coordinator* is completely embedded inside the MOM, as shown in Figure *18*.

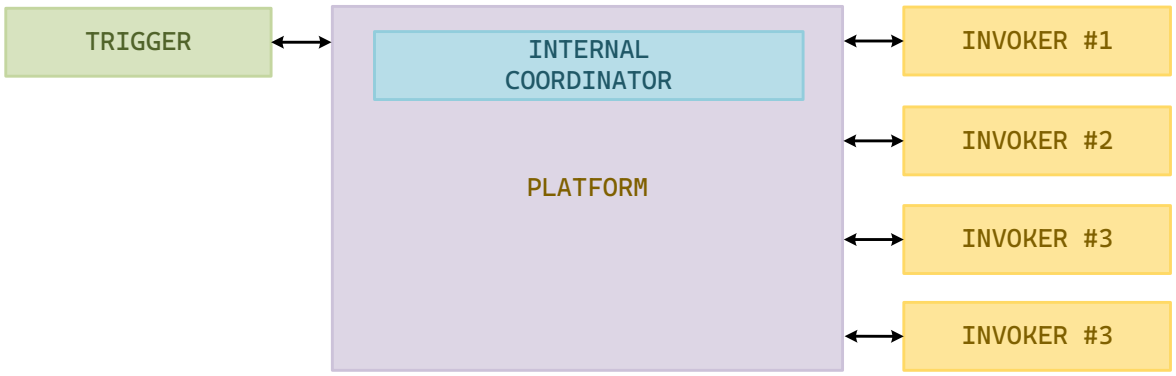

Figure 18 – Proposed function composition through the usage of an Internal Coordinator

Anticipating what will be thoroughly explained in paragraph 3.2.1 below, exploiting the internal broker interception capability, does not reply on external connections.

Since almost all components have been developed using Rust, it worth spends few words to highlight its amazing features.

Rust is a quite novel programming language, known for its memory safety features and blazing performance. It is a systems programming language developed by Mozilla Research designed to be fast, concurrent, and memory safe. Released in 2010, became a popular choice for developers building systems, web applications, and command-line tools.

Rust is known for its focus on safety and performance. The language was designed with a strong emphasis on avoiding common programming errors, such as null pointers (*null* not allowed at compile time), buffer overflows, and data races. This is achieved through Rust ownership model, which ensures that each piece of data is used in a predictable and safe manner.

The *borrow checker* [20] is an essential feature and is one of the parts that make Rust so unique. It forces the developer to manage ownership, i.e., it is a very efficient feature that helps to eliminate memory violation bugs, so problems are detected at compile time and rubbish collection is not necessary. It basically prevents value to be allocated as soon as the variable holding the value goes out of scope such as functions, associated functions, loop and inner scopes.

One of the unique features of Rust is its emphasis on zero-cost abstractions. This means that the abstractions provided by the language do not come with a performance overhead at runtime. This contrasts with other programming languages, where abstractions often come with a runtime cost (*virtual methods*). In Rust, abstractions are implemented in such a way that they can be optimized by the compiler, making it possible to write high-level code that performs as well as low-level code.

In addition to its focus on safety and performance, Rust has several other features that make it appealing to developers. One of these is its strong type system. Rust uses static typing, which means that type information is determined at compile-time, rather than at runtime. This helps to catch type-related errors early in the development process and makes it easier to maintain code over time.

Moreover, Rust boasts a great library supports and ecosystem porting from other programming languages, in fact, the interaction between client and the broker has

been made possible using *PahoMQTT* library, basically a Rust adapter of the well-known *PahoMQTT* [21] C library.

At its current version (0.12), *PahoMQTT* is compliant with the protocol v5.0, 3.1.1 and 3.1, supports TLS/SSL as transport protocol, Message Persistence, No-SQL adapter, High Availability, QoS 0,1,2 support and async/away operation.

Among all the features the library offers, the stresser is configured disregarding all of those that could cause either additional delays or mechanism that could influence the results.

As explained earlier, in order to support a minimum workable example, the requirement is that the proposal needs to set components up so that they follow the general processing archetypal MapReduce model.

A discussion of the mechanism and motives behind MapReduce will be provided before moving on to the implementation sections.

3.1.1. MapReduce Model

The MapReduce is a programming model, firstly introduced by Google in 2004 for processing large datasets in parallel across a cluster of computers.

This model consists of two user-defined phases, the *Map Phase* and the *Reduce Phase*, and an implementation-specific numbers of minor phases. In a nutshell, the map phase takes a large dataset and divides it into smaller chunks, which are then processed in parallel by multiple nodes in the cluster. The reduce phase takes the results of the map phase and aggregates the data into a single result.

The map phase takes a dataset and a map function as input. The map function takes a single item from the dataset as input and produces a set of key-value pairs as output. The map function is applied to each item in the dataset in parallel, producing a set of intermediate key-value pairs.

The reduce phase takes the intermediate key-value pairs produced by the map phase and subsequent transformation phase and aggregates the data into a single result. The reduce phase takes a set of intermediate key-value pairs and a reduce function as input. The reduce function takes a key and a set of values as input and produces a single output value for that key. The reduce function is applied to each

unique key in the intermediate key-value pairs, producing a set of final key-value pairs.

Digging deeply into the MapReduce architecture [22], it consists mainly of the following phases:

1. **Input Splits:** In this phase, the input is split into smaller chunks of data so that each of them becomes the elementary piece of data the mapper deals with.
2. **Mapping:** In the Mapping phase, the input data is analysed and separated into smaller segments, with the number of mappers equal to the number of input splits. The input splits are then subsequently transformed into key-value pair so that each mapper applies coding logic to these key-value pairs and creating an output with the same structure.
3. **Shuffling:** By eliminating duplicate values and arranging them, the shuffle phase prepares the output of the mapper phase for transmission to the reduction phase. In the mapper phase, the output is usually still presented as keys and values pairs.
4. **Sorting:** Shuffling and sorting happen concurrently. The output produced by the mapper is merged and sorted during the Sorting phase. Before beginning the reduction phase, the intermediate key-value pairs are sorted by key, and the values may be in any order.
5. **Reducing:** In the Reducing phase, the intermediate values from the shuffling phase are reduced to produce a single output value that summarizes the entire dataset.

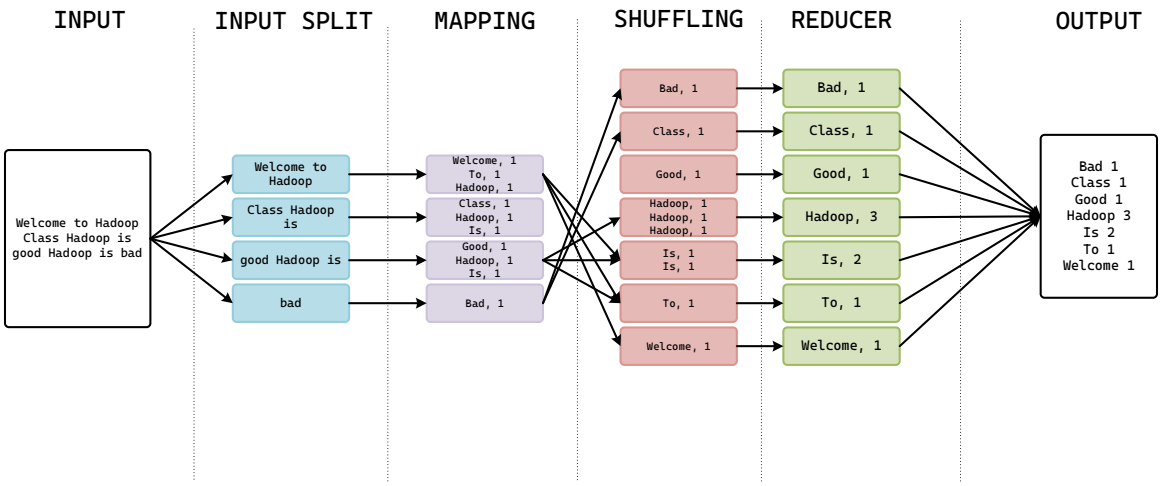

Figure 19 - MapReduce Phases

In Figure 19, is represented the idiomatic architecture of the MapReduce model previously described whilst, in order to give a concrete idea how it works, an elementary but typical example of a MapReduce application is word count.

Given a large text dataset, the goal is to count the number of occurrences of each word in the dataset. In the map phase, the map function takes each line of text as input and produces a set of intermediate key-value pairs, where each key is a word, and each value is a count of 1. In the reduce phase, the reduce function takes each word as a key and aggregates the count of occurrences for each word, producing a final count of occurrences for each word.

The *MapReduce* model provides several benefits for processing large datasets, taking advantage of the processing power of multiple nodes in a cluster. Secondly, it provides a way to distribute data processing across a cluster, reducing the amount of data that needs to be processed by each node.

3.1.2. Coordinator Implementations

The *coordinator*, in its MOM-embedded plugin version, need to exploit the internal middleware APIs. It, in fact, it overrides the `void afterSend(…)` method that will be fired as soon as a client publish a message. As shown in the Listing 1, upon a message retrieval, it is being filtered by the broker by its destination topic.

If the number of intercepted messages (that messages which match the filtering criteria) is equal to the configured value, statically coded as modulo 5 operation, the coordinator publishes a message on another queue where the reducer is listening to.

This simple logic has been implemented as shown:

Listing 1 – Internal Coordinator Plugin in Java

ThesisCoordinator.java

```
@Override
// Method overridden from the ActiveMQServerMessagePlugin
public void afterSend(ServerSession session, Transaction tx,Message
            message, boolean direct,boolean noAutoCreateQueue,
            RoutingStatus result) throws ActiveMQException {

//1. If the message has been correctly routed, hence, the message
can eventually published, then process it
  switch (result) {
    case OK:

//2a. If the destination queue is “prod_topic_done_int”, then it is
the message we are interested in
      if (message instanceof CoreMessage
          && message.getAddress().equals("prod_topic_done_int")
      ) {
        CoreMessage coreMessage = (CoreMessage) message;

//3. Unparse the message data buffer as JSON
        JSONParser jsonParser = new JSONParser();
        ActiveMQBuffer activeMQBuffer = coreMessage
        .getDataBuffer();

        ByteBuf buffer = activeMQBuffer.byteBuf();
        String messageJSONString = buffer
        .toString(Charset.defaultCharset());

//4. If the unparsing operation has been completed, then increment
the successful arrival messages.
        ThesisInterceptor.counter += 1;
//5. If the number of messages received is modulo 5, prepare for
event triggering into dedicated MOM queue
```

```
          if (counter % CONFIG_N_TRIGGER == 0) {
            try {
              JSONObjectmsg = null;
              while (
              !ThesisInterceptor.receivedJsonMessage.isEmpty()
              ) {
                msg = ThesisInterceptor.receivedJsonMessage.poll();
//6. Push the backed up multiplied number to the data structure
                numbersList.add(
                    (Double) ((JSONObject) msg
                              .get(CONFIG_FIELD_DATA))
                              .get(CONFIG_FIELD_DATA_NUMBER));
              }
              ThesisInterceptor.receivedJsonMessage.clear();

//7. Push last received message multiplied number to temp data
structure
              numbersList.add(
                  (Double) ((JSONObject) currentObject
                  .get(CONFIG_FIELD_DATA))
                  .get(CONFIG_FIELD_DATA_NUMBER));
//8. Create the event pushing all the multiplied number inside this
message
              JSONObject numbersObject = new JSONObject();
              numbersObject
              .put(CONFIG_FIELD_DATA_NUMBERS, numbersList);
              currentObject.put(CONFIG_FIELD_DATA, numbersObject);
//9. Prepare new message.
//Omissis
//10. Perform the event publishing on the MOM queue
              session.doSend(tx, msgResult, null, direct,
              noAutoCreateQueue);
//11. Reset messages received
            ThesisInterceptor.counter = 0;

          } else {
//2b. Back this message up till the number of received messages is
modulo 5.
            ThesisInterceptor.receivedJsonMessage
            .add(currentObject);
          }}
```

At the same time, in order to mimic the same behaviour but as external component, the following listing shows the coordinator implemented in Rust:

Listing 2 – External Coordinator in Rust

coordinator.rs

```
//1a. If the number of messages received is modulo 5, prepare for
event triggering into dedicated MOM queue
if counter % 5 == 0 {

//2. Reset messages received
  counter = 0;
//3.Message parsing
//Omissis
//4 Push last message multiplied number to temp data structure
  numbers.push(number);

  while !message_temporary_store.is_empty() {
      let recovered_message = message_temporary_store
      .pop_front().unwrap();

//5. Push the other backed up multiplied number to the data
structure
      numbers.push(recovered_message.number());
  }

  message_temporary_store.clear();

//6. Create the event pushing all the multiplied number inside this
message
  let data = MultipleInteger::new(numbers);
  let msg = Message::new(group, creation_timestamp, &dst_topic, 0,
  flow, data);

//7. Perform the event publishing on the MOM queue
  let msg: MQTTMessage = msg.into();
  if let Err(e) = cli.publish(msg) {
      error!("error while publishing message. {}", e);
  }

} else {
//1b. Back this message up till the number of receive messages are
modulo 5.
  message_temporary_store.push_back(m);
}
```

Both implementations are tasked to read incoming messages from a message queue and inspect their content. As far as concern the external coordinator, the

preconfigured message queue from which messages are read is the *prod_ topic_ done,* so before being able to receive any message it has to subscribe to that queue.

The embedded coordinator instead, since it does not rely on preliminary queue subscription, since it directly intercepts the messages, it filters them using the destination queue header field and discards all the messages not sent toward the *prod_ topic_ done_ int* queue.

Conversely, since both coordinators needs to publish in the same queue, where the reducing invoker is subscribed, they could potentially write on the *sum_ topic* queue firing the reducing event for the further processing of the data.

Besides that, there are no other major differences among the internal and external implementations apart from the fact that the latter need the *PahoMQTT* Rust library to establish a connection with the broker.

3.1.3. Invokers Implementations

The last but not the least important implemented architectural components were the invokers. Even though different invokers were adopted, one for the map phase and the other for the reduce phase, their implementation were rather easy because they accomplished pretty the same tasks.

As stated at the beginning of this chapter, the role of the invoker is to retrieve the message from the queue, read the invocation arguments from the message, the function name as long as the proper environment -e.g., the interpreter and environment variables- and execute the corresponding function. Upon function completion, it collects the results and publish them on the configured topic.

Since those invokers belong to the infrastructure layer but are distributed entities, thus not integrated into the MOM, they were completely developed in Rust.

The only valuable part in Listing *3* is that invoker calls the proper function as an external process yielding the result back via command piping. The part related to the command piping and the code related to the reducing phase, hence the summa, has been omitted for brevity.

Listing 3 - Rust Invoker

invoker.rs

```
//1. Multiply or Summation depending on the type of operation
Mode::Multiply => {
//2. Retrieve message from the underlying Paho MQTT Buffer
 for m in &receiver {
    match m {
       Some(m) => {
//3. Incoming message parsing
//Omissis

//4. Execute corresponding operation with the set programming
language (none, rust, javascript)
        let multiplied_number = match &command {
//5. Executes multiplication via mock function
         None => unexpensive_multiply_function(number),
         Some(i) =>
          match i {
           Interpreter::JavaScript => {
            match
//6. Executes multiplication via NodeJs
            execute_command(
              Interpreter::JavaScript,"function_prod.js",&[number]
             )
            {
//Omissis
            }
           }
           Interpreter::Rust => {
            match
//7. Executes multiplication via Rust executable
       execute_command(Interpreter::Rust,"function_prod",&[number])
            {
//Omissis
            }}}};
         let data = SingleInteger::new(multiplied_number);
         match dst_topic {
          Some(ref d) => {
//8. Construct the reply message toward the MOM
```

```
with the operation result
          let msg = Message::new(
           group, creation_timestamp, d, 0, flow, data);

          let msg: MQTTMessage = msg.into();
//9. Publish the newly created message to the MOM
          if let Err(e) = cli.publish(msg) {
           warn!(
            "thread-{}] error publishing message. {}",job,e);
          }
```

At this point, as long as the internal coordinator is integrated inside the broker, preliminary research has been conducted to address the right MOM implementation to fulfil the duty.

What will be shown, in the next section, is that MOM implementations can differ from many characteristics, varying from the architectural design choices taken, the support offered, the financial aspects among free and paid solutions, and the set of features they have.

3.2. ActiveMQ

There are many and many message-oriented middleware systems that have entered the market in the past recent years, so providing a wide range of features. The broad range of products available means it is tough to decide which messaging system is the most suitable for the specific use case. Even though is desirable message-oriented middleware having the most promising performance, measured as higher incoming and outgoing processed messages per second, it is not the only feature to take into account when MOM needs to be integrated into an architecture.

The following table summarizes principal features current middleware market proposals have:

Broker / Feature	ActiveMQ	ActiveMQ Artemis	HiveMQ [23]	JoramMQ [24]	Mosquitto [25]	RabbitMQ [26]	VerneMQ [27]
Open source	Apache 2.0	Apache 2.0	Commercial	LGPL, Commercial	EPL/EDL	MPL 1.1	Apache 2.0
MQTT version	3.x	3.x, 5.0	3.x, 5.0	3.x	3.1.1, 5.0	3.1.1	3.x, 5.0
Retain flag	YES	YES	YES	YES	YES	Partial	YES
Last will and testament	YES	YES	YES	YES	YES	YES	YES
Persistent Session	YES	YES	YES	YES	YES	YES	YES
MQTT Message Interceptor	YES	YES	YES	NO	NO	Partial	YES
QoS Level 0	YES	YES	YES	YES	YES	YES	YES
QoS Level 1	YES	YES	YES	YES	YES	YES	YES
QoS Level 2	YES	YES	YES	YES	YES	NO	YES
Bridging	Partial	YES	YES	YES	YES	NO	YES
Clustering	YES	YES	YES	YES	NO	YES	YES
REST Management API	YES	YES	YES	YES	NO	YES	YES
Management CLI	YES	YES	YES	YES	NO	YES	YES

Table 1 - Message Oriented Middlewares Comparison Chart.

Among all the previous alternatives, the choice fell on ActiveMQ Artemis mainly because it is open source, it has a very large community around the project and boast a nice documentation. Furthermore, it supports the most up to date MQTT (v.5.0) version thus giving the newest set of added functionalities [28] as long as all the QoS 0,1,2 levels even though not used in this work.

The most promising feature ActiveMQ Artemis has, is the native support for message interception, that will be explained in a while.

For its appealing features, ActiveMQ Artemis is one of the most popular open-source, multi-protocol, Java-based message broker which supports industry standard protocols so that users get the benefits of client choices across a broad range of languages and platforms.

It is capable to connects clients written in Rust, JavaScript, C, C++, Python, .Net supporting messages exchange via AMQP, STOMP or MQTT protocols.

ActiveMQ Artemis core is designed simply as set of Plain Old Java Objects (POJOs) with as few dependencies on external jars as possible. In fact, ActiveMQ Artemis core has only one jar dependency, *netty.jar*, other than the standard JDK classes because it uses some of the netty asynchronous communication classes internally.

Netty is an NIO client server framework which enables development of network applications simplifying and streamlines network programming such as TCP and UDP socket server.

Each ActiveMQ Artemis server is shipped with its own ultra-high performance persistent journal, which it uses for message and other persistence allowing the best persistence message performance, something not achievable when using a relational database for persistence.

Additionally, clients, potentially on different physical machines, interact with the broker adopting two APIs for messaging:

- **Core client API:** This is a simple intuitive Java API that allows the full set of messaging functionality without some of the complexities of JMS
- **JMS client API:** The standard JMS API is available at the client side.

If used, JMS semantics are implemented by a JMS facade layer (thin yellow layer between the *Core Client* component and the *User Application*) on the client side because, even though ActiveMQ Artemis does not speak JMS, it is designed to be used with multiple different protocols [29].

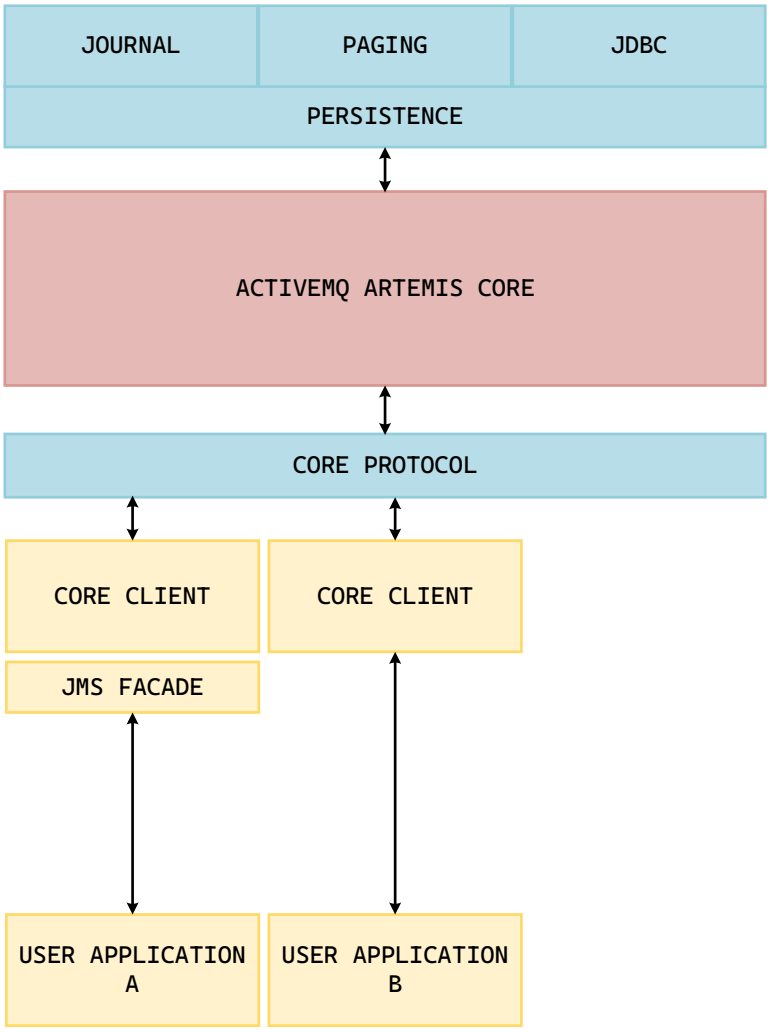

Figure 20 - ActiveMQ Artemis Architecture

When a user uses the JMS API on the client side, all JMS interactions are thus translated into operations on the ActiveMQ Artemis Core API before being transferred over the wire using the ActiveMQ Artemis wire format.

Considering all the above, at the time of this work, ActiveMQ Artemis 2.27.0 has been chosen as message-oriented middleware to act as message conveyor and supporting the coordination among all the architecture components.

3.2.1. Broker Interception Capabilities

Moving ahead for what concern the scope of this work, ActiveMQ Artemis offers several tools capable of intercepting messages, namely *interceptors, divertors and*

plugins. We will now explore their specificity, advantages, and disadvantage of those components and, eventually, identify the one which fits our need.

Divertors have been immediately discarded because, even though could intercept messages on queue name basis, their aim is to provide a transparent mechanism to forward message routed to one address to some other address. Anyway, divertors may be useful because empower the MOM with some degree of scalability, because messages could be transparently routed to queues belonging to different brokers allocated in a different machine.

In fact, when combined with *bridges* [29] can be used to create interesting and complex routings scenario. The set of diverts on a server can be thought of as a type of routing table for messages. Combining diverts and bridges permits the creation of a distributed network of trustworthy routing links between numerous geographically dispersed servers, hence enabling the creation of a global messaging mesh.

Interceptors, conversely, allow intercepting incoming and outgoing message arriving or leaving the broker, they inspect packets entering and leaving the server and, even if interceptors are meant to run on server, they could be also exploited and configured to run client-side thus distributing the intercepting operation directly on the clients.

Each packet that enters or leaves the server causes the incoming and outgoing interceptors to be called, correspondingly. This enables the execution of custom code, for instance, for packet inspection, filtering, or other purposes. The packets interceptors can alter the content of the message granting a range of possibility and flexibility for what concern message data manipulation but, likewise, potentially risky.

Depending on which protocol the message relies on, the interceptor should be chosen accordingly, and these interfaces are at developers disposal in order to cover all the protocols supported by the broker.

Once one of these interfaces has been implemented, the `broker.xml`, the broker main configuration file, needs to be updated with one of the following sections matching the direction of the intercepted message, inbound or outgoing accordingly:

Listing 4 - Interceptor Configuration

```
<configuration>
<!--add this section for intercepting incoming messages-->
<remoting-incoming-interceptors>
      <class-name>
            org.apache.activemq.artemis.jms.example
            .LoginInterceptor
      </class-name>
      <class-name>
            org.apache.activemq.artemis.jms.example
            .AdditionalPropertyInterceptor
      </class-name>
</remoting-incoming-interceptors>
<!--add this section for intercepting outgoing messages-->
<remoting-outgoing-interceptors>
      <class-name>
            org.apache.activemq.artemis.jms.example.
            LogoutInterceptor
      </class-name>
      <class-name>
            org.apache.activemq.artemis.jms.example
            .AdditionalPropertyInterceptor
      </class-name>
</remoting-outgoing-interceptors>
</configuration>
```

Since interceptors are protocol dependant, their Java implementation provides a clear and straightforward interface for message payload manipulation. This would have made interceptor the best choice for the experimentations.

Unfortunately, the method `boolean intercept(…);` does not provide any mechanism to publish messages other than the one the message is intended for.

The only way to achieve this purpose is to rely on a client-side JMS compliant capable to open a local but networked connection. Networked connections would have stacked another layer of abstraction, thus adding more unwanted delays. For this reason, interceptor have been classified as not suitable for the scope of this thesis.

At last, *plugins*, follow similarly set up as interceptor do. They require developer to implement `ActiveMQServerPlugin` interface, override the appropriate methods, put the `*.jar` archive inside the `artemis/<instance_name>/lib` folder and instruct the broker to be aware of it adding the following tag inside the `broker.xml` configuration file:

Listing 5 - Plugin Configuration

```
<configuration>
   <broker-plugins>
      <broker-plugin class-name="it.thesis.interceptors.Coordinator">
      </broker-plugin>
   </broker-plugins>
</configuration>
```

`ActiveMQServerPlugin` interface, shown in Listing *6*, is none other than a collection of interfaces that, in turn, have a collection of methods invoked as callback when some event occurs.

Listing 6 - ActiveMQServer Plugin Interface

```
package org.apache.activemq.artemis.core.server.plugin;

public interface ActiveMQServerPlugin extends
      ActiveMQServerBasePlugin,
      ActiveMQServerConnectionPlugin,
      ActiveMQServerSessionPlugin,
      ActiveMQServerConsumerPlugin,
      ActiveMQServerAddressPlugin,
      ActiveMQServerQueuePlugin,
      ActiveMQServerBindingPlugin,
      ActiveMQServerMessagePlugin,
      ActiveMQServerBridgePlugin,
      ActiveMQServerCriticalPlugin,
      ActiveMQServerFederationPlugin,
      ActiveMQServerResourcePlugin {}
```

The `ActiveMQServerMessagePlugin`, in fact, provides a broad set of methods hooked to different broker-related events inherited from a super set of interfaces.

Among all those interfaces the one which allows the interception of the messages is the `ActiveMQServerMessagePlugin`, endowing the developer a series of methods as described below:

- `void beforeSend(…)`: Invoked when a message is about to be sent to the broker;
- `void afterSend(…)`: This method is executed whenever a message is sent toward its destination queue, but before it has reached the broker;
- `void onSendException(…)`: When a sending operation fails, this method provide a convenient callback to handle this situation.
- `void beforeMessageRoute(…)`: Whenever a message reaches the broker a routing decision should be taken to point out at the correct forwarding queue, if existing. This method allows to add some logic before this decision is made.
- `void afterMessageRoute(…)`: Conversely from the predecessor, this method is fired after the routing has been performed regardless of whether it is successful or not;
- `void onMessageRouteException(…)`: Sometimes, routing could fail due to different reasons, and when this happens, this method is called;
- `void beforeDeliver(…)`: This method is executed after the routing is successfully performed, before the message leaves the broker this met;
- `void afterDeliver(…)`:Whenever the message leaves the broker, that is no acknowledgement should be awaited from the receiver, this method is executed without taking into consideration the message delivery.

Recalling the main reason that led us to consider interceptors inadequate, was the lack of a message publishing mechanism integrated within the Artemis API. The methods of the `ActiveMQServerMessagePlugin` plugin, on the other hand, allow the previous interaction through the `ServerSession` object passed as argument.

The `SeverSession` object, in Artemis, represents a JMS-compliant session, that is a single-threaded context for producing and consuming messages leveraging direct Artemis API internal interaction, flaunting the best achievable performance.

Eventually, this let us consider *plugin* convenient as *internal coordination* component, capable of intercept all messages passing through the broker, fulfilling all the coordination tasks and to publish messages into supposed topic.

Topics are used as designated locations in which the publisher wishes to post messages and where subscribers may receive certain messages. A string that has one or more subject levels separated by a forward slash (/), from left to right, serves as the topic identifier. As one descends the tree, the hierarchy will become lower. indicating that analysis at the first subject level is given at first.

Topic naming may be used as a wildcard to subscribe to many subjects or to filter among several topics names depending on the demands of the client.

Multi-level and single-level wildcards are two separate categories of wildcards [28]:

The plus sign (+) may be used to substitute a single-level in the whole topic name (e.g., "`sensors/+/temperature`" - subscribers can get a message from the client that broadcasts temperatures of all rooms).

A multi-level wildcard may be used with the hash symbol (#) to replace numerous subject levels. The last character in the subject should be the multi-level wildcard. As an example, "`sensors/room/#`" subscribers may obtain a message from the client that publishes all measurement of the room meaning getting all messages of subjects that start with the pattern before the wildcard character.

When the broker gets the first published message on the given subject, the topic will be automatically created if not explicitly set on configuration indicating that subject was not already created by the broker. This feature makes it possible for clients to communicate with one another even if message queues are not created thus permitting a more scalable environment.

As discussed in this chapter, ActiveMQ Artemis supports several mechanisms for message interception. *Plugin* implementation, however, provides a more specific set of features for the intended goal than all the other alternatives explored, allowing for full interaction with different aspects of the MOM.

In the next chapter, the two types of deployment adopted and how the different components of them interact will be illustrated. Furthermore, given the need to collect metrics that have the purpose of summarizing the results, the workflow for generating them will be shown in detail.

4. Experimental Setup

Before going through all the results, this chapter introduces and illustrates how the overall architectures and deployment were conducted. Specifically, there will be shown how multi-cloud and single-cloud deployments were simulated and where, on each of them all the components took place. Subsequently to this, in order to introspectively step into the workflow timings will be show how data have been collected.

4.1. Deployment

As anticipated in the first chapter, adopting multi-cloud strategies can guarantee significant advantages. Despite this, one of the downsides when using these types of deployments are the response times which, in models such as FaaS, are particularly important and therefore determine, in the decision-making phase, which FaaS implementation will be chosen.

Given these circumstances, the present thesis work aims to highlight improvements in response times when FaaS architectures are deployed even in multi-cloud environments.

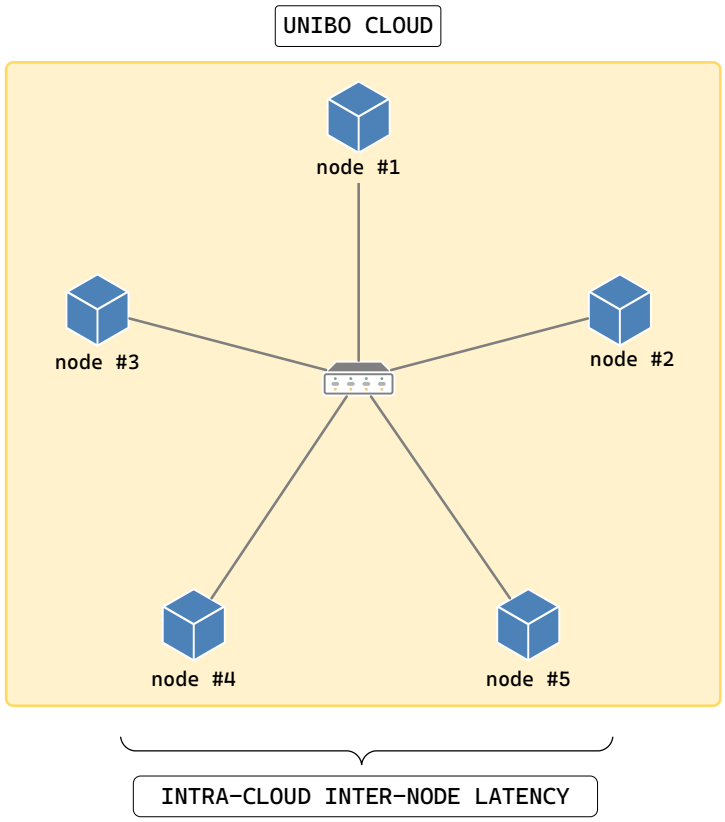

Figure 21 - Single-cloud physical schema. The virtual nodes instantiated are connected through virtual links which can sustain high traffic volumes with negligible delays, order of magnitude lower than physical links.

Intuitively, expectations are to obtain concretely different results in a multi-cloud environment, where response times are generally higher caused by the continuous round-trip time of exchanged messages among geographically distributed nodes.

Even if single cloud-deployments may consist in significant latencies [30] when dealing with distant zones, the intra-cloud communications can leverage on dedicated and trusted backbone links, routing optimizations or even SLAs [31], which could keep latencies generally lower than in multi-cloud deployments.

Single-cloud deployments were simulated, such as that illustrated in Figure *21*, exploiting private UniBo datacenter were one-way intra-cloud inter-node latency were about 500µs.

In multi-cloud scenarios, certain optimizations implemented at the architectural level are almost non-existent due to the different technical nature, but even organizational and business-related ones.

As shown in Figure *22*, three out of five nodes were deployed on the UniBo datacenter, while the remaining two nodes run on the Microsoft Azure cloud.

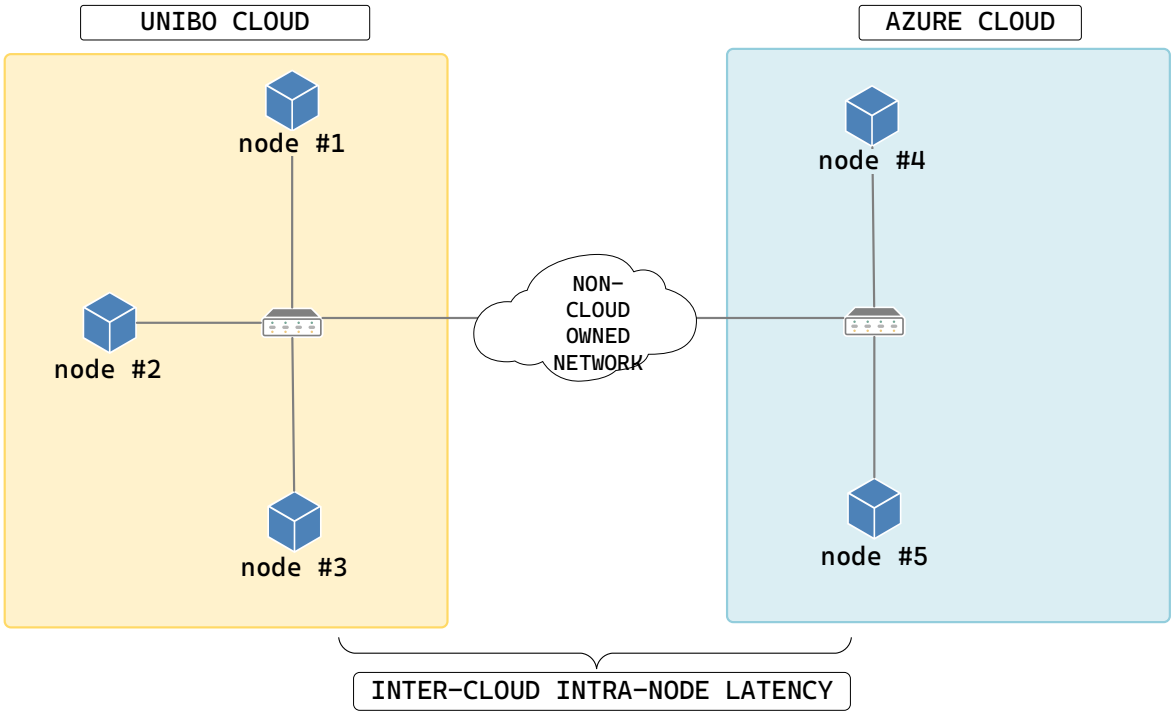

Figure 22 – Multi-cloud physical deployment schema. The illustrated infrastructures foresee the adoption of two cloud environments, Microsoft Azure and Unibo Datacenter respectively. Communications among these deployments rely on the general public internet connection without the adoption neither of leased nor proprietary links.

Experimentations were conducted by means of at least five nodes which, in the two proposed scenarios, they differ only in their location depending on whether a single-cloud or multi-cloud deployment is being considered.

As a result, when in a multi-cloud scenario, resulting end-to-end inter-cloud inter-nodes latency of 35ms $\pm$ 3ms.

Each node is instantiated as a virtual machine with two virtual processors and two cores each, 8 GB of RAM, a virtual network card with 1Gb/sec of maximum bandwidth. running Ubuntu 20.04.

Independently from the actual deployment used, **node #4** runs the Artemis ActiveMQ MOM, which listens on port 1883 to accept incoming connections from the MQTT clients.

In two other nodes, namely **node #5** and **node #3,** run the invokers involved during the mapping phase and the reducing phase respectively. Additional invokers instances will be spawned accordingly to the test performed, with a total of 8 invoker nodes, 4 for the multiplication function and 4 for the summation function invocations.

On **node #1** runs the traffic process generator, namely the *stresser*, which is being used in place of the trigger to simulate different load patterns. Interactions will therefore be directed towards the MOM, hence, not mediated by a *trigger* as in typical FaaS architectures. Although this could be seen as a limitation, it actually allows to better isolate what is undergo verification. Adding more components would have made the tests less faithful by introducing unwanted latencies and distortions.

In the last node, so **node #2**, run the external coordinator that is the component that performs the coordination logic among the map and reduce phases. The internal coordinator executes the same machine where the MOM is instantiated because it is instantiated as ActiveMQ plugin.

4.2. Metrics Collection

To collect the experimental data generated during the test assessment, the architecture must be capable of gathering metrics, i.e., data and information used to evaluate the effectiveness of the suggested solution and the system capacity to adopt a strategy via the use of MOM capabilities.

The delay between transmitting and receiving an event is of significant relevance and is known as the end-to-end latency. Nonetheless, with end-to-end latency collection, numerous metrics, such as mean, minimum, maximum, and the standard deviation from that RTT, may be determined.

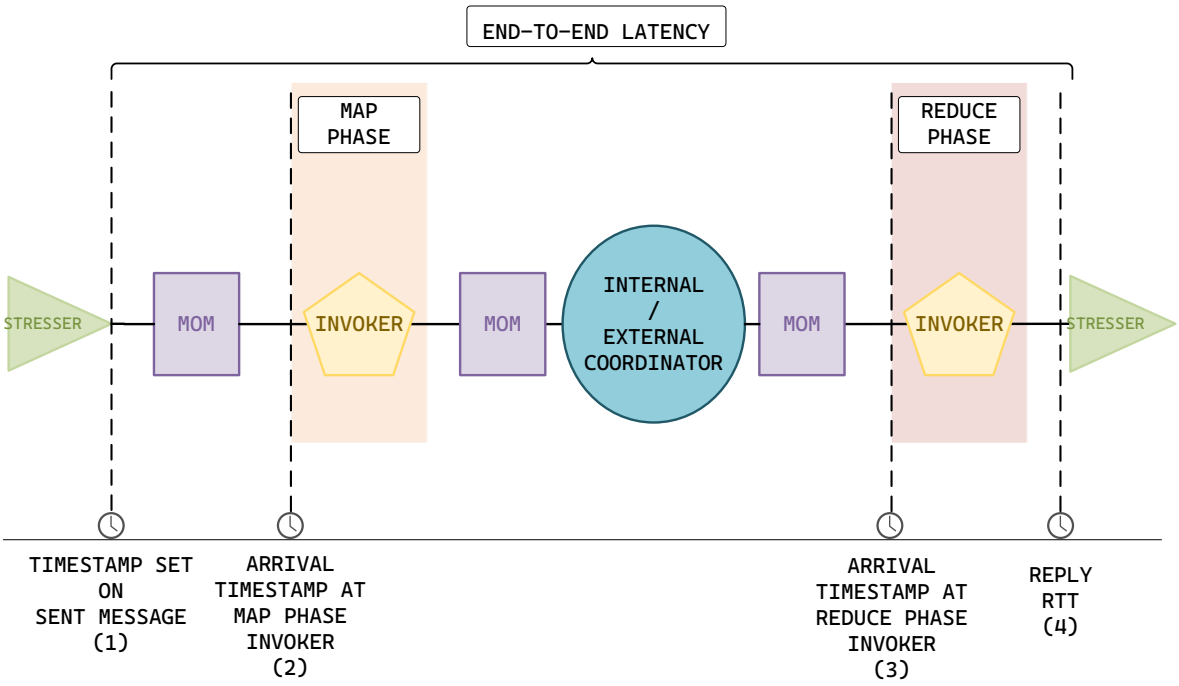

Figure 23 - Metrics calculation workflow. The actual physical node placement may vary based on the deployment used, but rather the Round-Trip time is calculated according to this schema.

End-to-end latencies generated by the internal coordination mechanism are predicted to be lower than those produced by the external coordinator, particularly when tests are run in multi-cloud deployments rather than single-cloud one.

As a result, a monitoring service based on timestamping activities has been added by design at crucial points of the message flow.

To collect such timestamps, as shown in Figure *23*, when the *stresser* is about to send message toward the MOM, it appends a timestamp representing the instant in which the message leaves the stresser (1).

As long as the message is being extracted from the MOM queue by an invoker, the one either at the mapping or at reducing phase (2-3), it is in charge of calculate

the arrival delay by subtract the message timestamp from its local-generated one. In this way, each invoker is responsible to generate its own delay perspective that, taken as a whole, will allow us to build the complete system view decomposition.

Recalling that the MapReduce model is meant to produce result by data aggregation, is normal to expect that some of the message initially sent by the stresser will eventually be collapsed into one due to the mapping phase. To figure out which message timestamp need to be considered to calculate the RTT at the end of the workflow (5), it has been considered the subtraction among the received message timestamp and the message timestamp of the lowest message ID part of the same received message group. For group, it is denoted a set of messages whose IDs is included in the same range, defined as:

$$5 * n \leq ID_x \leq 5 * (n + 1), \qquad \forall n \in N$$

Equation 1 - Message Group Definition

This is feasible since IDs and timestamps are monotonically generated by the stresser, so that the subtraction always succeeds.

The problem now is to decide which message will be sent back from the reducing phase. The simplest adopted solution is to record the highest IDs and, thus, its latest timestamp which will build the reply message up fields when the number of messages received modulo 5 equals to 0.

With this set up put in place, our implementation is capable to have a simple metric monitoring framework to record sufficient activities log entries used to produce the results shown in the next chapter.

5. Experimentations and Relevant Results

This section presents the experimentations carried out for the evaluation as a whole, varying the types of architectures proposed, the load in terms of messages sent by the *stresser*, as well as the evaluation through the processing capabilities comparison either of internal or external coordination techniques, identifying possible advantages but even possible drawbacks of the proposed solutions.

As anticipated in Chapter 3, the broker configuration involves instantiating non-durable queues so as not to add additional delays to the message flow and because delivery constraints are considered not valuable for the sake of this work. For the same purpose, client configuration foresees the sending of QoS 0 message only. In fact, with such setting turned on QoS 1 or QoS 2, what could be experienced at some time during tests execution, is the message swapping to disk even though RAM were not completely full. This would lead to great loss in performance, but most importantly, in test results spoiling.

Tests were performed by creating unique usage scenarios combining four execution variables:

1. **Model of deployment used:** The use of single-cloud and multi-cloud configurations will allow us to evaluate the eventual benefits which comes from two scenarios that differ, substantially, in terms of deployments extensiveness.
2. **Active number of listening invokers:** What we want to test, in fact, is not only a possible performance improvement (a certainly desirable feature), but also the scalability of the infrastructure as the interacting components grows. For what concern components replication, if the invoker executes the multiplication function and it is firstly deployed on the UniBo datacenter, whenever multiple invokers assessment test is going to be performed, the invoker replication sticks on the same CSPs where the first replica was deployed; same applies to invoker executing the addition function. For this purpose, three configurations will be used, respectively named 1x, 2x and 4x:

a. **1x:** Involves the usage of a mapper and a reducer, so the resulting workflow is comparable to a sequential execution.
b. **2x:** Involves the usage of two mappers in parallel and two reducers in parallel.
c. **4x:** Involves the usage of four mappers in parallel and four reducers in parallel.

3. **Representative workload:** Specifically, by means of the *stresser*, three types of loads will be generated depending on the scenario to be evaluated, as follows:
   a. **Stream of incoming requests emitted at steady regime:** This type of load is intended to evaluate the properties of our serverless platform with a constant regime of incoming requests. This scenario is particularly valuable for assessing the responsiveness of the system in the stability condition, i.e., what is normally expected to be the average response times during usual workloads.
   b. **Stream of incoming requests emitted at increasing rate:** This scenario mimics a typical traffic pattern that occurs when a large number of events need to be processed at once.
4. **Programming languages used:** Since serverless architectures are as agnostic as possible with respect to the programming language used, we opted for two function invocation written in as many different programming languages, and another case without any function invocation.

   In the primer, the languages have been chosen taking into consideration their average execution speed [32], so that could emerge different results consideration:
   a. **JavaScript:** High-level language and interpreted, has been selected as a representative of that class of languages that need an additional infrastructure to support their execution.
   b. **Rust:** Once compiled it is capable of achieve higher performance than the primer classes considered.

By this end, we use three different types of businness-logic invocation criteria: embedded in the invoker code as a Rust mock function, also referred as *no function call*, with a negligible execution time, an external function through the invocation of a Rust executable with execution times of about 2700µs ± 240µs and lastly through the invocation of a JavaScript function executed via the usage of the *nodejs* interpreter with an execution time of about 360ms ± 12ms [33].

For each sent message, in addition to message payload application data(not relevant for the test), the stresser adds the sending timestamp. This is because to obtain a Round-Trip Time estimation as precise as possible it is calculated as the difference between the message returning timestamp and the one saved inside the payload, hence a subtraction between local timers.

The choice to include the timestamp within the message payload is enforced by the need to calculate the system decomposition as in Section 4.2 above.

In this case, differently how worked previously, latency is calculated by measuring the difference between the arrival timestamp (for example in the map phase) and the timestamp inserted within the message payload. However, since these two timestamps belong to different nodes, some synchronization mechanism needed to be adopted to reduce the time skewness as much as possible. For this scope timers have been synchronized using the NTP [34] protocol, through which, as in Figure *24*, it was possible to obtain timing offsets in the order of hundreds of µs, considered acceptable for the overall results.

```
ntp_adjtime() returns code 0 (OK)
  modes 0x0 (),
  offset 177.321 us, frequency -20.360 ppm, interval 1 s,
  maximum error 245942 us, estimated error 734 us,
  status 0x2001 (PLL,NANO),
  time constant 6, precision 0.001 us, tolerance 500 ppm,
```

Figure 24 - NTP estimate time synchronization

Besides that, in all these scenarios, the composition length is kept constant to 2, representing a common option in real-world use case with just one map and one reduce operations.

The following results will be organized as follows: for each workload iteration, starting with a constant message emission rate and then as a stream of incoming requests at increasing rate, will be compared the results obtained from the single-cloud and multi cloud deployment. Each iteration will be also repeated for each invoker function execution configuration.

Additionally, the parallelization level of the invokers ranges from one (1x), thus a sequential execution, two (2x) and four (4x), as parallel executions. The parallelization level is meant to evaluate the scalability of the infrastructure at increasing number of participating entities while keeping constant all the other variables.

Furthermore, at the end of each test evaluation section, will be evaluated the scaling capability of the proposed solution taking into consideration the CPU resources utilization as well as memory consumption expressed as memory allocation. Since the platform resource consumption could be pulled out only under the assumption of heavy loads, this last assessment is performed using the 4x configuration with no function invocation.

Each test producibility is obtained consistent using the same input context condition data, computational steps, and conditions of analysis cleaning caching data saved inside `artemis/<instance_name>/data/` folder by ActiveMQ Artemis broker.

5.1. Stream of Incoming Requests Emitted at Steady Regime

In this first experiment, we would like to investigate the system behaviour under a steady regime, hence, the *stresser* issues a number of requests at a constant rate per second simulating a typical FaaS load scenario.

Note that in case of excessive results gap among data series, to let clearer readings, the y-axis has been set in logarithmic scale.

Each chart could be divided into 3 sections, grouped by same scaled colour, that, starting from the left, deal with sequential, parallelization of 2x and 4x executions respectively. Furthermore, each pair of charts are meant to compare tests run, but in single-cloud and multi cloud scenarios, keeping all the other execution variables fixed hopefully possible observations could be advanced.

All tests presented in this section were carried out for a fixed time limit of 300 seconds.

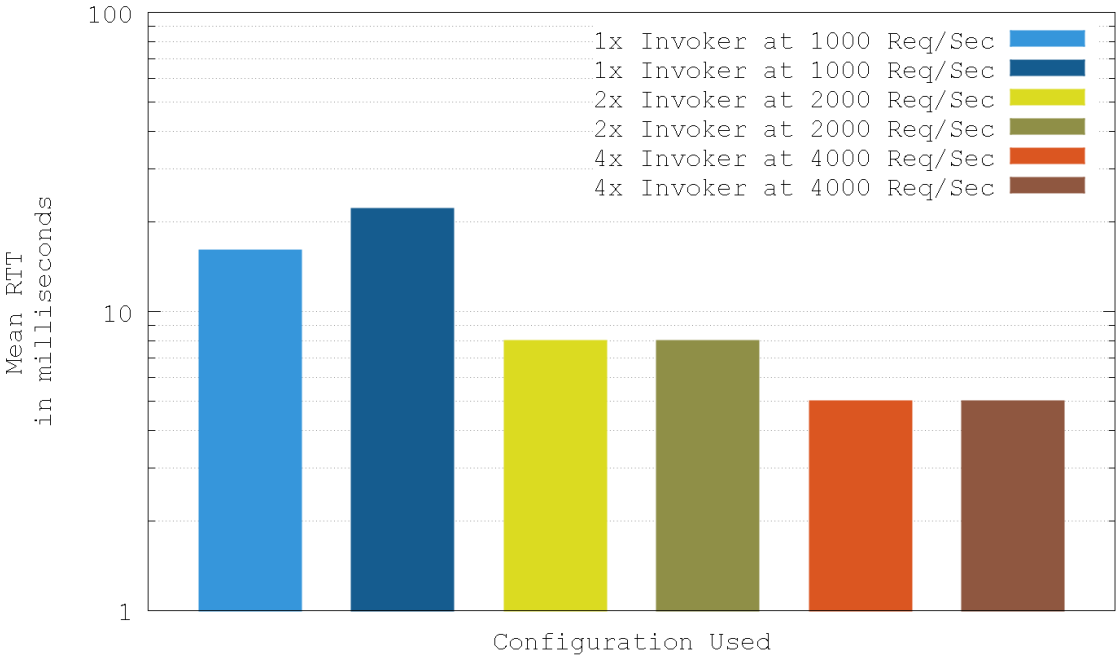

Figure 25 -Bar chart representing average end-to-end latency with constant message flow in presence of different kind of interceptors, light colour for internal and dark colour for external, at increasing level of invokers parallelism without function call in single-cloud scenario.

Figure *25* illustrates the relationship between different throughput at the maximum admissible and sustainable constant rate when no function invocation occurs in single-cloud scenario.

The tests are carried out with the constant issuing rate of 1000, 2000 and 4000 requests per second as far as concern the parallelism considered of 1x, 2x, and 4x respectively.

With parallel invoker executions the total average does not gain any relevant performance gain in which remain rather constant at 8ms and 5ms when 2x and 4x respectively. However, when it comes with sequential execution, the situation changes dramatically with a net improvement of 38% going from 22ms down to 16ms round-trip time. This behaviour is attributable to the fact that when intra-nodes delays are comparable with that related to the time spent executing the whole processing pipeline, including coordination tasks and subsequent communication overhead, benefits become more evident. Augmenting the infrastructure parallelization, however, flows processing execution time drops to a level only dependant to the communication overhead, hence, since single-cloud deployment consists of negligible inter-node delays, our solution does not provide any relevant results.

Overall, from a scalability point of view, our solution, in the worst-case scenario, is capable to deliver the same performance as well as the solution with the external coordination technique.

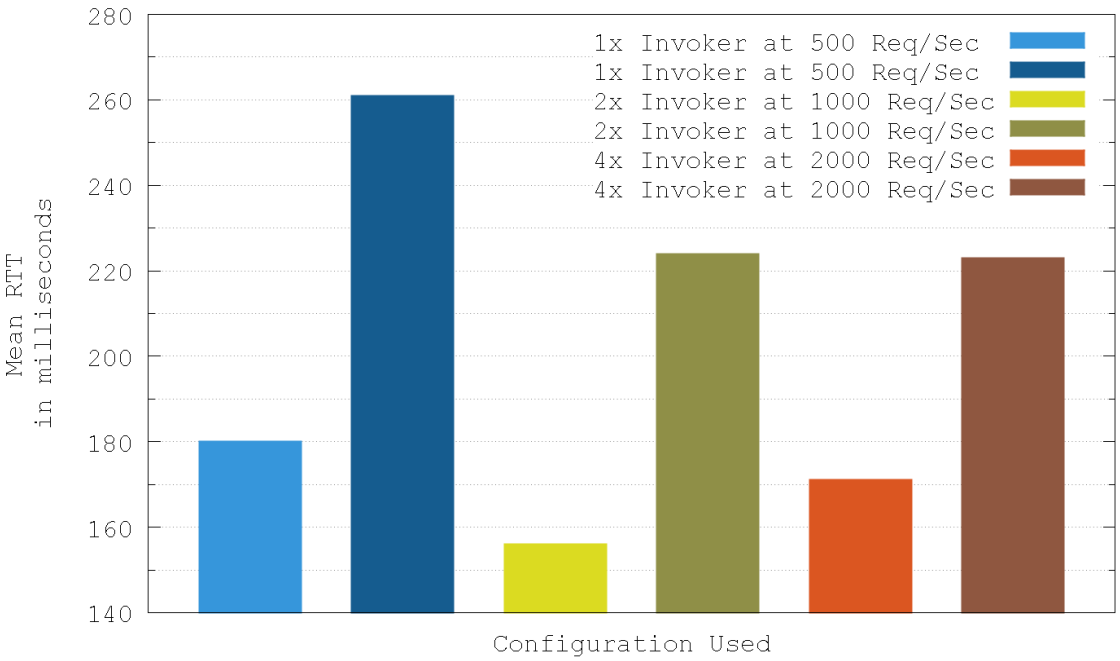

Figure 26 - Bar chart representing average end-to-end latency with constant message flow in presence of different kind of interceptors, light for internal and dark for external, at increasing level of invokers parallelism without function call in multi-cloud scenario.

In Figure 26, tests repetition consists in maintaining the same set up but considering a multi-cloud deployment scenario. What immediately emerges is that using the internal coordination component allows great timing improvement especially when in multi-invokers cases. Differently for what happens in a single-cloud deployment where timings keep under the tenth of millisecond, and thus adopting an embedded coordination logic provide some substantial changes just in sequential execution model, instead, benchmarking a distributed architecture leads to more incisive results especially when in a parallelized architecture.

As mentioned in the previous test, the average timings, when in a 1x setup, we got a 38% improvement, whilst in the latter these result keeps rather similar with an overall benefit in terms of 36% cut delays going from an average RTTs of 213ms down to 137ms.

Conversely, 2x and 4x configurations, timings, drop from 172ms to 125ms and 166ms to 122ms respectively with an average 28% reduction, whereas there were no timings improvement in the single-homed scenario.

Even in this case, our solution linearly scales as well as the solution adopted, i.e., with the usage of external coordinator, hence does not point out any performance drawbacks.

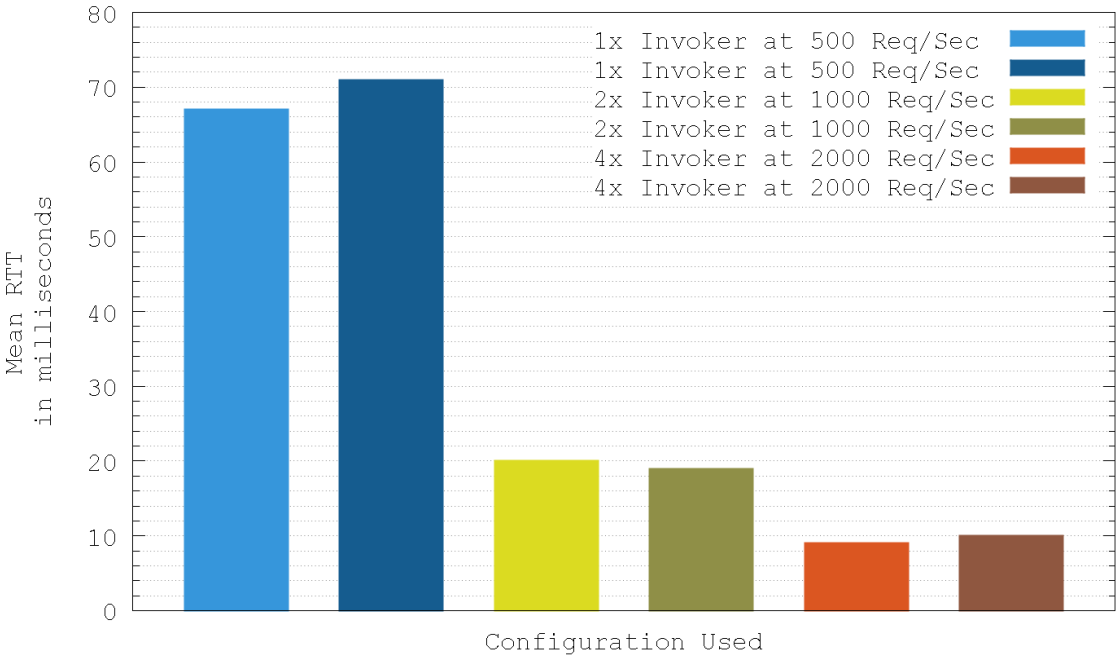

Figure 27 – Bar chart representing average end-to-end latency with constant message flow in presence of different kind of interceptor, light for internal and dark for external, at increasing level of invokers parallelism with Rust function execution in single-cloud scenario.

Figure *27* summarizes test results when invokers, this time, run a Rust function as executable.

When tests are performed with the invocation of the Rust executable, the sending rate, however, have been adjusted accordingly to avoid message enqueuing caused by unprocessed message backlog. This led subsequent speed reduction at 500 requests per second for sequential processing (1x), 1000 requests per second with a parallelism of 2x and, ultimately, 2000 requests per second on the 4x case.

For the same reasons, apart from overall higher execution time, the proposed solution shines when internal coordinator is used in sequential, single-cloud deployment with a humble, but somehow significative, 8% delay reduction, whilst the situation remains more or less unaltered as far as concern performances in multi-

invoker scenarios. When the system is subjected to a 2x or 4x invoker parallelization the time discrepancies remain too little to be considered valuable from the performance point of view.

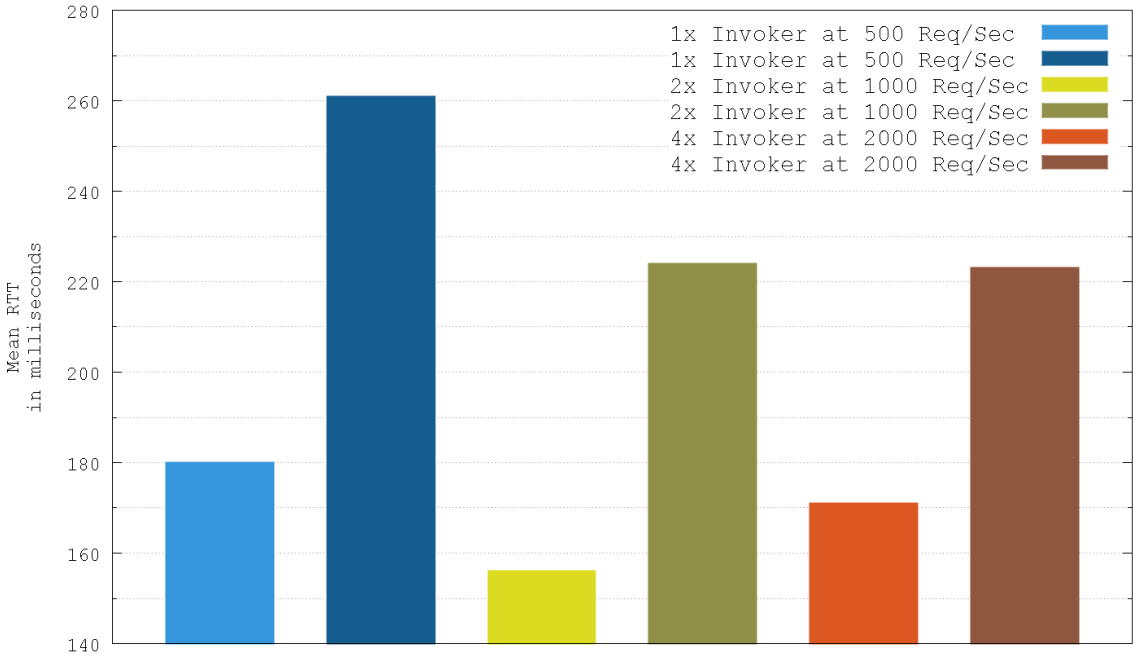

Figure 28 – Bar chart representing average end-to-end latency with constant message flow in presence of different kind of interceptors, light for internal and dark for external, at increasing level of invokers parallelism with Rust function execution in multi-cloud scenario.

When comparing multi-cloud and single cloud deployments, the primer accomplishes better results, as shown in Figure *28*. Sequential execution model achieved an average RTT of 81ms lower in multi-cloud deployment (from 261ms to 180ms) with a total reduction of about 31%.

On the parallelized platform instead, when dealing with 2x setup, it achieves an average RTT drop from 225ms to 156ms, but despite being relatively lower than the previous one, 69ms against 81ms, it constantly performed a nice 32% reduction.

The scalability trends are anyway maintained when the number of invokers, per phase, is incremented to 4x, in fact the usage of the external coordinator, the workflow RTT stabilizes at about 223ms whereas the platform with the internal coordination schema reaches 172ms and therefore a difference of 51ms, with a relative percentage reduction of 23%.

Next test series will evaluate the last case-study related to the invoked function and the way the system will perform in conjunction with very slow consumers.

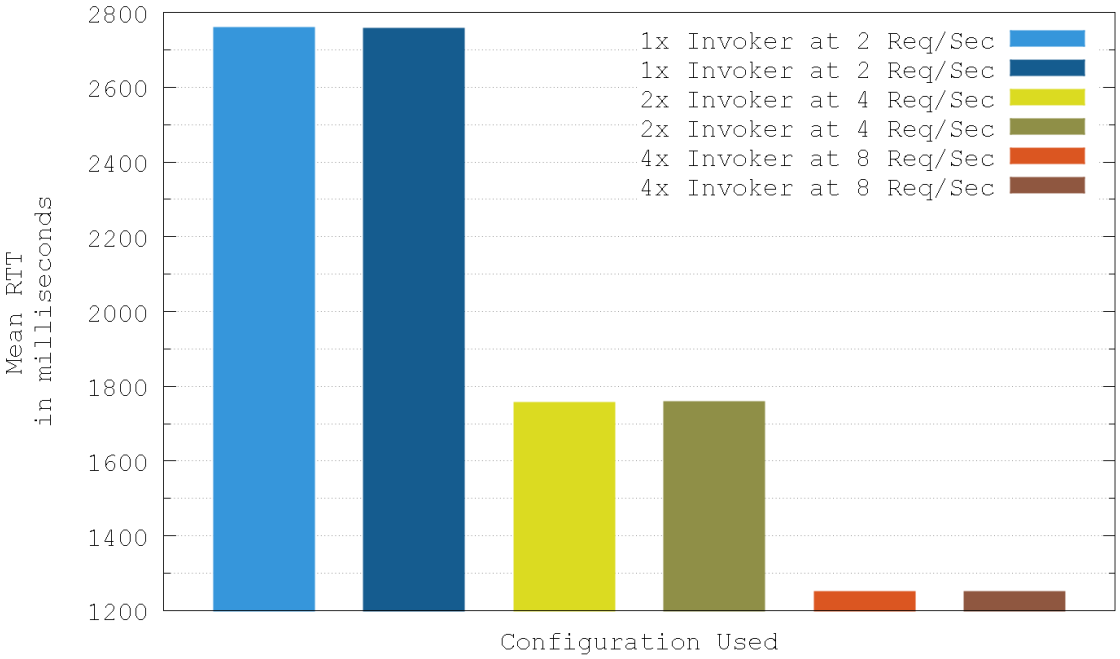

Figure 29 – Bar chart representing average end-to-end latency with constant message flow in presence of different classes of interceptors, light for internal and dark for external, at increasing level of invokers parallelism with JavaScript function execution in single-cloud scenario.

On the other hand, when invoker is tasked for the JavaScript *nodejs* function execution, results tend to flatten out their differences. The cue here is that the average RTT difference obtained through an internal or external coordination mechanism represents less than 0,01% of the total execution time. This behaviour is due to the interpreter execution time known to be even two orders of magnitude slower than the Rust counterpart.

That said, the architecture was benchmarked with a constant message emission rate of 2, 4 and 8 requests per second producing the following result: for a sequential execution the internal coordinator performed 3ms worse than the external with the corresponding values of 2756ms and 2759ms respectively, while in case of 2x concurrent invokers per phase, they get the values of 1757ms and 1756ms, hence with no substantial differences. The situation remains practically unchanged in the

4x scenario, where both configurations recorded average RTTs stick at the same 1251ms value.

The explanation behind these values could be found in the exaggeratedly high invocators execution time which let the platform to handle message exchange with minimal or no synchronization mechanism overhead, hence, invalidating the performance gain that could arise.

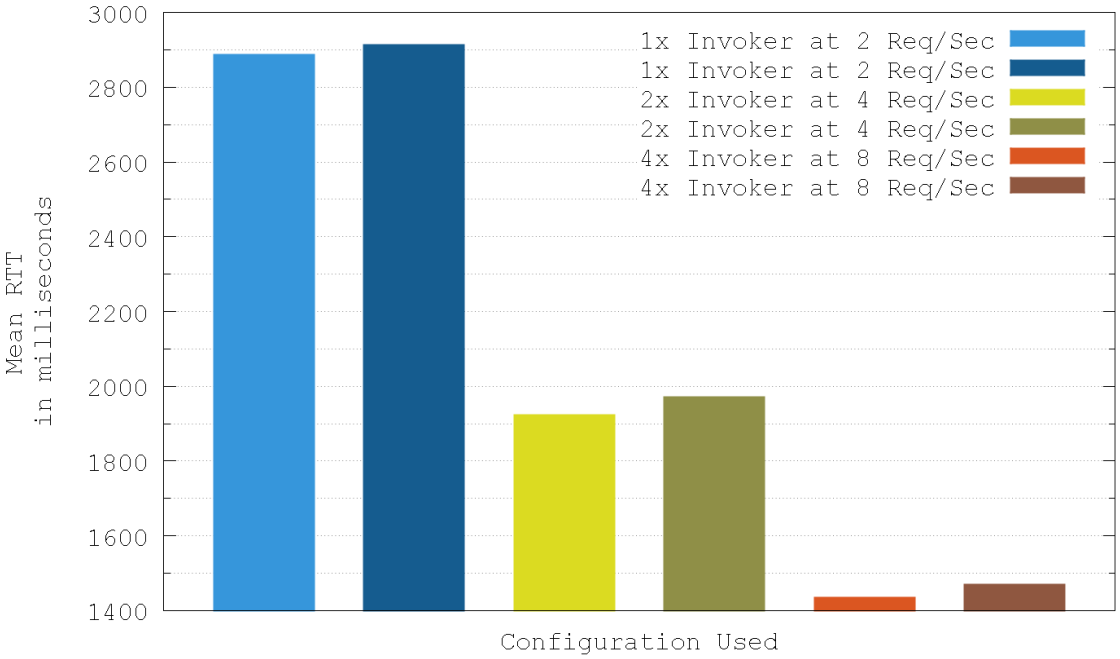

Figure 30 - Bar chart representing average end-to-end latency with constant message flow in presence of different classes of interceptors, light for internal and dark for external, increasing level of invokers parallelism with JavaScript function execution in multi-cloud scenario.

Differently from the previous test, as show in Figure *30*, the multi-cloud setups take advantage of reduced message time-of-flight. In x1, with its 2914ms external coordination RTT is about 30ms above the internal one with 2884ms, but if assessments are repeated with 2x and 4x invoker parallelization, the gap increases from 1923ms to 1972ms in the primer and keeps increasing more consistently in the latter with a 1434ms drop from 1470ms, thus a saving of almost 36ms in the average case.

At the end of these evaluations, we generally see that the proposed solution, either in case of single or multi-cloud scenarios, is capable of delivering, at least, the same

performance levels, which are highly dependent from number of coordination messages exchanged.

In the case of external coordination, implementing such a simple coordination logic, foresees the exchange of two more coordination messages. When in an internal coordination architecture, these messages cut-off which results, as shown in Figure *31*, in a total average time saving of 30ms to 100ms when message arrives to the Reduce Phase.

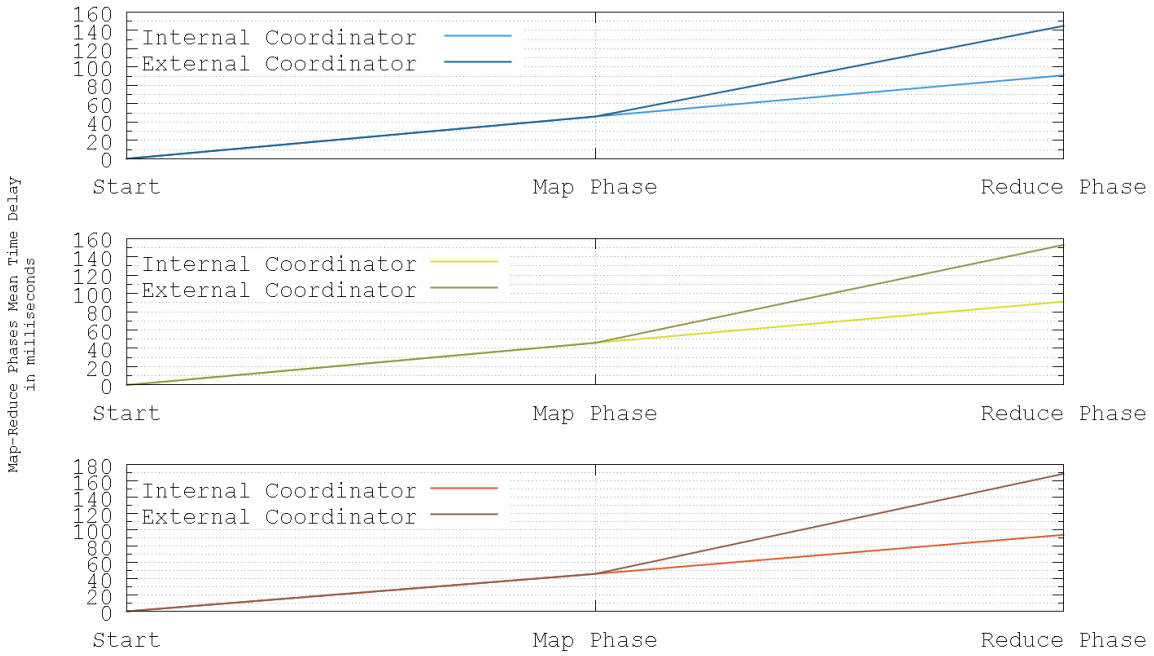

Figure 31 - Trend lines chart comparing 1x, 2x, 4x invoker setup, from top to bottom, average messages time of arrival at each processing pipeline stages, with constant message flow of 4k requests per second, in a multi-cloud scenario. Values are averages of the entire test execution.

Note that all these tests were carried out in a multi-cloud scenario, because it magnifies the gaps among all execution thanks to network communication overhead, imperceptible otherwise.

The next two indicators, as stated at the beginning of this chapter, will let us draw what are the MOM related resources utilization implications because they could be determinant when reasoning about the system scalability. To inspect those variables had been taken into account the worst-case scenario, that is the system in

subjected to the highest incoming request rates of 4000 requests per second in a single cloud-scenario and 4x parallel invokers per phase.

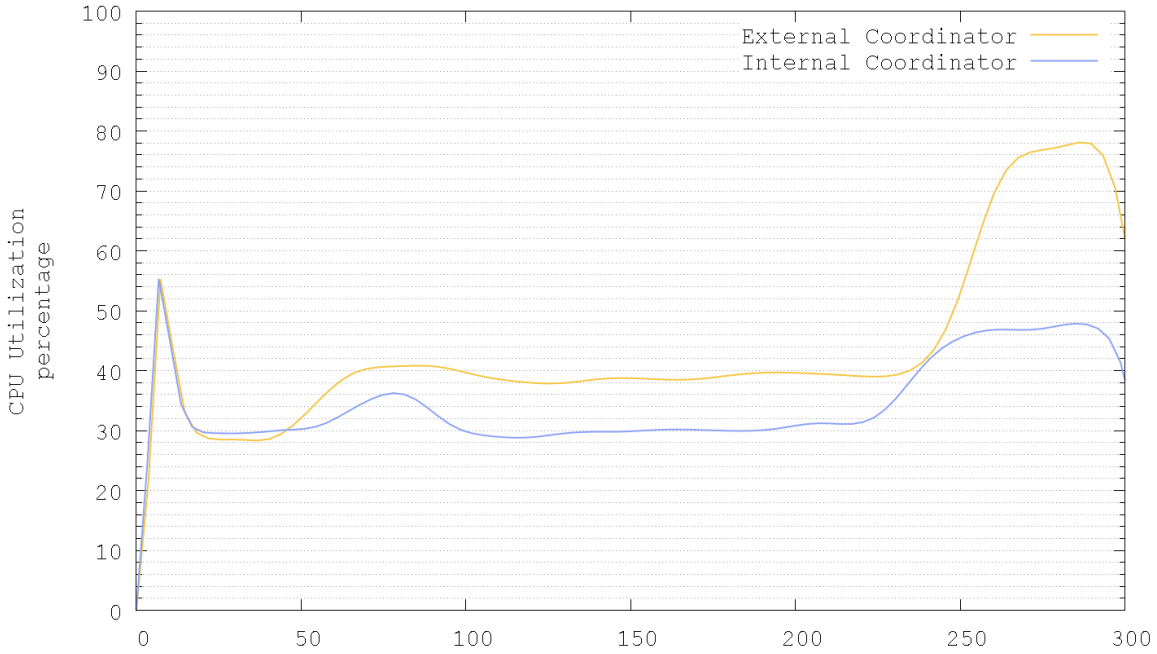

Figure 32 - Trend lines chart tracking CPU usage in a 4k constant incoming requests per second with Rust executable invocation, 4x invoker parallelism and single-cloud scenario. Data resolution is of 1 sampling per second.

How can we see in Figure 32 - , when the messages started to arrive at the MOM, a CPU percentage utilization spike occurs for both solutions, reaching its maximum at around 55%. This is due to the fact all tests were executed with preliminary cache invalidation and temporary file deletion that leads the broker to instantiate the necessary data structure and temporary files once again. The CPU usage then, after a brief oscillatory moment tends to converge to a steady value. The difference is that, for most of the execution time, our solution has at least 10% constant CPU less utilization up to when compared with external coordination processes. This is because the latter, whenever the coordinator needs to get all the requested information, they need to be read from the appropriate queue and, from the MOM perspective, is a CPU-bound operation because it sees a series of state synchronization operation which translates to higher CPU resources utilization.

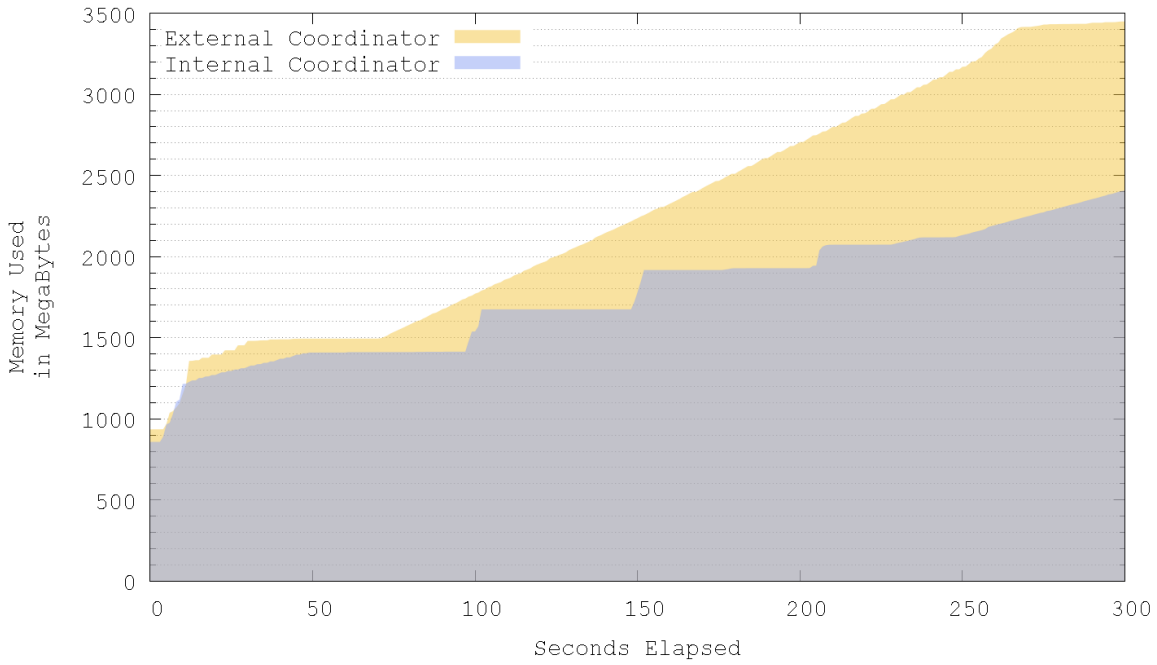

Figure 33 - Trend area chart tracking CPU usage in a 4k constant incoming requests per second with Rust executable invocation, 4x invoker parallelism and single-cloud scenario. Data resolution is of 1 sampling per second.

ActiveMQ Artemis delegate the memory management on the Java Virtual Machine and it is an important characteristic to remember because the resource releasing up to the underneath JVM completely. This means that at some point, not shown in Figure *33*, the garbage collector will be not-deterministically tasked to release the memory causing, even though for a short time, additional resource consumption.

Instead, what can be seen from Figure *33*, is that our solution tends to fill the available memory slower and so, when in real-word scenario, to reduce the number of garbage collector run.

The behaviour here let us consider our solution, in general, more memory efficient, more stable, and then less prone to sudden performance unwanted latencies.

5.2. Stream of Incoming Requests Emitted at Increasing Rate

In this second series of experiments, the platform is subjected to an incremental burst of concurrent requests so that its behaviour can be examined. To this end, the *stresser* produces a burst of invocation requests starting from 250 to 10k requests per second with a step increment of 250 messages per second.

This way, we intentionally exacerbate the queuing effect filling the queues up with invocation requests, as the invokers or the coordinator, will not be able to consume them properly.

The tests here proposed will follow the same structure adopted in Section 5.2 above, but additional data information has been collected and presented.

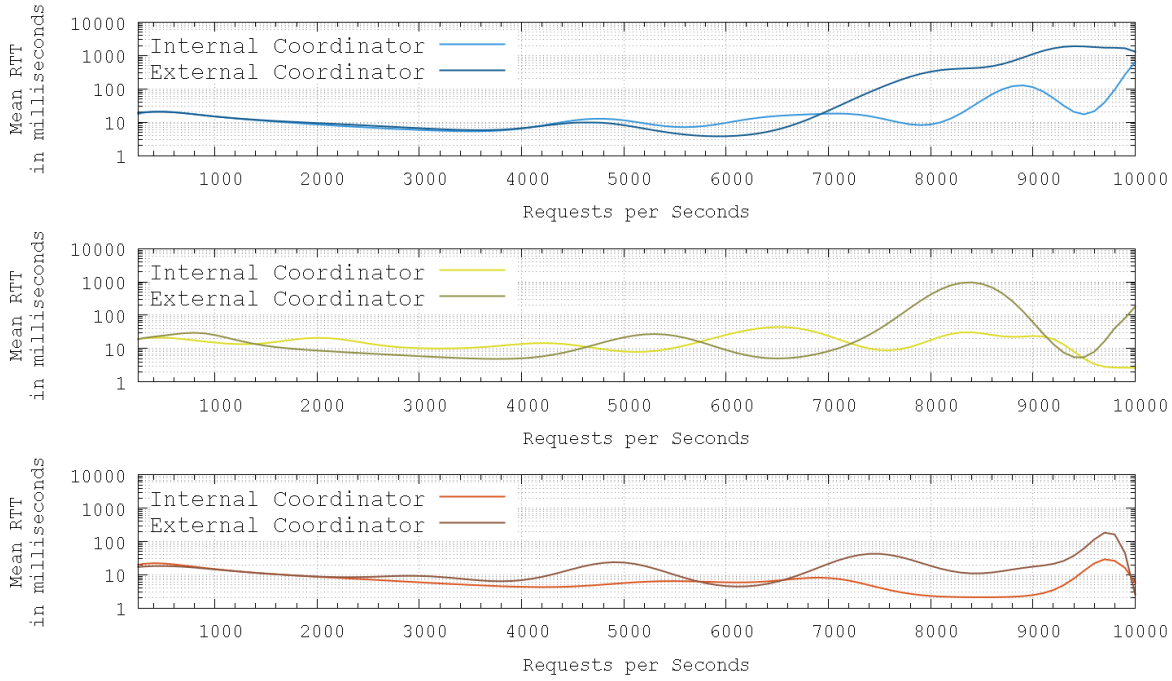

Figure 34 – Trend lines chart comparing 1x, 2x, 4x invoker setup, from top to bottom, in a burst of 250 to 10k requests per second with an increment of 250 requests per second without function call in single-cloud scenario. Every message burst is reported as a single latency value, which represents the average of the message latency values of that burst.

The chart in Figure 34 has been splitted into three different plots with logarithmic scaled y-axis, based on architecture parallelism, to improve readability. In fact, what we are interested in is the relative performance obtained into the single group, rather than comparing them.

What catches the eye is the fact that, in such single-cloud scenario, our solution keeps, for most of the execution timeframe, below the counterpart. Even when the system starts to enqueue the requests, observable by the increasing oscillating behaviour, our solution tends to maintain overall more consistent responses. At that point, in fact, the performance obtained are roughly two orders of magnitude lower. For example, when messages emission rate reaches 8k requests per seconds, the recorded values dropped from thousands down to tenth of milliseconds.

Another desirable effect of the adoption of an internal coordinator is that, limiting the broker internal synchronization mechanism, it is capable to postpone the moment in which the MOM starts to enqueue the incoming messages. In other words, the oscillating behaviour is shifted onward allowing heavier workloads with the same computational available resources.

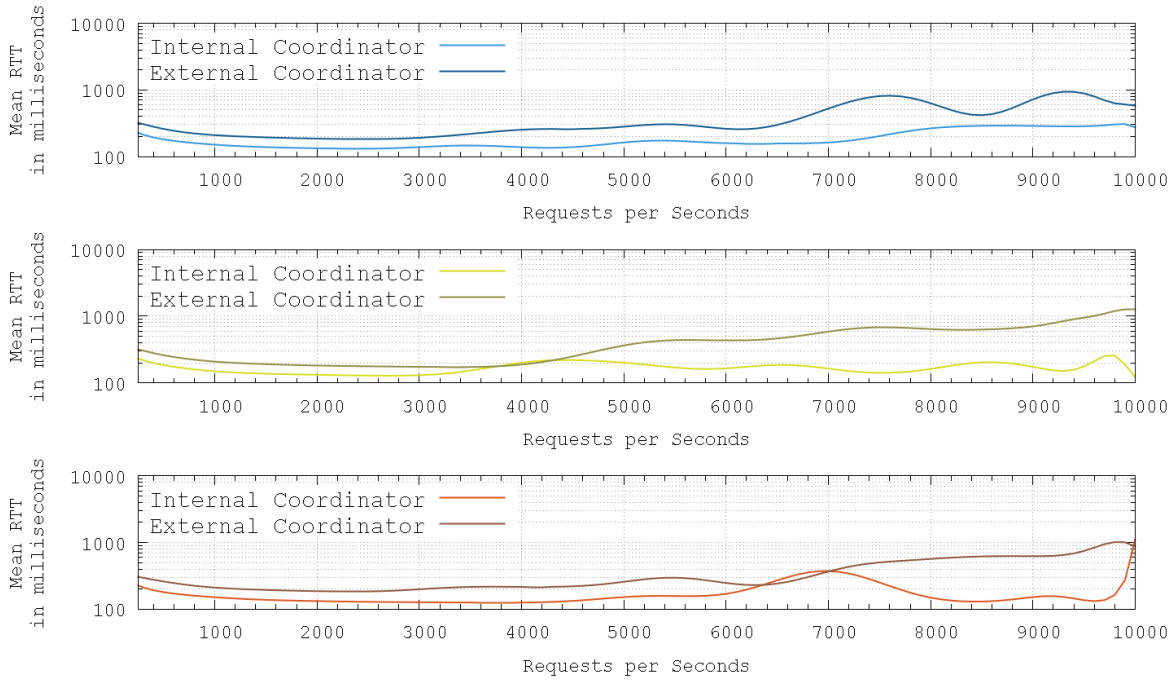

Figure 35 - Trend lines chart comparing 1x, 2x, 4x invoker setup, respectively from top to bottom, in a burst of 250 to 10k requests per second with an increment of 250 requests per second without function call in multi-cloud scenario. Every message burst is reported as a single latency value, which represents the average of the message latency values of that burst.

Aside from the previously advantages, in the multi-cloud environment depicted in Figure 35, brought the additional delay reduction due to the MOM close proximity of the coordinator component. In fact, as long as rates remain far below the oscillatory behaviour, the platform average RTTs can boast a halved time reduction from about 200ms to 100ms. Furthermore, since this behaviour occurs independently from the grade of parallelism, it confirms the scaling capabilities even when the number of interacting entities increases.

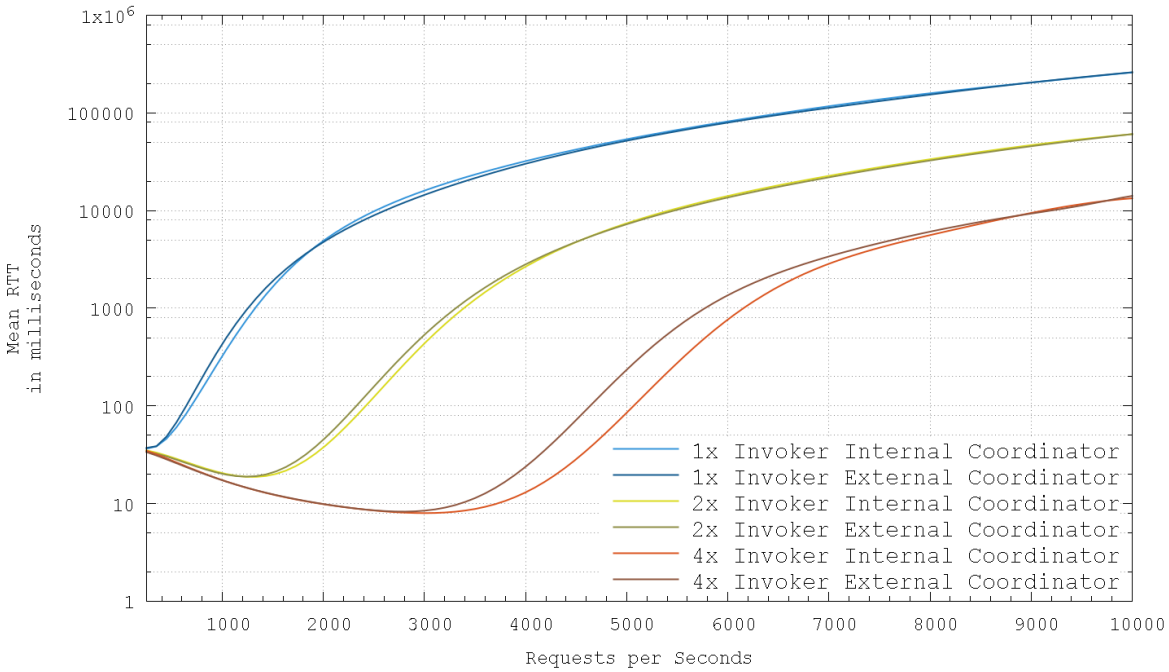

Figure 36 - Trend lines chart comparing 1x, 2x, 4x invoker setup in a burst of 250 to 10k requests per second with an increment of 250 requests per second with Rust function execution in single-cloud scenario. Every message burst is reported as a single latency value, which represents the average of the message latency values of that burst.

Moving next to Rust executable invocations, Figure 37 (note the log scale in y-axis) shows that when on single-cloud scenario the most preeminent accomplished results, coherently for what happens in last test, occur with the highest level of parallelism. With these setups in place, however, the enqueuing effect started to emerge at 400, 1200 and 3000 requests per second with 1x, 2x and 4x respectively. Interesting is the fact that, doubling the numbers of invokers, the local minimum shifts 3 times onward from 1x to 2x and 2,5 times onward from 2x to 4x. This suggests that, from a scalability perspective, the platform is capable to support concurrent ingress requests that does not linearly grow with the number of invokers.

With the bottleneck effect in place, hence, in the ranges just after the local minimum, our solution is capable to delivering an average 20ms RTTs saving.

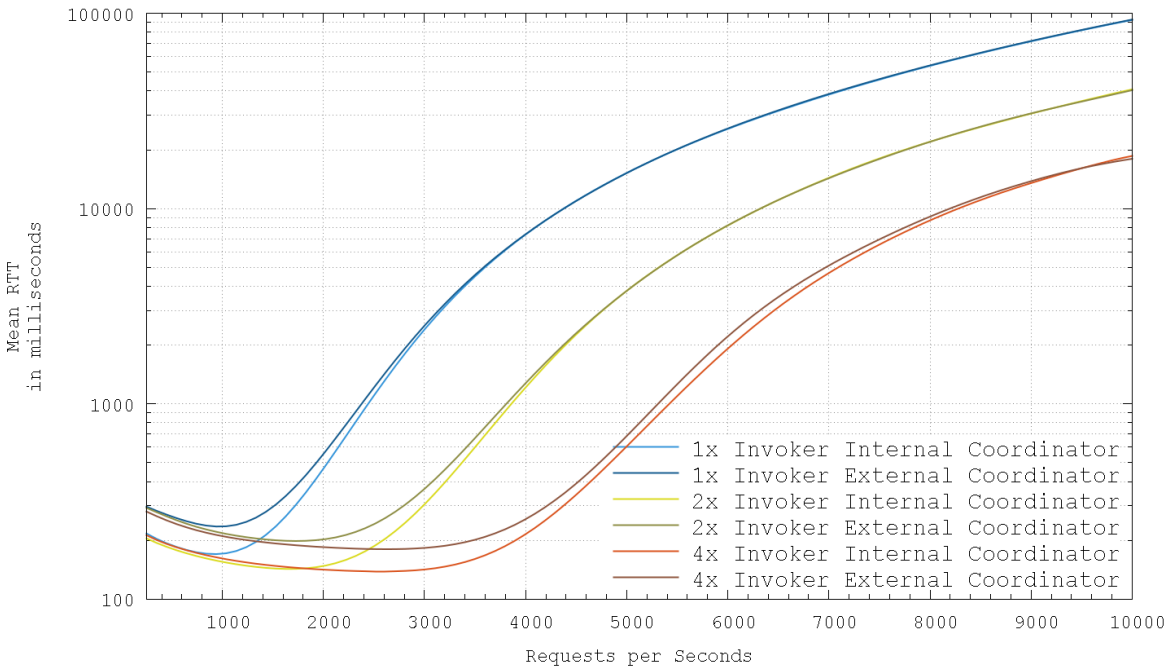

Figure 37 - Trend lines comparing 1x, 2x, 4x invoker setup in a burst of 250 to 10k requests per second with an increment of 250 requests per second with Rust function execution in multi-cloud scenario. Every message burst is reported as a single latency value, which represents the average of the message latency values of that burst.

Multi-cloud scenario, as foreseeable, has higher latencies gap among the twos coordinators implementations, chiefly due to the distributed location of the architectural components. Embedding the coordination logic inside the MOM leads a total reduction of about twice the average coordinator-MOM RTT pairs, that, as in Figure 37, of about 80ms.

Ultimately, scaling the number of invokers up does not degrade the overall architecture performance.

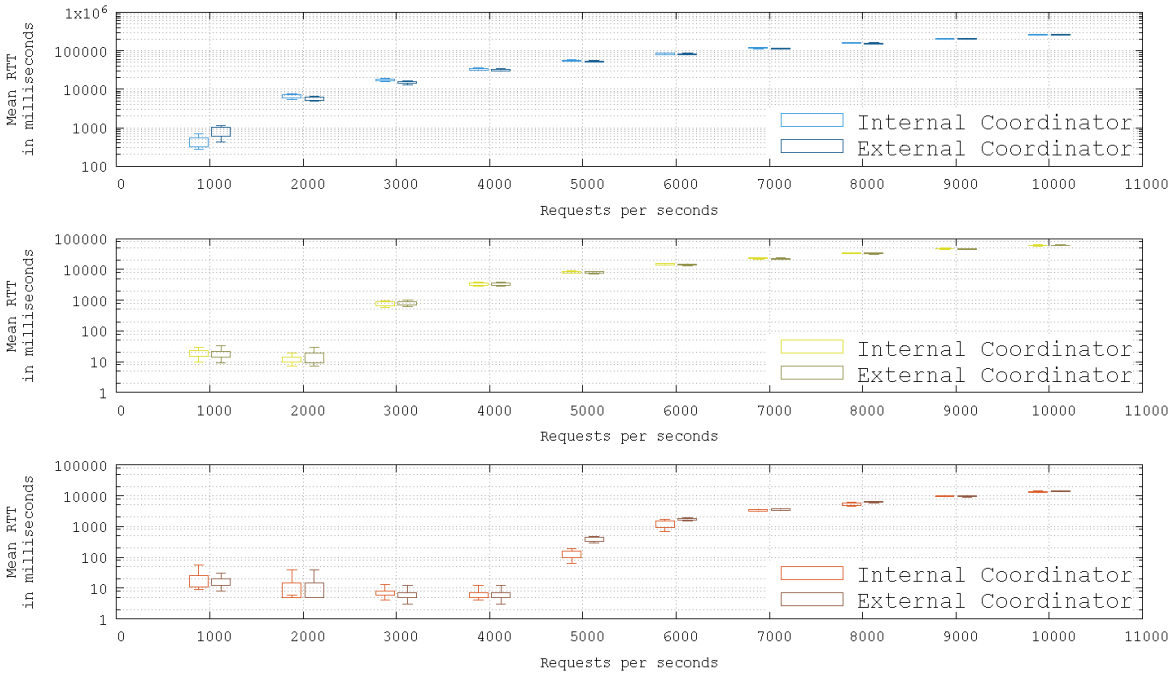

Figure 38 – Candlestick chart comparing 1x, 2x, 4x invoker setup, respectively from top to bottom, in a burst of 250 to 10k requests per second with an increment of 250 requests per second with Rust function call in single-cloud scenario. Every message burst is reported as a single value set, which represents the average of the message values of that burst. Each candle encloses 4 distinct parameters: the minimum as bottom whisker, the maximum as top whisker, the first and third quartile as the upper and lower box boundaries.

Until now, the principal metric taken into consideration, to compare the various architectures proposed and consequently determine the goodness of our solution, has been that of the average round trip delay. Unfortunately, the average value cannot be considered the only indicator, in fact having two sets whose average coincides, could be, actually, very different. As is well known, distributed application, are particularly sensitive in average value variation, therefore, a careful analysis of the latter could have important implications regarding the performance evaluations of the proposed solution. Chart in Figure *38*, It shows a different view of the iteration in single cloud, while the data has been selected generating a discrete probability distribution per 1000 units. The bar charts with whiskers presented here provide four additional features to those seen above. The upper whisker provides the maximum value recorded during the session, while the lower mustache, as easy to expect, the minimum value. The other two indicators can be obtained respectively

from the upper and lower box edges which provide us, jointly, the values included in the range defined as the distance one time the standard deviation from the mean value. With the results obtained from the test in in Figure 36, emerge that the solution with internal coordination is comparable to the solution with external coordinator in terms of minimum and maximum average latency. In addition, even the sampled distribution remains unchanged, i.e., the solution we propose does not introduce further relevant shaping skewness on the distribution.

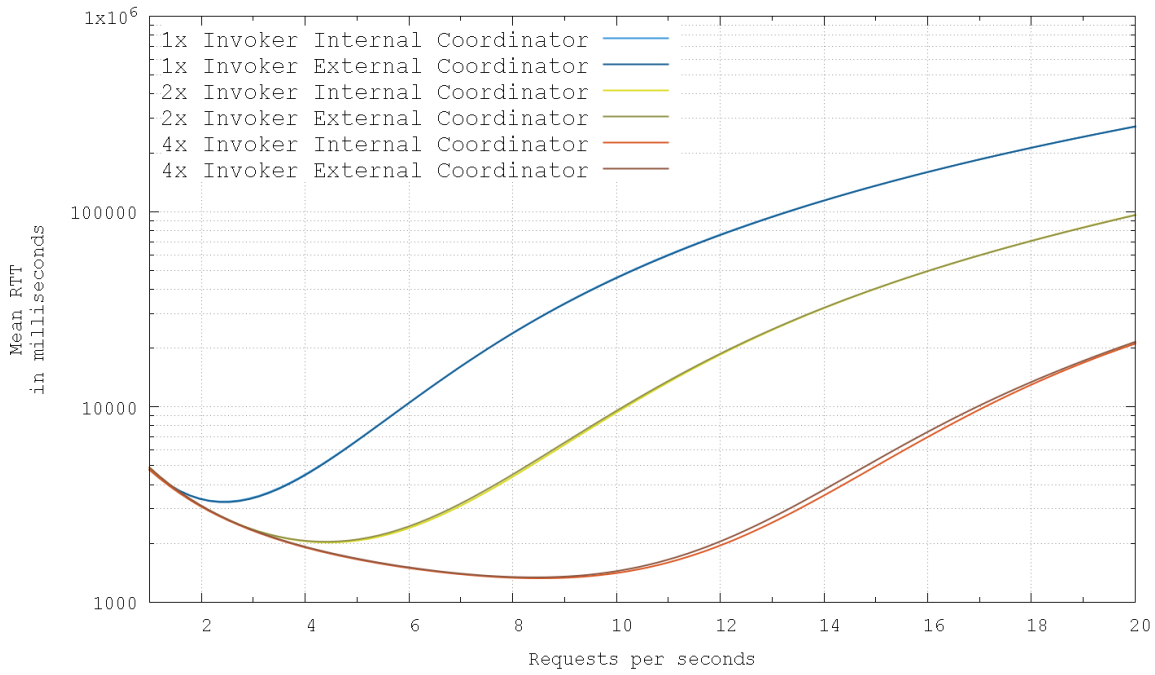

Figure 39 - Trend lines comparing 1x, 2x, 4x invoker setup in a burst of 1 to 20 requests per second with an increment of 1 request per second with JavaScript function execution in single-cloud scenario. Every message burst is reported as a single latency value, which represents the average of the message latency values of that burst.

In the event depicted in Figure 39, due to the extremely low *nodejs* execution and to keep tests reasonably short, we had to recalibrate *stresser* emission rate, so that, it produced a burst of messages ranging from 1 to 20 requests per second with a step increment of 1 message per second every 5 seconds.

In the same manner to what happened when a stream of incoming requests are emitted at steady regime, the benefit compared to the average RTT is to be

considered negligible suggesting, either in single-cloud, above, than multi-cloud deployments below.

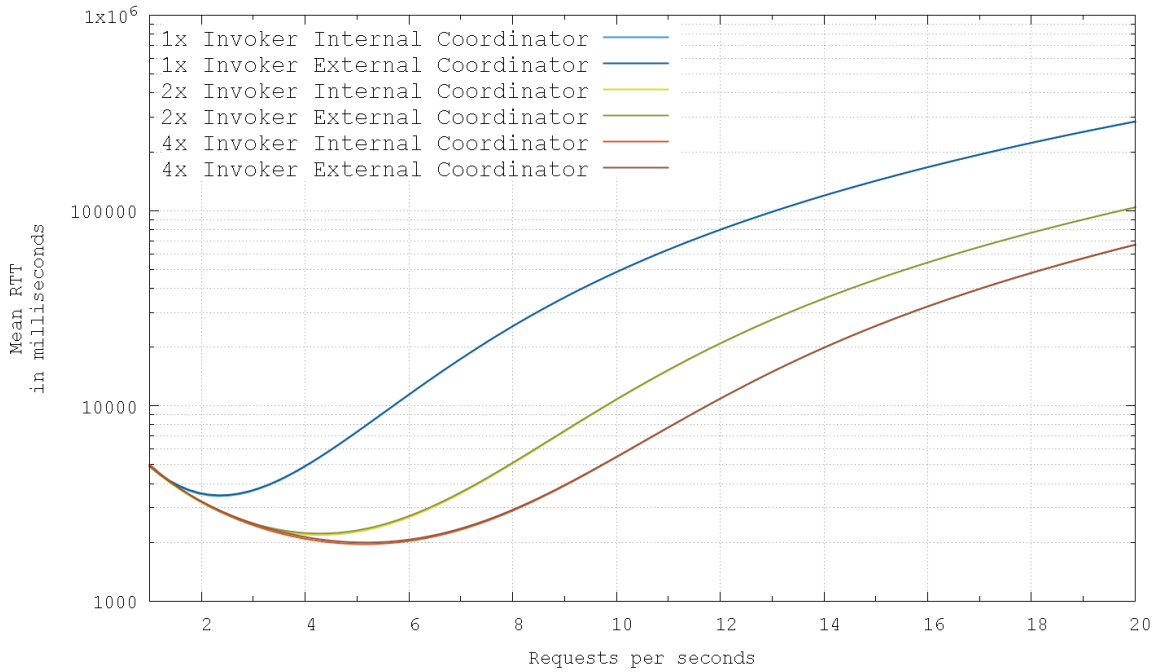

Figure 40 - Trend lines comparing 1x, 2x, 4x invoker setup in a burst of 1 to 20 requests per second with an increment of 1 request per second with JavaScript function execution in multi-cloud scenario. Every message burst is reported as a single latency value, which represents the average of the message latency values of that burst.

As a general consideration, what is evident from the experimentations shown in trend lines charts above is that, regardless of whether in single or multi cloud, the parallelism adopted and the actual function invocation mechanism, they present an initial sustained delay time.

The explanation behind this phenomenon, as already explained during the previous chapter of this thesis, it is attributable to the *cold start* problem.

With subsequent incoming requests, albeit depending on factors such as the number of requests and the invocation function, the architecture manages to satisfy more requests than the incoming ones. Consequently, as requests per second factor increases, the average response time tends to decrease gradually until it reaches a local minimum. This local minimum is to be considered as the maximum number of requests that the setup can sustain before starting to enqueue the requests. After this point, normally, trend is monotonically or globally increasing indicating that

new incoming requests compete with existing ones, causing enqueuing with an increasing delay in response.

As already discussed in the previous section, in Figure 41, the messages average time of arrival to the various stages of processing pipeline is shown.

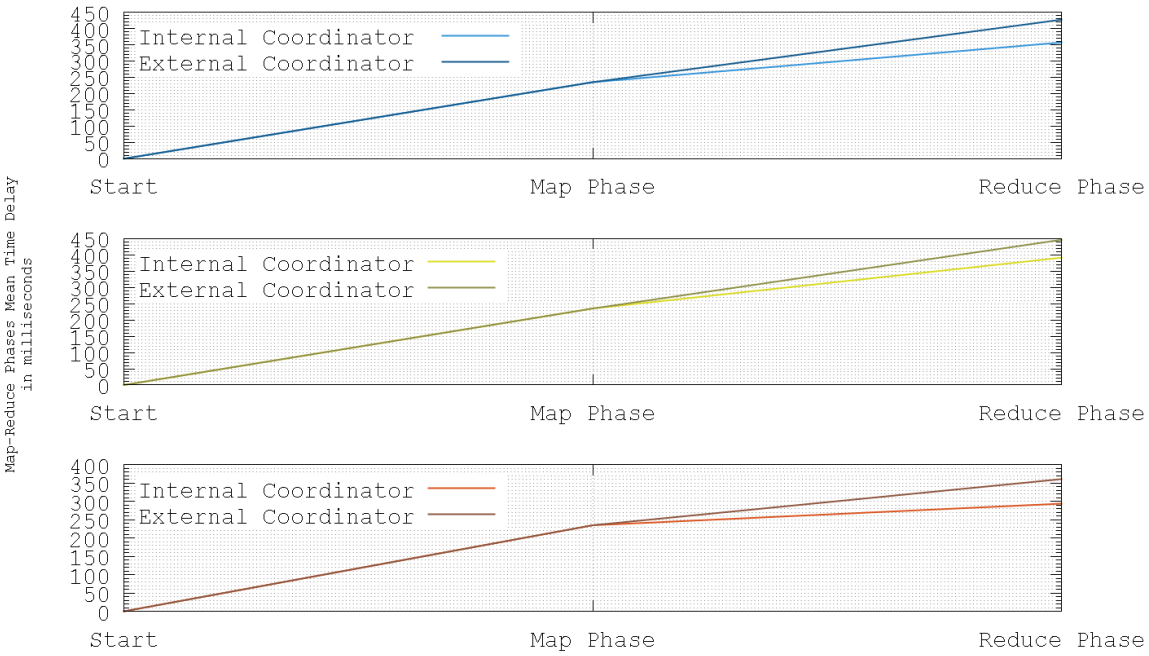

Figure 41 - Trend lines chart comparing 1x, 2x, 4x invoker setup, from top to bottom, average messages time of arrival at each processing pipeline stages, in a burst of 250 to 10k requests per second with an increment of 250 requests per second, in a multi-cloud scenario. Values are averages of the entire test execution.

As before, the solution with internal coordination at the MOM guarantees globally, compared to the total execution time of the test, the best arrival time to arrive at the reduce phase. As you can see, the improvement is always attributable to the coordination phase thanks to which the Reduce phase sees messages delivered with a lower latency of about 50-60ms. In fact, with sequential configuration the messages reach the reduction phase with an average delay of 357ms against 427ms, with a reduction of 17%., with 2x invokers the delay is expected at 391ms against 446ms with an improvement of 13% and finally with four invokers per phase there are 294msagainst 361ms with an improvement of 19%. Even in this situation, the

platform seems to maintain the desired scalability properties by not suffering a degradation of performance as the number of parts in the game increases.

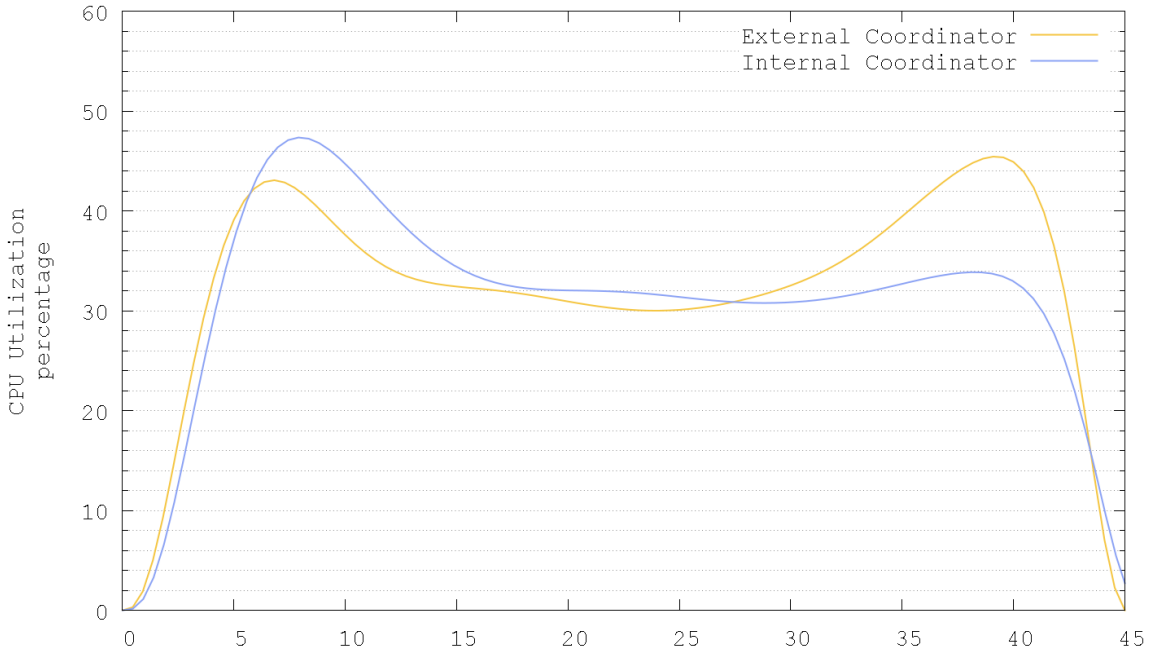

Figure 42 - Trend lines tracking CPU usage in a burst of 250 to 10k requests per second with an increment of 250 requests per second with Rust executable invocation, 4x invoker parallelism and single-cloud scenario. Data resolution is of 1 sampling per second.

When it comes to analyse the CPU utilization, instead, as shown in Figure *42* and as it was already explained on page 79, since the internal coordinator shares and competes the same hardware resources with the MOM, when the first message is being published, the broker has to handle them and instantiate the plugin, allocating all the needed resources.

However, this initial and inevitable peak represents only 5% of the CPU resources additionally with respect to the external coordination alternative and, due to its transient nature, it is intended to disappear quickly with minimal or even no real-world performance loss, considering that it would occurs once per platform reboot only.

After that moment anyway, our solution keeps pretty the same behaviour of the other all along the test execution.

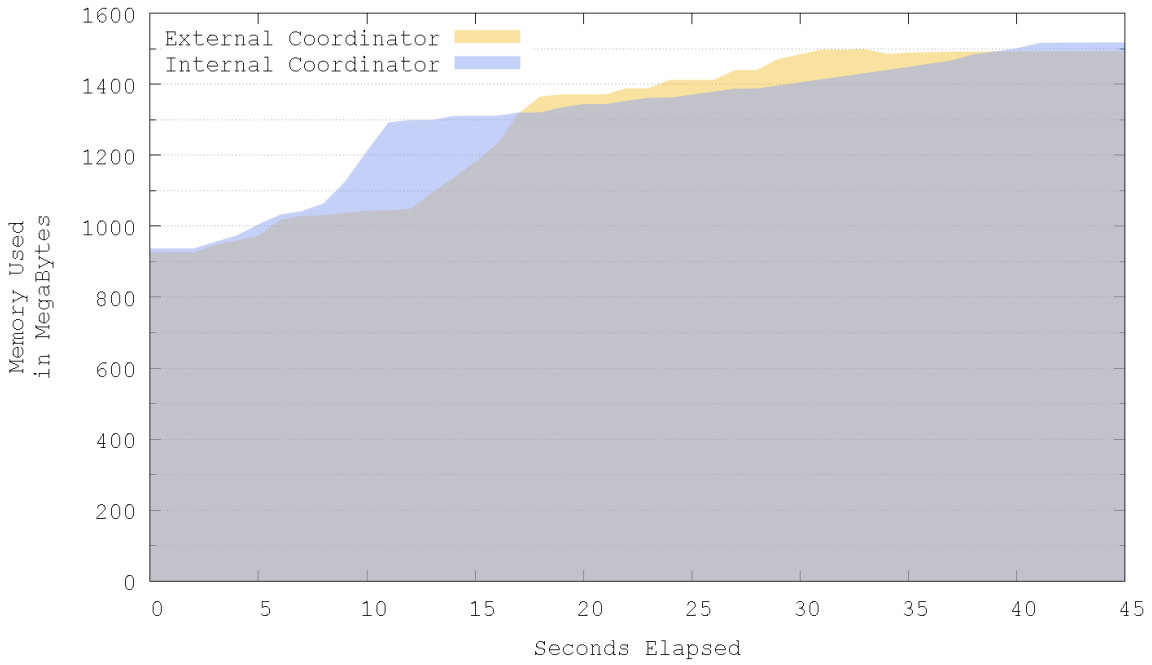

Figure 43 - Trend area chart tracking RAM in a burst of 250 to 10k requests per second with an increment of 250 requests per second with Rust executable invocation, 4x invoker parallelism and single-cloud scenario. Data resolution is of 1 sampling per second.

As far as concern the memory allocation, there is no substantial differences among the internal coordination strategy and the external one. In contrast for what happened in Figure 33, where the tests were carried out for 300 seconds, this one barely lasted 50 seconds and, the broker, were capable to follows a quasi-linear growth till the maximum occupation of 1500MB.

In summary, we started from the evaluation of the system in the case of constant workloads where we wanted to ascertain the architecture capability to support a typical FaaS workflow when fully operational. Subsequently, we moved on to incremental loads to simulate an information flow that would bring the architecture to saturation of its computational resources. The objective in this case was not only to understand if the solution was able to support more load than the counterpart, but to evaluate its scalability in terms of occupied resources.

At the same time, the scenarios proposed in the aforementioned configurations were tested in single cloud and multi-cloud deployment models in order to be able

to evaluate any improvement, in terms of average latency, compared to solutions that envisage an external coordination system for the composition of functions.

By specifying the results obtained in the case of a constant flow of requests, what emerges is the ability of the system to provide consistent improvements in terms of delays especially when compared to the execution time of short-lived functions in a single cloud scenario with sequential execution. When, on the other hand, this execution is performed in parallel contexts, again in single-cloud scenarios, these delays are in line with the results obtainable with solutions that envisage external coordination.

Our proposal, on the other hand, promises excellent performance when the deployment model is entrusted to a multi-cloud service where, the co-locality of the coordination logic with the MOM, allows a marked improvement in average response times, both in the scenario sequential than parallel.

However, regarding the use of resources, the use of an internal coordination system showed a more linear usage, while it overperformed in constant load conditions, where it proved to be clearly superior in managing the memory more effectively.

In the scenario where the trials were subjected to incremental load, however, the solution generally showed similar properties in terms of resource scalability in both single and multi-cloud scenarios. Furthermore, although highlighting a global improvement in response times, especially in the presence of short-lived functions, multi-invoker and multi-cloud, the internal coordination of the functions manages to manage message bursts more consistently than the externally coordinated solution.

Ultimately, because its ease of implementation and for the same reasons stated above, we ultimately sustain the effectiveness of our proposal whenever the usage of a MOM-centric function composition architecture is being used to adopt a MOM-embeddable coordination mechanism.

6. Conclusions and Future Developments

The serverless model of cloud computing encourages a greater degree of abstraction. This type of cloud computing enables customers to concentrate on the creation of business logic while transferring all responsibility for resource management to the platform provider. The ability to compose functions to create complex processing workflows is a feature that is becoming more prevalent and appealing in serverless computing, meanwhile, its flexibility uplifts the modularity and reusability of functions.

It is evident that serverless technology, thanks these many reasons, not least those of an environmental nature [35], is spreading at an exponential rate with a growing market demand. However, as anticipated, there is no de facto solution to the numerous problems afflicting serverless architectures, such as IT security and the level of performance currently achieved, which however, in corporate and mass scale scenarios are inadequate and not up to par to totally supplant current technologies.

However, the peculiarity of these architectures, i.e., the efficiency of resources, both computational for CSPs and financial for customers, make them one of the technologies most under research and development of this millennium.

In fact, many technologies advances have been introduced since their first development which, over time, have also allowed their adoption in, albeit limited, business scenarios for carrying out tasks with no disappointing expectation.

What has been done with this work aimed to exploit run-time optimizations through the usage of a message-oriented middleware as a conveyor of messages among all the FaaS components while experimenting how all these behave under different workloads and scenarios conditions.

Choosing the right message-oriented middleware implementation has been absolutely essential to fulfil all the intended objectives since ActiveMQ Artemis revealed to have, out-of-the-box, the required features among the commercial or non-commercial alternative solutions in message-oriented middleware, letting us to implement an interceptor as a plugin. In particular, as of today, we created a specific plugin that counts the number of messages having a specific destination in its header

and fires a specific event when the preconfigured condition is met. The metrics here collected let us to consider the validity of our proposal, exposing strengths and weaknesses, and its ability to come up with better performance while enforcing less resource consumption and minimal memory footprint.

However, multiple possible extensions could be considered. Firstly, would be interesting to evaluate the same proposal relying on a more complex testbed and execution environment, with an increasing number of invokers per phase or additionally increasing the pipeline stage number of mapping or reducing phases. Secondly, the realization of a more complex *Event Synthesizer* logic with enhanced event generation or advanced processing logic accordingly to workflow definitions. In fact, such definitions here implemented in a modulo counting, could be extended considerably introducing a new set of technologies specifically meant to describe and generate a super set of rules and logic implications, such as Domain-Specific Languages tools [36]. Those Event Synthetizes, as long as Queue Monitoring components could be implemented as plugins getting all the MOM potentials.

Thirdly, the development of a valid alternative to Rust command invocation since, especially in case of *nodejs* instantiation, it demonstrated a few margins of improvement and responsiveness during the tests. In fact, what could be exploited is the adoption of accurately meant framework which avoid the instantiation of the whole compiler.

Lastly, due the exceptional ActiveMQ Artemis performance and promising scalability features, it would be interesting to evaluate how distributed FaaS architecture behave in very sparse and far away nodes interacting in a multi-MOM scenario. For the same reason, since connectivity outage and even occasional networks disruption may occur during FaaS execution, it might be worthwhile to evaluate function composition in a MOM QoS-enabled deployment.

With that being said, we really belief that serverless architectures would be able to consolidate their presence on the market once current critical issues have been resolved or at least mitigated, continuing to be successful in important production choices replacing traditional systems in different application areas with an optimized usage of computing resources.

Index Of Figures

Index Of Tables

Index Of Listing

Index Of Equations

Bibliography

[1] National Institute of Standards and Technology, [Online]. Available: https://www.govinfo.gov/content/pkg/GOVPUB-C13-74cdc274b1109a7e1ead7185dfec2ada/pdf/GOVPUB-C13-74cdc274b1109a7e1ead7185dfec2ada.pdf. [Accessed 12 01 2023].

[2] M. Kleppmann, "Distributed Systems," Cambridge, 2020, pp. 34-44.

[3] IBM, "What is virtualization?," [Online]. Available: https://www.ibm.com/topics/virtualization. [Accessed 02 12 2022].

[4] OpenFaaS, "Scale-to-Zero," [Online]. Available: https://docs.openfaas.com/openfaas-pro/scale-to-zero/. [Accessed 01 02 2023].

[5] The Cloud Native Computing Foundation, [Online]. Available: https://www.cncf.io/about/who-we-are/. [Accessed 20 11 2022].

[6] Serverless, Inc, "Serverless," [Online]. Available: https://www.serverless.com/. [Accessed 07 02 2023].

[7] P. Rajan, "Serverless Architecture - A Revolution in Cloud Computing," Bengaluru, 2019.

[8] F. M., "Serverless Architectures," [Online]. Available: https://martinfowler.com/articles/serverless.html. [Accessed 02 12 2022].

[9] IBM, "What are message brokers?," [Online]. Available: https://www.ibm.com/topics/message-brokers. [Accessed 20 12 2022].

[10] Y. Du, W. Peng and Z. Li, "Enterprise Application Integration: an Overview," in *International Symposium on Intelligent Information Technology Application Workshops*, Shenyang.

[11] A. Corradi, A. Sabbioni, L. Rosa, A. Bujari and L. Foschini, "A Shared Memory Approach for Function Chaining," 2021. [Online]. [Accessed 20 10 2022].

[12] L. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah and O. Tardieu, "The Serverless Trilemma - Function Composition for Serverless Computing," p. 14, 2017.

[13] Amazon, "Step Function," [Online]. Available: https://aws.amazon.com/it/step-functions/. [Accessed 05 02 2023].

[14] P. Castro, P. Cheng, K. Chang, S. Vatche and L. Baldini, "Serverless Computing: Current Trends and Open Problems," [Online]. Available: https://arxiv.org/pdf/1706.03178.pdf. [Accessed 09 01 2023].

[15] A. Sabbioni, L. Rosa, A. Bujari and A. Corradi, "DIFFUSE: A DIstributed and decentralized platForm enabling Function," 2022. [Online]. [Accessed 15 09 2022].

[16] N. Kratzke, "A Brief History of Cloud Application Architectures," MDPI, 2018.

[17] B. Snyder, D. Bosanac and B. Davies, ActiveMQ in action, 2008, p. 375.

[18] F. Buschmann, K. Henney and D. Schmidt, Pattern-Oriented Software Architecture, vol. IV, Wiley Series, pp. 444-446.

[19] E. Gamma, R. Helm, R. Johnson and J. Vissides, Design Pattern - Elements of Resusable Object-Oriented Software, Addison-Wesley, 1995, p. 251.

[20] S. Klabnik and C. Nichols, "The Rust Programming Language," [Online]. Available: https://doc.rust-lang.org/book/. [Accessed 16 09 2023].

[21] Icraggs, eclipse, [Online]. Available: https://github.com/eclipse/paho.mqtt.c. [Accessed 01 02 2023].

[22] IBM, "What is MapReduce?," [Online]. Available: https://www.ibm.com/topics/mapreduce. [Accessed 20 10 2022].

[23] HiveMQ, “Reliable Data Movement for Connected Devices,” [Online]. Available: https://www.hivemq.com. [Accessed 20 12 2022].

[24] Scalagent Distributed Technologies, “JoramMQ,” [Online]. Available: http://www.scalagent.com/fr/jorammq/technologie/protocole-mqtt. [Accessed 16 02 2023].

[25] Mosquitto, “Eclipse Mosquitto An Open Source MQTT Broker,” [Online]. Available: https://mosquitto.org/. [Accessed 01 02 2023].

[26] RabbitMQ, “RabbitMQ,” [Online]. Available: https://www.rabbitmq.com/. [Accessed 17 02 2023].

[27] VerneMQ, “VerneMQ,” [Online]. Available: https://vernemq.com/. [Accessed 04 02 2023].

[28] OASIS Standard, “MQTT Version 5.0,” 2019. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html. [Accessed 19 01 2023].

[29] Apache Foundation, “ActiveMQ Artemis Documentation,” [Online]. Available: https://activemq.apache.org/components/artemis/documentation/latest/using-jms. [Accessed 06 02 2023].

[30] Microsoft, “Azure Network Round-Trip Latency Statistics,” 02 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/networking/azure-network-latency. [Accessed 15 02 2023].

[31] Microsoft, “Microsoft Global Network,” [Online]. Available: https://learn.microsoft.com/en-us/azure/networking/microsoft-global-network. [Accessed 01 02 2023].

[32] N. Heer, “A Repo Which Compares The Speed Of Different Programming Languages.,” [Online]. Available: https://github.com/niklas-heer/speed-comparison. [Accessed 18 01 2023].

[33] OpenJS Foundation, "NodeJs," [Online]. Available: https://nodejs.org/en/. [Accessed 14 02 2023].

[34] Network Time Foundation, "Reference Library," [Online]. Available: https://www.ntp.org/reflib/. [Accessed 15 01 2023].

[35] M. Aldossary, K. Djemame and I. Alzamil, "Energy-aware cost prediction and pricing of virtual machines in cloud computing environments," Future Generation Computer Systems, 2018.

[36] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov and C. Wu, "Serverless Computing: One Step Forward, Two Steps Back," Berkley.

[37] Apache, "Core Bridges," [Online]. Available: https://activemq.apache.org/components/artemis/documentation/1.0.0/core-bridges.html.

[38] OASIS Standard, "MQTT Version 3.1.1," 2019. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. [Accessed 15 01 2023].

[39] Apache Foundation, "ActiveMQ Performance," [Online]. Available: https://activemq.apache.org/performance. [Accessed 09 11 2022].

[40] R. Bolscher, "Leveraging Serverless Cloud Computing Architectures," Twente, 2019.

[41] S. Ghemawat and J. Dean, "MapReduce: Simplified Data Processing on Large Clusters," 2008.

[42] S. Klabnik and C. Nichols, The Rust Programming Language, 2018 ed., San Francisco: No Starch Press, 2018, p. 523.

[43] A. S. Gillis, "What is the internet of things (IoT)?," [Online]. Available: https://www.techtarget.com/iotagenda/definition/Internet-of-Things-IoT. [Accessed 01 02 2023].

[44] HTML, “Guida Node.js,” [Online]. Available: https://www.html.it/guide/guida-nodejs/. [Accessed 3 01 2023].