ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Dottorato di Ricerca in
COMPUTER SCIENCE AND ENGINEERING

Ciclo XXXV

SETTORE CONCORSUALE:
09/H1 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

SETTORE SCIENTIFICO DISCIPLINARE:
ING-INF/05 SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

Serverless Middlewares to Integrate Heterogeneous and Distributed Services in Cloud Continuum Environments

PRESENTATA DA: ANDREA SABBIONI

COORDINATORE DOTTORATO:
PROF.SSA ILARIA BARTOLINI

SUPERVISORE:
PROF. ANTONIO CORRADI

ESAME FINALE ANNO 2023

Abstract

The application of modern Information and Communication Technologies (ICT) technologies is radically changing many fields pushing toward more open and dynamic value chains fostering the cooperation and integration of many connected partners, sensors, and devices. As a valuable example, the emerging *Smart Tourism* field derived from the application of ICT to Tourism so to create richer and more integrated experiences, making them more accessible and sustainable. From a technological viewpoint, a recurring challenge in these decentralized environments is the *integration* of heterogeneous services and data spanning multiple administrative domains, each possibly applying different security/privacy policies, device and process control mechanisms, service access, and provisioning schemes, etc. The distribution and heterogeneity of those sources exacerbate the complexity in the development of integrating solutions with consequent high effort and costs for partners seeking them.

Taking a step towards addressing these issues, we propose *APERTO*, a decentralized and distributed architecture that aims at facilitating the blending of data and services. At its core, APERTO relies on *APERTO FaaS*, a *Serverless* platform allowing fast prototyping of the business logic, lowering the barrier of entry and development costs to newcomers, (zero) fine-grained scaling of resources servicing end-users, and reduced management overhead. APERTO FaaS infrastructure is based on asynchronous and transparent communications between the components of the architecture, allowing the development of optimized solutions that exploit the peculiarities of distributed and heterogeneous environments.

In particular, APERTO addresses the provisioning of scalable and cost-efficient mechanisms targeting: i) function composition allowing the definition of complex

workloads from simple, ready-to-use functions, enabling smarter management of complex tasks and improved multiplexing capabilities; ii) the creation of end-to-end differentiated QoS slices minimizing interfaces among application/service running on a shared infrastructure; i) an abstraction providing uniform and optimized access to heterogeneous data sources, iv) a decentralized approach for the verification of access rights to resources.

Contents

1 Introduction **1**

2 Distribution and Service Models towards the Cloud Continuum **8**

2.1 Cloud Deployment Models 9

2.2 Models of Distribution of Cloud Continuum 13

2.2.1 Multi-Cloud 14

2.2.2 Edge Computing 15

2.2.3 Towards the Cloud Continuum 18

2.3 Cloud Service Models 19

2.3.1 MaaS 20

2.3.2 IaaS 21

2.3.3 PaaS 22

2.3.4 SaaS 22

2.3.5 Everything as a Service (XaaS) 23

3 Function as a Service (FaaS) **25**

3.1 Architecture composition 27

3.2 Function Composition 34

3.3 Cloud Continuum enabled FaaS platform 37

4 Service and Data Integration in Highly Distributed Scenarios **39**

4.1 Smart Tourism Service and Data integration 40

4.2 APERTO5.0 45

4.3 APERTO5.0 Architecture Components Full Description 50

5 APERTO FaaS **57**

5.1 Bridging Layer 59
5.2 Delivery Layer 61
5.3 Processing Layer 63
5.4 Controller and Management Layer 64
5.5 Data Persistence Layer 66
5.6 Access Control Layer 71
5.7 End-to-end QoS Service Differentiation 77
5.8 Distributed Task Composition over Cloud Continuum Resources . . 87

6 APERTO FaaS Reference Implementation 95

6.1 Bridging Layer 95
6.2 Delivery Layer 97
6.3 Processing Layer 99
6.3.1 DLF (Dynamically Loaded Function) 100
6.3.2 WASMF (WASM Function) 101
6.3.3 FSpawn (Function Spawn) 102
6.4 Controller and Management Layer 102
6.5 Data Persistence Layer 103
6.6 Authorization Layer 106
6.7 End-to-end QoS Service differentiation 108
6.8 Distributed task composition over cloud continuum resources . . . 113

7 Experimental assessment and Test results 119

7.1 Processing startup methods Comparison 120
7.2 Data Persistence Performance 123
7.3 Authorization Performance 126

7.3.1 Component Performance 127
7.3.2 Decentralized Performance vs Centralized Performance . 129
7.3.3 Policy Performance 129
7.4 QoS Service differentiation evaluation 131
7.4.1 Event Delivery QoS Differentiation 135
7.4.2 Processing Prioritization 139
7.4.3 Full Stack 141
7.5 Function Chaining performance comparison 144
7.5.1 Constant-rate stream of incoming requests 146
7.5.2 Incremental rate stream of incoming requests 148
7.5.3 Burst of incoming requests 151
7.5.4 MoM enabled load balancing 152
7.6 Experimental assessment summary considerations 155

8 Conclusion and Future Works **161**

1 Introduction

The ever-increasing attention to topics of sustainability, personalization, and accessibility is driving many innovative changes and developments in many sectors like Smart Cities, Industry 5.0, and Smart Transportation. These sectors based on recent developments in network infrastructures, such as WiFi6 and 5G, aim at integrating an ever-increasing number of data sources and services coming from sensors, the Internet of Things (IoT), mobile devices, and even other business partners. Although cloud computing is at the forefront of this technological ecosystem, a one-size-fits-all approach is no longer feasible. In fact, neither the cloud nor the networks connecting these objects to the cloud were designed to serve and process the enormous data volumes originating from geographically dispersed and heterogeneous endpoints.

Modern cloud solutions are then evolving toward the so-called *Cloud Continuum (CC)* a model representing a continuum of resources and aiming at integrating large-scale centralized public cloud deployment with distributed and **near-the-device** computational resources such as edge computing or private clouds.

The *Cloud Continuum* constitutes a revolutionary paradigm, proposing a shift from a centralized to a fully distributed architecture, and many new forms of cloud computing support models are emerging.

Among them, Function as a Service (FaaS) is gaining more and more attention in the market and in academic research also thanks to its low development effort, **event-centric** architecture, and **fine-grained** scalability of functions. This service delivery model generally requires minimum development and management effort from customers, emphasizing the absence of control and knowledge of the developer of where the code will be put in execution. Thanks to this paradigm

and the associated platforms, the development process of new services and the upgrade of existing ones is limited to the creation of the business functionality since the configuration, setup, and deployment are transparently managed by the infrastructure. In FaaS the complete absence of control over the infrastructure enables the fast creation and deployment of new services over large and, in principle, unlimited, heterogeneous infrastructures.

The evolution of these technologies is suggesting many scenarios where a continuum of data sources, services, and computational resources belonging to different providers and business actors interconnects and cooperates in order to create a more integrated and optimized value chain benefiting end users and businesses at all levels. However, the **heterogeneity** in protocols, formats, capabilities, and constraints characterizing these scenarios hampers the acceptance of common standards and poses a high barrier in terms of complexity and costs for the creation of integrating solutions.

To solve these problems we propose APERTO, a layered architecture for service and data integration in distributed scenarios involving many actors with different offerings and needs. APERTO aims at proposing a standard architecture mitigating the problem of heterogeneity in integration by proposing an architecture organized in layers, hiding aspects related to the technologies used, and grouping the different tasks. The aim of APERTO is not only to exploit better-existing resources for greater business value but also to stimulate the creation of novel services toward the whole potential of aggregation and augmentation of the platform. APERTO is a platform that aims to provide a single access point for advanced and augmented information and facilities to third-party organizations, both private and public, by presenting a unified view of services and information.

Core of adaptation, integrating, and fast-development capabilities of APERTO is a novel Function as a Service platform, called APERTO FaaS, specifically developed to exploit heterogeneous resources available on the CC. APERTO FaaS promotes **strong decoupling** and **asynchronous** interactions among architectural components facilitating the integration of heterogeneous and unreliable resources. These properties also facilitate the creation of abstractions and optimizations enabling customers to exploit transparently peculiar characteristics of each cloud continuum resource. To support integrating heterogeneous distributed data sources and services APERTO FaaS integrates specific functionalities supporting developers to cope with the complexity of these scenarios.

Emerging Cloud Paradigms is seeing a rising number of frameworks and programming languages concurrently employed to compose and create services. Those units of execution, whether implemented via microservice or functions, are characterized by more ephemeral liveness as a consequence of the ever-increasing adoption of fast auto-scaling techniques. These changes prevent the adoption of mechanisms realized at the programming framework layer to uniformize and optimize operativity on data storage.

To mitigate this problem APERTO FaaS integrates abstractions and optimizations specifically designed to decouple user-defined business logic from operational aspects, such as protocols, dialects, and location, characterizing interactions with data stores. These abstractions facilitate the execution of arbitrary workloads integrating **heterogeneous data sources** available in the continuum of resources.

The execution on resources available in the CC, however, poses new challenges in terms of QoS differentiation between the different services developed and hosted in the continuum of resources. The types of applications that are

requested to run on these distributed infrastructures are very differentiated and with very differentiated requirements, from latency upper-bounds to maximum allowable downtime and reliability; moreover, the ICT infrastructures hosting them include very heterogeneous resources and tend to employ more and more cloud continuum virtualized resources, which are typically positioned close to IoT sensors and actuators for greater efficiency [1]. At the same time limited availability across the continuum of resources in terms of computational capacity, network throughput, or latency can create situations of contention among the workflows executed. To fill this relevant gap, APERTO FaaS proposes a novel approach able to coordinate the different QoS mechanisms available over technologies across the stack of virtualized FaaS invocations in the cloud continuum to properly manage **end-to-end QoS** in terms of jitter, latency, and en-queuing time.

Advanced and complex capabilities of coordination in the cloud continuum are essential not only to guarantee the respect of constraints in terms of QoS, but also for the rising need of composing elaboration of information and execution of services dislocated in different sites creating more complex and integrated workloads and services. By decoupling complex functionalities into simpler ones, composition enables smarter management of complex tasks and improved multiplexing capabilities. Moreover, the intrinsic distribution of our targeted scenarios makes consolidation of services and data on the same infrastructure cost-ineffective and in some cases technically impracticable.

Mechanisms of orchestration and compositions over the continuum of resources are an ever-appealing feature enabling integration of service and data provided by heterogeneous partners and devices with near to the source execution of business logic. However, heterogeneity and distribution of resources and data in-

trinsic in scenarios of interest for this thesis hampers the creation of composition solutions with accessible complexity barriers for developers belonging to many realities.

Moving a step closer to addressing these issues, APERTO FaaS proposes an innovative solution for FaaS function composition enabling optimized information and service integration and aggregation over CC resources. APERTO FaaS exploiting the FaaS models enables to express integration of sources as a composition of functions abstracting from their complexity and heterogeneity. The composition mechanism of APERTO FaaS, by relying on principles of transparency and asynchronicity in components interaction, opens also to the creation of **local optimizations** taking advantage of single-site capabilities such as high-bandwidth networks or optimized node to node communications.

Finally, in scenarios including many partners and devices is easy to understand the need for an efficient mechanism to regulate the rights of the different actors. Current cloud solutions rely on centralized verification services that hardly adapt to distributed scenarios where the fault of a portion of the infrastructure could lead to severe unavailability of the service. We then propose a **decentralized authorization laye**r able to verify request permission to access services or resources by exploiting distributed computational resources available in the cloud continuum. Our proposal, furthermore, promotes a decoupling abstraction able to separate access control logic from customer define ones by encouraging modularity and reusability of code. The authorization layer is then specialized to enable the verification of access right at the triggering of the function of the FaaS platform, providing a boost in the verification process and reducing consequently end-user perceived latency.

In short, we believe that this thesis advances significantly the state-of-the-art literature in the field by proposing the following primary original contributions: (i) APERTO architecture to abstract the aspects related to the technologies used and group the different tasks and address the problem of heterogeneity and dissemination of information and services in contexts characterized by multiple actors and partners. (ii) APERTO FaaS platform enabling to integrate heterogeneous resources available in the cloud continuum and to promote asynchronous and decoupled interactions among architectural components to achieve better scalability and fault-tolerance (iii) An original cloud persistence layer abstracting and decoupling from single storage implementation and optimizing data operations performance in FaaS platforms. (iv) An orchestration mechanism able to effectively achieve a strong end-to-end differentiation of service QoS leveraging on heterogeneous prioritization mechanisms (v) An efficient function composition architecture enabling to execute distributed workloads integrating the execution of multiple functions leveraging on peculiarities of cloud continuum resources (vi) A decentralized and distributed access control architecture decoupling customer-defined business logic from the different authorization domains. (vii) An original implementation of the proposal based on cloud-oriented technologies and patterns that are shaping the FaaS and CC markets along with experimental results that quantitatively show the feasibility of the proposed approaches and the efficiency of the proposed implementations

The remainder of the thesis is organized as follows. The first two chapters (2, 3) give some background on Cloud Computing evolution over the past years with a focus on new models of distribution and cloud services. We will then focus on FaaS computing, the capabilities of this new cloud computing model, and the def-

inition of roles of recurring architectural components composing these platforms. Section 4 gives to the reader an overview of factors at the base of the creation of distributed and heterogeneous scenarios and presents our proposal for an architecture aiding the effective integration of services and data in those scenarios. Section 5 proposes our novel FaaS platform, the central technological component of our integration architecture followed by implementation choice and technologies integrated into our first prototype. We will then present an extensive test bed demonstrating the feasibility and benefits of our proposed architecture(sec. 6). Finally, in Sec. 8 conclusions are drawn and some directions for future work are discussed.

2 Distribution and Service Models towards the Cloud Continuum

In the past years, *Cloud Computing* has changed absolutely ICT by providing convenient access through the network to **virtually unlimited** computing resources (e.g., networks, servers, storage, applications, and services). In this chapter, we will walk through the evolution of this concept from the originally centralized model, providing a limited variety of offering models, to an infrastructure integrating a continuum of distributed resources offered to customers through many differentiated service models.

At the base of the cloud computing model is the provisioning of computational resources on which customers can run applications and services. These resources are configured and rapidly provisioned by the cloud provider, with reduced customer management effort, and following a **pay-per-use model** [2]. As a support to the billing methods of Cloud Platforms, providers have integrated extensive monitoring solutions to keep track of the usage of Information Technology (IT) resources. This allows the cloud provider to charge consumers depending on how many resources were used for a specific time frame. In that sense, measured usage is closely related to the **on-demand** characteristic. The extensive monitoring of resources enables to discharge users from many duties in terms of physical resource management and orchestration that are demanded to the Cloud Provider.

In Cloud Computing, in fact, the end user is not necessarily aware of provided resources exact location, but they may be able to specify the region or zone like country, state, or even Data Centers. To support the ever-increasing demand for Cloud resources many companies have developed distributed infrastructure

spanning many regions all over the globe and providing ready-to-use resources to many customers. The huge **availability** of resources distributed across many regions has enabled to cloud platform to provide unprecedented resiliency capabilities. In fact, cloud services provide robust systems through the distribution of redundant copies of the resources across physical locations. In case of one resource becomes deficient, another redundant copy can take it over. Thanks to the resiliency of cloud-based IT resources, cloud consumers can increase both the *reliability* and *availability* of their applications.

A key characteristic of Cloud architectures is their **rapid elasticity**. Cloud services can transparently scale by provisioning and releasing resources, as required in response to changing runtime conditions. In many cases, the demand for resources changes throughout the year, month, or even the day. Elasticity allows customers to change the allocation of resources dynamically, adding facilities to deal with service peak times and removing them when they are no more needed, paying just for the facilities they use. From a customer perspective, this reflects the great flexibility capacity of cloud platforms to accommodate the varying needs of businesses. As an example, a startup can begin by leasing minimal computing, storage, and communication facilities. Then, it can adapt to the requirements by adding more resources.

2.1 Cloud Deployment Models

Before Cloud Computing emerged, private clusters and grid computing made use of many parallel machines to solve highly demanding jobs, while utility computing provided services managed and delivered remotely by one or more providers through a pay-per-use subscription model. The concept of cloud computing has

gradually evolved from these models and the way resources are owned and organized defines different kinds of models in which cloud offerings can be divided. In particular, a cloud deployment model represents a specific type of environment, primarily distinguished by ownership, size, and access. Since the early beginning, four models of deployment emerged: i) the *public cloud* accessible by every customer in the world, ii) the *community cloud* reserved to users belonging to a federation of entities, iii) the *private cloud* confining access to a resource in the boundaries of a company, and iv) the *hybrid cloud* integrating multiple of aforementioned models (Fig 1).

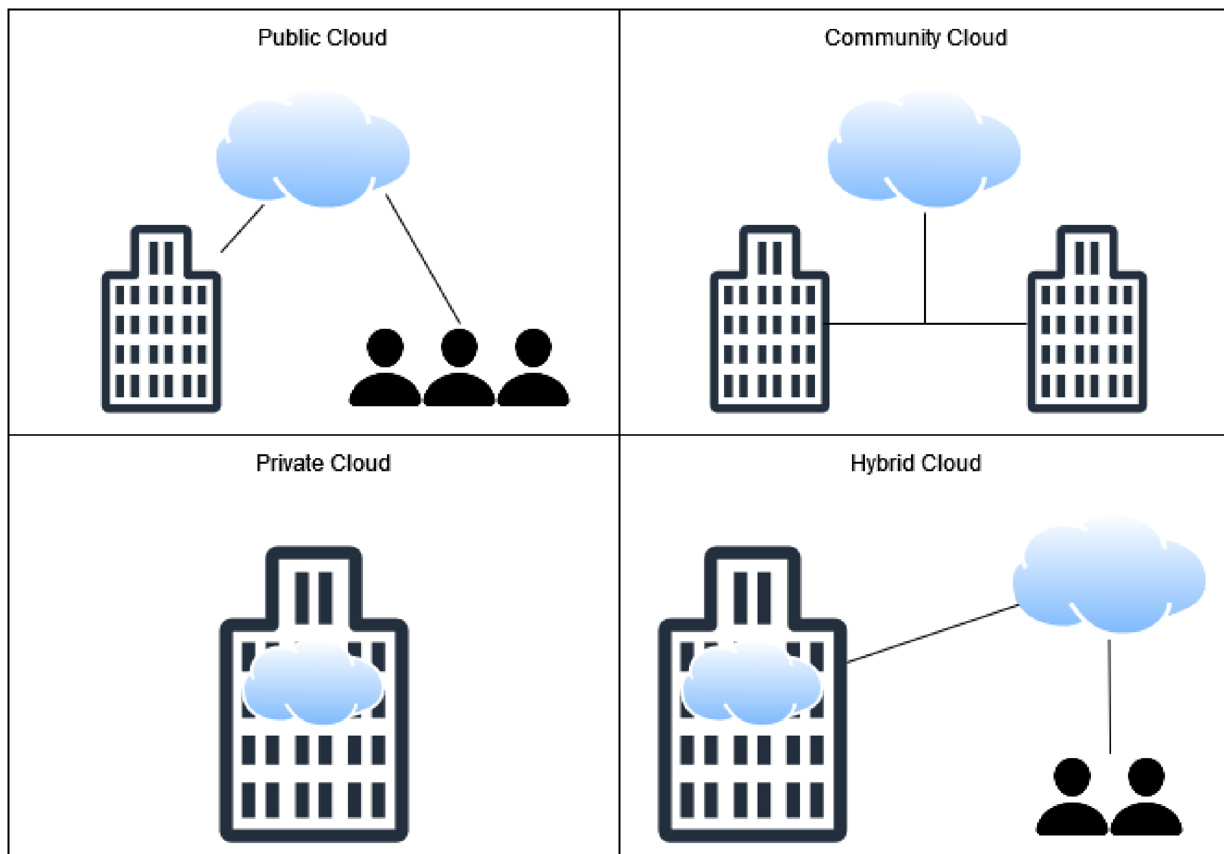


Figure 1: Cloud deployment models, distinguished by infrastructure ownership and permission to access resources

A *public cloud* is a type of cloud computing that delivers computing resources (e.g., storage, networking, servers, applications, and services) over the internet from every location all over the world. It allows potentially **any users** to access and use these resources on a pay-per-use basis, rather than having to build and maintain their own infrastructure. Public clouds are owned and operated by third-party cloud service providers, which offer their resources to the general public over the Internet. These providers are responsible for the maintenance, security, and operation of the underlying infrastructure, as well as the development and management of the cloud services. Public clouds offer a number of benefits, including flexibility, scalability, and cost-efficiency. They allow users to access a wide range of resources on-demand, and to scale their usage up or down as needed, without the need to invest in and maintain their own infrastructure.

A *community cloud* is a type of cloud computing that is shared by a specific community of users, such as a group of organizations with common interests or a specific industry. It is typically owned and operated by one or more of the organizations in the community and is tailored to address the specific needs and requirements of that community. Like a public cloud, a community cloud delivers computing resources (e.g., storage, networking, servers, applications, and services) over the Internet. However, it is designed to meet the specific needs and requirements of a particular community and is usually only accessible to members of that community. A community cloud can offer a number of benefits, including cost-efficiency, security, use **case-oriented optimizations**, and compliance. By sharing resources among a group of organizations with similar needs, a community cloud can help to reduce the cost of infrastructure and management, while providing a secure and compliant environment for the community's specific needs.

Private clouds are typically owned and operated by the organization using them, or by a third party on behalf of the organization. They may be hosted on-premises, in a data center owned and operated by the organization, or they may be hosted by a third-party provider and accessed over a private network. Private clouds offer a number of benefits, including security, compliance, and **control**. They provide a secure and compliant environment for sensitive data and workloads and allow the organization to have more control over the infrastructure and services being used. They can also offer increased performance and reliability, as the resources of the private cloud are dedicated to the organization and not shared with other users.

A *hybrid cloud* is a type of cloud computing that combines the benefits of both public and private clouds. It allows organizations to use a mix of on-premises, private cloud, and third-party, public cloud services, depending on their specific needs. In a hybrid cloud environment, an organization can use the public cloud for tasks that don't require a lot of customization or sensitive data and the private cloud for more sensitive or critical workloads. The differentiation among private and public resources also enables a first **decentralization**, exploiting data and processing locality for latency-sensitive workloads. This allows the organization to take advantage of the scalability and cost-efficiency of the public cloud, while still being able to maintain control and security over certain workloads. A hybrid cloud can offer a number of benefits, including flexibility, scalability, and cost-efficiency. It allows organizations to choose the most appropriate cloud environment for each workload, based on their specific requirements, and to easily move workloads between the different environments as needed. It can also help organizations to avoid vendor lock-in, as they are not dependent on a single cloud provider.

These early definitions mainly focus on the ownership of the infrastructure and the typology of users allowed to access provided services. However, they also highlight the first necessity of hybrid solutions putting the basis for a first decentralization of the cloud.

2.2 Models of Distribution of Cloud Continuum

The progressive diffusion of Cloud Computing technologies providing easy access to convenient, low-management, and scalable resources leads to the pervasive adoption of these technologies to support the ever-increasing demand and integration of IT services. The increasing reliance of many companies on the services offered by cloud computing has highlighted the importance of keeping these services always accessible and available. A single centralized cloud approach could represent a single point of failure for the overall architecture due to possible network connectivity problems, human errors, unpredictable failure, or natural disasters. From a business perspective, service **availability** is particularly crucial: despite vendors claiming the highest levels of service up-time, outages keep happening for any cloud provider in any geographic zone ranging from a few minutes blackout on a single data center to an entire geographic regions disruption for hours and even days [3]. With an ever-increasing reliance on Cloud services, these phenomena of unavailability can cause substantial financial loss and major disruptions.

The pervasive adoption of cloud services also highlighted the rising problem of vendor lock-in experimented by customers when the willingness to migrate from one cloud provider to another to follow cost logic or take advantage of different capabilities of the provider. A customer experiments with a lock-in when

the cost and effort associated with the migration from a service of a cloud provider to one of another provider are so high that makes the migration unfeasible or not cost-effective. This phenomenon can happen for a variety of reasons, such as operational cost to execute the migration or high developer effort to migrate from custom cloud provider API to the new ones.

2.2.1 Multi-Cloud

One first approach to address cloud availability and lock-in problems could be the employment of the *Multi-Cloud* pattern. Multi-cloud refers to the use of multiple cloud computing services from different providers in a single hybrid infrastructure. This can include using services from public cloud providers like Amazon Web Services[1], Microsoft Azure[2], and Google Cloud Platform[3], as well as private cloud and on-premises infrastructure. The Multi-Cloud approach can then be seen as a generalization of the original definition of hybrid cloud when considering the integration of multiple *public cloud* providers at once.

There are several reasons why an organization might choose to use a multi-cloud strategy. One of the main reasons is to avoid vendor lock-in, which occurs when a company becomes dependent on a single provider for their cloud services. By using multiple providers, an organization can ensure that they have a backup plan in case one provider experiences an outage or a change in pricing. Additionally, using multiple cloud providers can also help to optimize costs, as different providers may offer different prices for the same services. The integration of a multi-cloud strategy also allows organizations to take advantage of the unique

[1] Amazon Web Services (AWS) https://aws.amazon.com/en/
[2] Microsoft Azure https://azure.microsoft.com/en-us
[3] Google Cloud Platform (GCP) https://cloud.google.com/gcp

features and services offered by each provider. For example, one provider may offer better data storage options, while another may have more advanced machine learning services. By using multiple providers, an organization can choose the best option for each specific use case. The outcome is a more resilient unlocked-in multi-cloud ecosystem where not only the load can be subdivided on multiple providers but can also exploit some sort of geographical proximity to serve services from the nearest cloud region [4].

Implementing a multi-cloud strategy can be challenging, as it requires **coordination and management** across multiple platforms. This can be managed by using tools like cloud management platforms, which can provide a single point of control for multiple cloud environments, or with the integration and adoption of common standard cloud technologies widely spread across providers(OpenStack, Kubernetes) which can abstract the cloud providers layer.

While desirable, the granularity of these solutions is still too coarse to take advantage of strict locality and handle the huge amount of data and strict requirements imposed by the increasing number of devices connected and smart services requested.

2.2.2 Edge Computing

Edge Computing is a computing paradigm shifting from logically centralized cloud infrastructure to distributed and closer to customer computational resources.

The convergence and the increasing ubiquity of wireless Internet access and market penetration of Internet of Things (IoT) technology have given rise to the so-called Internet of Everything (IoE) era where billions of connected things create and exchange data through cloud data platforms. The pervasiveness of these

devices is opening new applications and development of ICT technologies to many sectors, such as smart cities applications [5], mobile gaming [6], cognitive assistance applications [7], and Industry 4.0 [8].

Centralized and remote solutions such as classical Cloud solutions hardly grasp on latency, bandwidth, security, and Connectivity availability needs that those innovative developments seek. An increasing number of industries, for example, are utilizing technology that demands fast analysis and reaction. However, cloud computing alone is not adequate for meeting these demands due to delays caused by the distance between the data source and the network, leading to inefficiency, delays, and unsatisfactory customer experiences. At the same time data continuously produced by those devices requires the availability of adequate bandwidth to be moved in order to be processed and arises many security and privacy concerns. Finally, the increasing dependency on cloud services for a rising number of processes and services makes the connectivity availability to those services a critical factor.

Fog and *Edge* Computing, commonly unified under the umbrella of Edge Computing, are two, relatively new, paradigms of computing that have been proposed to address these challenges. In Edge Computing storage and computing resources are placed **closer to devices** that produce and consume data or services [9]. Edge resources are **tailorable** on use case scenario-specific needs providing different functionalities, such as computing offloading, data storage, caching, or coordinating and redistributing service requests from clouds to users [10]. Moving data processing and analysis to the edge helps speed system response, enabling faster interactions thanks to the reduced number of network hops to be traversed in order to reach computational and storage resources. The reduced number of network

hops traversed joined with the development of new network connectivity technologies, such as WiFi-6 and 5G also promises to make available unprecedented network bandwidth supporting parallel and continuous transmission of requests and data.

The Edge resource could operate not only as a primary endpoint but also as a fallback in case of disruption of the availability of Cloud resources or connectivity to them, increasing in this way the reliability of the entire system. Finally, closeness to the producer of data opens new frontiers in privacy-preserving computation and aid in meeting regulation constraints in terms of privacy and security of users such as the ones described in the General Data Protection Regulation (GDPR) regulation [11].

The distribution of computational resources can span multiple tiers [12] from the data centers to end-users/devices depending on different scenario needs [8], [13] (Fig. 2).

The first tier is the edge computing offer developed by major public cloud providers, also called *Edge Cloud* for its closeness to traditional cloud computing offers. As a consequence of the rising demand for edge resources, many public cloud providers, such as Amazon, Microsoft Azure, or Google Cloud have differentiated their offers by disseminating smaller data centers in many regions by providing closer, with respect to the traditional offer, cloud resources still accessed through the internet.

A common second tier, commonly known also as *Near Edge*, is located between the core or regional data centers and the last-mile access [14]. This tier is commonly owned and operated by a telco provider or internet service provider providing access to the same edge resources to multiple customers at once.

The third deployment model is the *On-premises Edge*. This edge tier is located in the last mile access and can include edge nodes operated by enterprises (e.g., a retail store, a factory, a train) [15] or by customer-specialized edges, such as nodes located in a residential house in smart building scenarios [16] or in cars to support smart mobility.

Finally, the last Edge tier often called *Far Edge* [17] includes all those clustered and non-clustered systems directly connected to sensors/actuators via heterogeneous protocols, including non-internet protocols, and providing computational and storage resources.

2.2.3 Towards the Cloud Continuum

Depending on the specific need, each use case scenario could decide to adopt only one of the aforementioned cloud distribution models or more than one of them. Both multi-cloud and edge computing are emerging as promising architectural patterns, which could potentially accommodate the expected demands in terms of service availability and quality profiles. In fact, the development of an infrastructure that extends beyond centralized data centers, from the cloud to the edge toward the so-called *Cloud Continuum*, **combining** the large-scale data processing capability of cloud computing with the location-aware, geographically distributed, low latency data processing capability of edge computing is more and more expected an attractive solution direction.

From the integration of the different models in the cloud continuum many opportunities emerged but also significant research and technical challenges have to be faced in order to make this paradigm effective. Among those, the orchestration, organization, and management of these decentralized infrastructures are of partic-

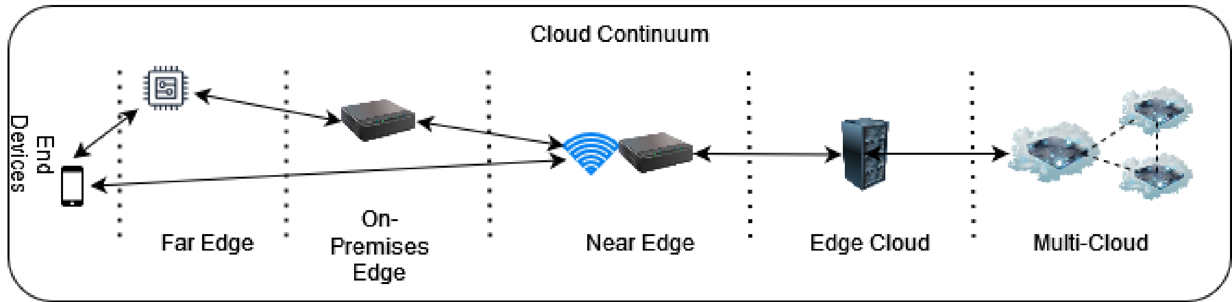


Figure 2: Cloud Continuum integrating computational resources available from multiple Cloud, Edges, and On-premises.

ular interest as constituting a key enabler for the effective adoption of this model. The integration of the many **heterogeneous technologies** and platforms available in the continuum of resources poses the primary problem of system integration, with systems in need of cooperating and coordinating through different protocols, and data formats. To effectively enable integration and cooperation among sites the urgency of making interaction among them reliable and scalable arises together with the need for novel mechanisms of synchronization and coordination.

The complexity resulting from such **distributed** and heterogeneous infrastructure can increase development costs and effort for new services and then hamper the adoption of the CC paradigm in many scenarios. New cloud computing models of services, such as serverless computing, abstracting completely where customer-defined logic is executed promise to mitigate this issue and lower the barrier to the adoption of CC infrastructure in many fields, such as Smart Cities, Smart Agriculture, or Tourism.

2.3 Cloud Service Models

At the foundation of the Cloud Computing concept is the offloading of responsibility for the creation and maintenance of an application or service to a third

party, the *cloud provider* as shown in Figure 3. From the early beginning of cloud computing developments, 4 models of services emerged differentiating cloud offers based on which of the management and configuration duties are offloaded from customers to cloud providers.

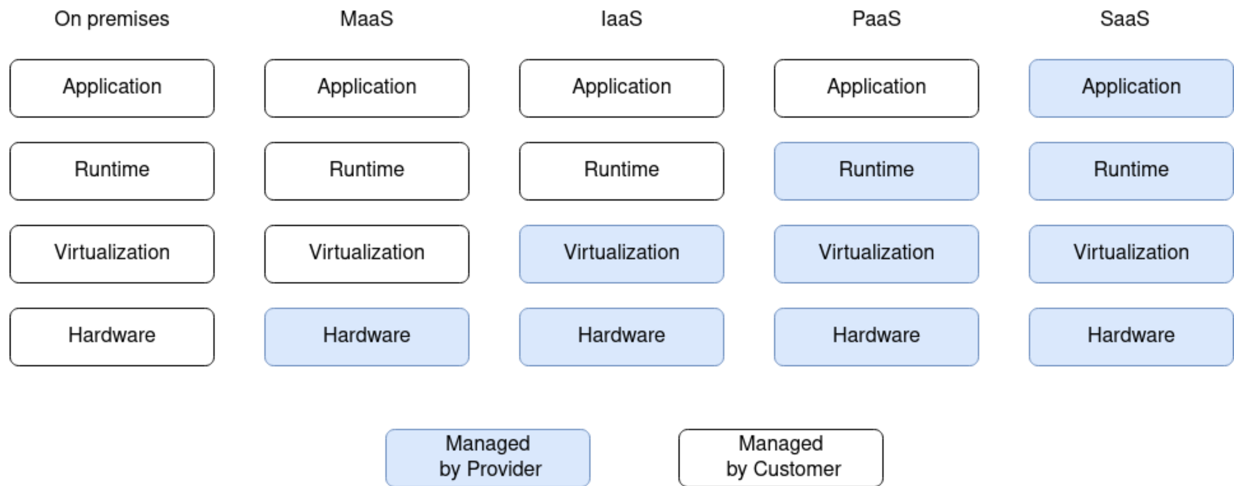


Figure 3: Levels managed by Provider or by Customers in the different Cloud Computing Service Models

2.3.1 MaaS

Moving from a classical on-premises installation where the customer has **full control** and responsibility on the entire stack, from installation and management of physical resources to the creation and accessibility of applications and services, the first model that we encounter is Metal as a Service (MaaS). MaaS is a type of cloud computing service providing customers with dedicated, bare-metal servers managed by providers. MaaS providers generally offer a range of physical servers with different hardware configurations, such as different CPU and memory options. Customers can choose the configuration meeting their needs and use the server for any purpose they choose. Even, network connectivity among nodes

while physically installed by the provider, its configuration, and maintenance are entrusted to the customer.

MaaS is a good choice for organizations that require high-performance, low-latency infrastructure and have their own team or a third party to manage all the infrastructural aspects. Thanks to the total control of the infrastructure, the customer is able to operate any optimization required while having the guarantee that applications belonging to other customers are run on the same infrastructure causing possible contentions on resources.

2.3.2 IaaS

Infrastructure as a Service (IaaS) is the cloud computing model that takes advantage of modern virtualization technologies abstracts and provides ready-to-use configured **virtual infrastructures** to customers. The IaaS offering includes a variety of services, including virtual machines, storage, and networking enabling the user to customize the characteristics and performance of the cloud-provided virtual infrastructure. Virtual machines, also known as instances, are one of the main building blocks of IaaS providing a way for customers to run their own operating systems and applications on virtualized hardware. This allows customers to have the flexibility and the responsibility to choose their own software and configure their environment accordingly to their needs. The pervasive use of virtualization technology in IaaS infrastructures revolutionized the market by providing the ability to cloud providers to abstract from physical resources and run workloads belonging to different customers on the same infrastructure transparently.

2.3.3 PaaS

The adoption of the IaaS model discharge customer to the duties of installing and maintaining physical infrastructures but still requires them to define infrastructure characteristics, configure single virtual components, and install all the necessary software and dependency to run developed services. The Platform as a Service (PaaS) further reduces customer efforts by providing a ready-to-use execution environment on which services and applications can run. One of the main benefits of PaaS is that it abstracts away many of the complexities of managing and scaling infrastructures, allowing developers to focus only on building their applications.

PaaS platform can aid customers in the entire development cycle by providing environments already configured for developing, testing, and deploying applications without the need to manage the underlying infrastructure and configure the framework. To further support and speed up development phases, PaaS infrastructures are usually juxtaposed by a series of services ready to use and already configured, such as databases, cognitive, and storage forming the so-called *Backend as a Service (BaaS)*. As per the PaaS application, also the use of those services is completely transparent to customers, with the execution and scalability of the service completely delegated to the Cloud Provider.

2.3.4 SaaS

Software as a Service (SaaS) is the highest cloud model of services allowing customers to access and use software over the internet, without the need to install or manage the software themselves. With SaaS, the software and its underlying infrastructure are totally managed by the cloud provider, who takes care

also of the **entire software lifecycle** from the development to the maintenance, updates, and security. Customers access the software typically through an interface such as a web browser or a mobile application, and only pay for the services they use, usually on a subscription-based model. SaaS is commonly used for a wide range of business applications, such as Customer Relationship Management (CRM),Enterprise Resource Planning (ERP), human resources management, and collaboration or productivity tools like Office 365, Google Suite, or Salesforce.

One of the main benefits of SaaS is that it eliminates the need for customers to purchase, install, and maintain software, and reduces IT management costs. It also enables the customer to access the software from any device with internet access while still leveraging on rapid scalability of cloud infrastructures. SaaS is becoming more and more popular due to its cost-effectiveness, ease of use, accessibility, and scalability. As long as the customer has a stable internet connection, they can access the software and the provider takes care of the rest.

2.3.5 Everything as a Service (XaaS)

Starting from these four models the cloud providers differentiated service model offers over time in order to create a more pervasive and complete offer suiting in a finer-grained way customer needs. The progressive subdivision into specific services such as Storage as a Service and Domain Name System (DNS) as a service as well as the integration of new technologies such as containers with its relative model of service Container as a Service or the creation of ready-to-use services such as the Database as a Service led to the definition of the so call *Everything as a Service (XaaS)*. With the term XaaS, we indicate the continuous development of cloud service offer in order to integrate new technologies and ex-

tend cloud computing capabilities. One of the branches of this differentiation, in particular, aims to progressively discharge customers from management, configuration, and orchestration effort in order to concentrate only on the business logic of the application or service.

An emerging model of services that is gaining more and more interest in the market is the Function as a Service (FaaS) model. In the FaaS model customer is in charge only to create and upload the code representing the business logic desired and associate it with the triggering of an event. The infrastructure then takes care of executing, in a completely transparent way for the customer, the code at each triggering of that particular event.

In the following chapter, we will focus on this model, its capabilities, and constituting elements to provide essential background in order to present our innovative solution based on the FaaS model for the integration of services and data in distributed and heterogeneous scenarios.

3 Function as a Service (FaaS)

Function as a Service [18] is a novel cloud computing model in which customer-defined logic is put into execution dynamically when triggered by an event. The abstractions employed by the FaaS platforms give to customers the illusion of the **absence of the entire infrastructural layer** making this model part of the so-called *Serverlles* Computing model family. The customer-defined logic is represented in this platform by the *function* which represents the atomic unit of execution. The association between the event and the corresponding function that is executed at its reception is called *FaaS Workflow* and is configured by the customer after the upload of the code representing the function. The relation that can be settled between events and functions is many-to-many with a single event that can trigger the execution of multiple functions as well as multiple events that are processed by the same function.

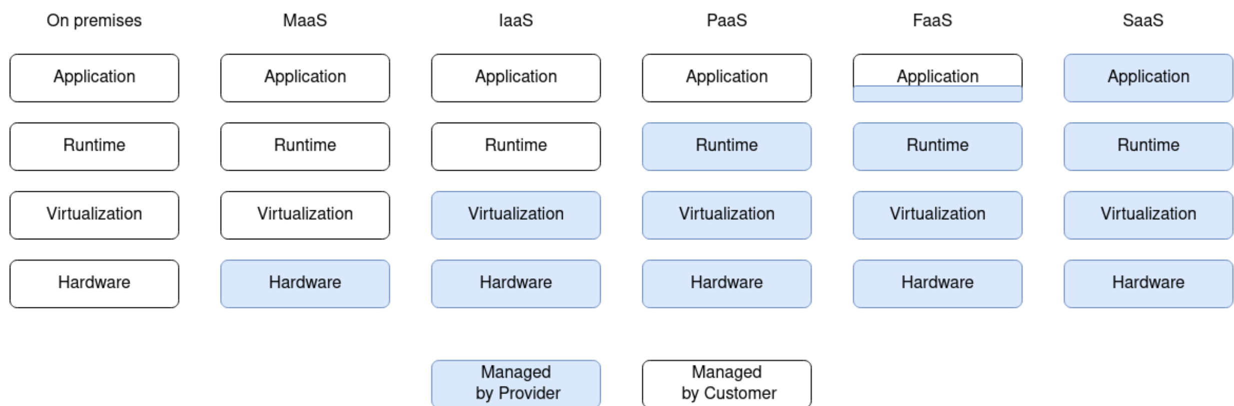


Figure 4: Cloud Computing models of service with FaaS as an intermediate offer between PaaS and SaaS.

In the FaaS paradigm, customers have no control over where and exactly when their code is executed. In these platforms, in fact, the entire stack on which the user-provided code is executed, from the hardware to the execution runtime, is

provided **already configured and managed** by cloud stakeholders (Fig. 4). The client is only in charge of the creation and uploads in the platform of the function code and the configuration of the workflow. The customer has very little control not only over the configuration of the execution environment but also over the computational resources available for the execution. From a customer perspective, in fact, FaaS functions are **ephemeral** entities whose lifecycle is bounded by the processing of a single activation event.

The support to a specific programming language for the creation of the function is in charge of the infrastructure provider, even if most of the public cloud providers support an ever-increasing number of programming languages including Java, Python, JavaScript, etc. The execution environment in which the code is executed is defined and provided by the FaaS cloud provider and gives to customer a **low degree** of freedom for customization. That limits the freedom of the developer but enforces the control of the Cloud Provider facilitating optimizations and management operations such as runtime updates or security fixes.

With respect to the Platform as a Service model, the FaaS model differs especially for a partial loss of control of Application layer aspects (Fig. 4). In FaaS platforms, in fact, developers are not requested to implement protocols like HTTP, TCP, or RabbitMQ to interact with the functions deployed.

The different ingress points to trigger the function are already provided by the platforms and associated with the function workflow configuration. That behavior creates a **separation** between the business logic of the function and the logic to interact with them which encourages code reusability and fastens the development of new services. Moreover, as already stated, the user has no control over the

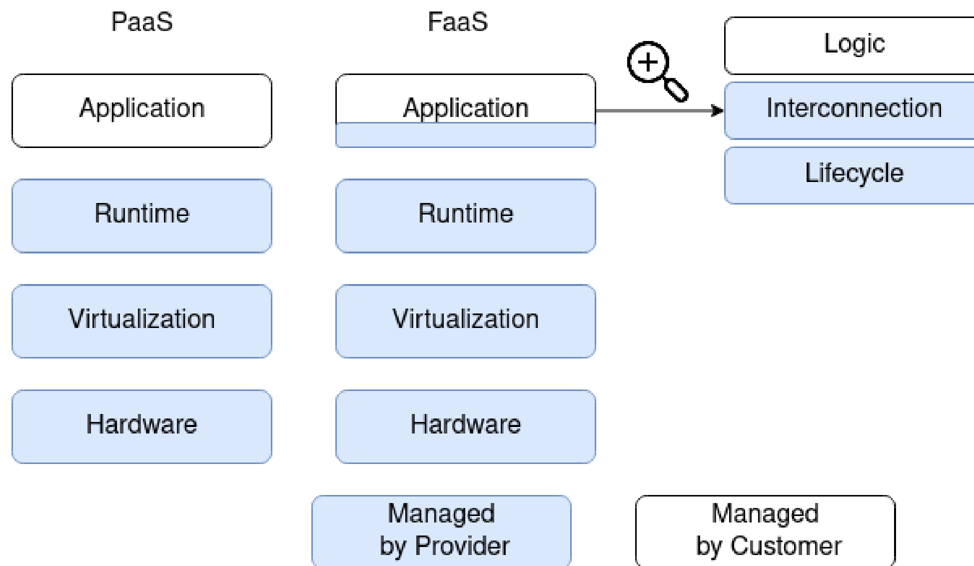


Figure 5: Zoom-in showing in details differences among PaaS and FaaS models in terms of duties managed by provider or customer.

lifecycle of the functions which are instantiated dynamically at the arrival of each event.

FaaS platforms are so characterized by fine-grained per-request scaling of resources: at any time the number of resources employed by the FaaS platform automatically tends to be proportional to the number of requests issued. This scaling mechanism also leads to *Zero-scaling* capability, which allows services and applications not in use to consume almost no resources. The zero-scaling features in public cloud provider FaaS platforms introduced the **per-request charge** where the customer pays only for a single activation of functions deployed.

3.1 Architecture composition

In the past years, many FaaS architectures have been proposed by both industry and academia, targeting specific needs or challenges. From the analysis of these proposals and main Open Source and private FaaS solutions three

main architecture elements embodying essential behavior realizing FaaS platforms emerged: the *Trigger* acting as a bridge between the platform and external world, *Controller* managing infrastructural aspects and workflows activation and execution, and the *Function Executor* executing code representing functions (Fig. 6).

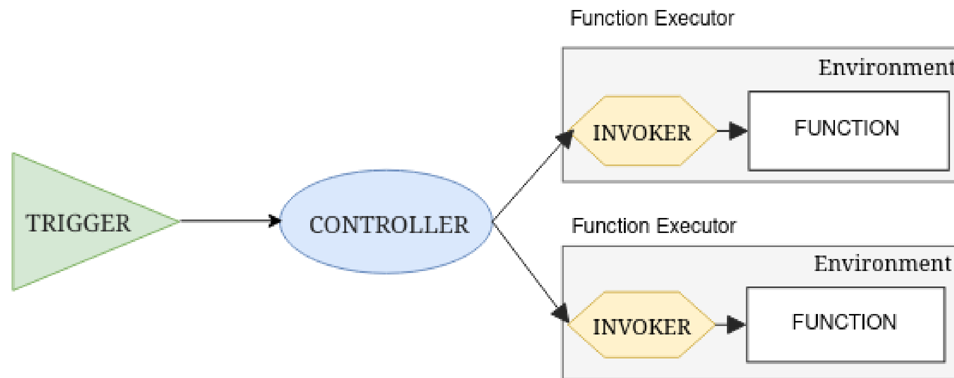


Figure 6: High-level FaaS architecture components.

The *Trigger* is the logical entity responsible for receiving or sensing external information and converting them into internal events *triggering* functions execution. Triggers typically receive requests from heterogeneous sources and the interaction with them can employ different protocols and formats.

Depending on the nature of the source, the trigger embodies different forms of interaction spanning from active ones with the trigger initiating requests or passive ones by listening for incoming connections. The lifecycle, management, and scaling of the trigger component are completely demanded of the FaaS platform which can take advantage of the intrinsic **stateless** nature of this component.

The *Controller* works at both the *Infrastructural* and *Application* layers (Fig. 7), managing infrastructural elements of the platform and coordinating the execution of function workflows. The main objective of the controller is to monitor and coordinate these two layers in order to apply new customer configurations and maintain workflows responsive thanks also to the metrics coming from a pervasive

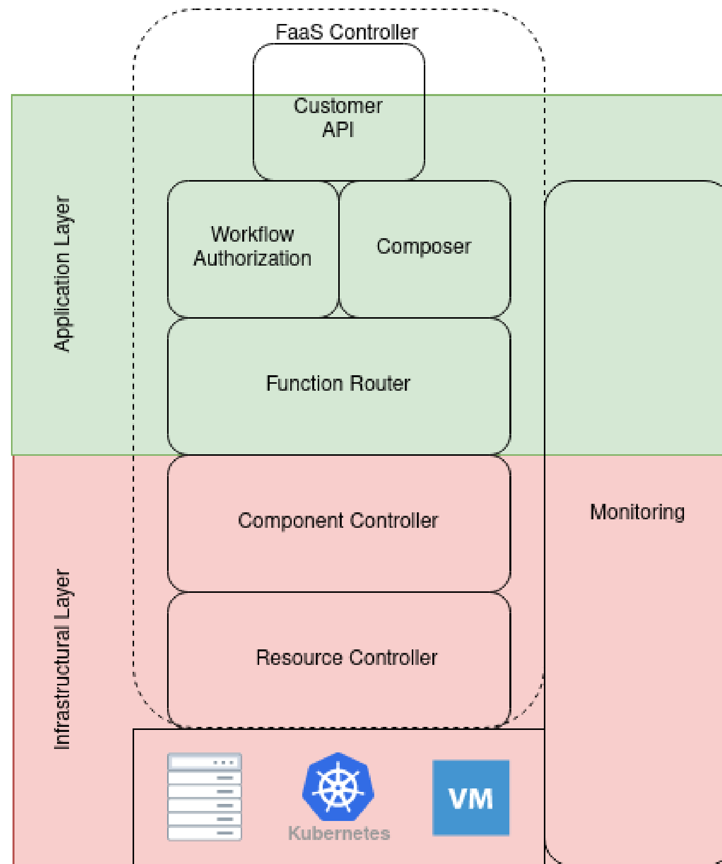


Figure 7: High-level decomposition of a FaaS controller into its constituting component addressing different orchestration and management duties affecting infrastructural and application layers.

Monitoring framework traversing layers of the architecture. In FaaS platforms, in fact, Monitoring capabilities are provided and configured by the platform across the entire stack of the platform from the Application Layer to the Hardware one. This pervasive monitoring gives the Controller essential information to take informed decisions in order to maintain the platform responsive.

From a Top-Down perspective, the Customer interacts with the *Configuration API* by submitting to it Workflows and functions. Configurations received on these layers are memorized and propagated to other components in order to activate the Workflow on the platform. The Configuration API Controller can im-

plement and expose different protocols to receive configurations, like HTTP or gRPC as well exploit different configuration formats, such as JSON, TOML or XML.

The rights of a particular user to access the resources described in the workflow configuration are checked by the *Authorization Controller*. This Component can be also involved in the process of workflow execution verifying the rights of a user or service to trigger the execution of a specific function workflow.

The *Function Router* is the controller component that decides which Function Executor redirects each event incoming from the Trigger. This is the component that realizes the logic described in each workflow and distributes the computational load among the nodes of the cluster. Is the Function Router that realizes the logic of **load balancing** and **fault tolerance** in the process of event distribution and triggers actions of scaling.

For event distribution, the Function Router can interact with other components either realizing synchronous service call invocation (Fig. 6) or exploiting asynchronous pub-sub-based communication (Fig. 8). The former requires that each received event is first passed to the controller and processed there to decide the next hop in the invocation chain. In the latter, the controller typically exploits pub-sub *Message Oriented Middleware (MOM)* to deliver events to nodes. The integration of a MOM encourages a more asynchronous and decoupled interaction between the architecture component facilitating their distribution and management. Moreover, the function router can exploit built-in capabilities for message delivery of each MOM solution adopted, such as differentiation in QoS, delivery semantic, or advanced load balancing.

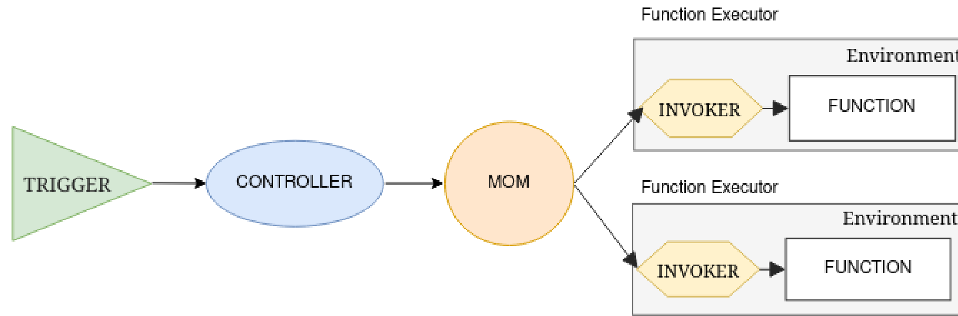


Figure 8: FaaS architecture introducing a MOM as a medium for event delivery to Function Executor nodes.

The *Component Controller* (Fig. 7) is the process in charge of provisioning and configuring architecture components of the platform. It is this component that, having received a workflow configuration and code, configures the infrastructure component to activate the workflow. For example, this controller packages the code representing the function and deploys it to execution nodes, or sends the configuration to an HTTP Trigger to make it accept requests at a specific path.

The provision of computational resources needed to deploy new components or execute functions is demanded to the *Resource Controller*. Depending on the specific deployment environment the provisioning of resources can be directly done by the Resource Controller, e.g. execute the command to scale a specific service on the local node, or by an external cluster controller. In the latter case, the Resource Controller acts as an adapter to specific cluster solutions (e.g. Kubernetes, OpenStack), on which the serverless platform is deployed, by requesting through the specific API the computational resources needed and providing abstraction to the Component Controller to provision the platform component on it.

Function Executor are agents installed on computational resources provisioned by the Resource Controller and usually leveraging on multiple distributed

virtual physical nodes and are in charge of hosting and putting into execution functions for each incoming event. Nodes hosting this process, either virtual or physical, are then called Function Executor Nodes or in short *Executor Nodes*.

Functions are executable codes expressed in one of the supported languages, and uploaded by customers ahead of time in conjunction with a configuration. The configuration is the set of information exploited by the FaaS platform to know to which external events the execution of a function is associated with and in which execution environment should be run. The *execution environment* does not only specify the language framework but also all the possible dependencies, the targeted operating system, and the architecture on which the uploaded function code has to be put in execution. In particular, the process responsible for waiting for events and for starting function invocation is the so-called *Invoker* or *Watchdog* [19]. The distribution and architecture of the invoker can follow three main architecture models namely: (i) invoker per function creating an invoker instance for each function deployment, (ii) invoker per node instantiating a single invoker instance in each node either physical or virtual in the cluster, and (iii) invoker per cluster providing the invoker as a service deployed on the cluster (Fig. 9).

In the *invoker per function* model, adopted by platforms such as OpenFaaS [19], each invoker instance is associated with exactly one function. The resulting invoker is a lightweight process with minimal impact on resources, and this pairing of function-invoker simplifies the management and orchestration plane, at the cost of greater resource utilization. Moreover, the absence of coordination among invokers located on the same node could inhibit optimizations, such as shared resource access or computational resource reservation.

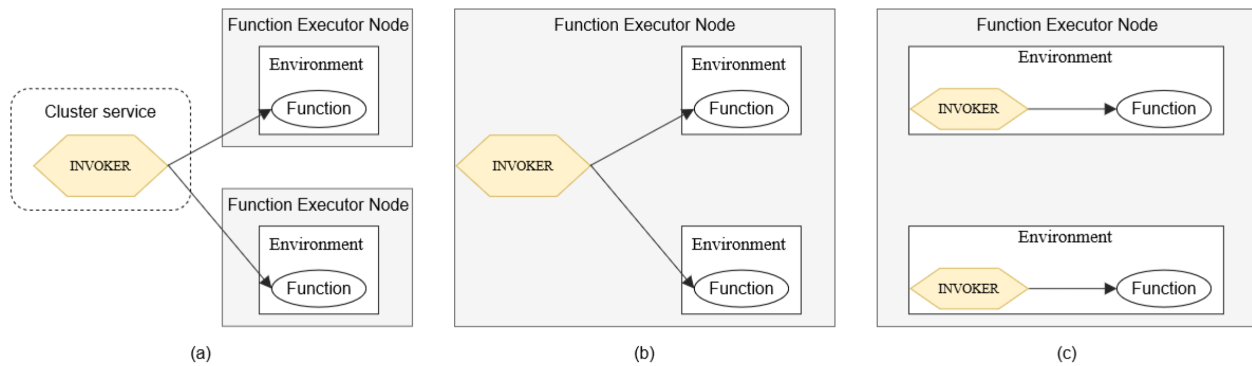


Figure 9: High-level FaaS Invoker architecture approaches: (a) per cluster, (b) per node, and (c) per function, respectively.

In the *invoker per node* model, adopted by platforms such as OpenWhisk [20], each executor node hosts exactly one invoker component, responsible for receiving the different events from the controller and through a multiplexing mechanism invokes the associated function. This model has the advantage of reducing the number of invoker instances running concurrently on the same host with respect to the per-function approach, thus potentially contributing to better resource utilization. However, this comes at a cost of a more complex control logic handling request multiplexing to the invokers, with a consequent increase in complexity in orchestration and management.

In the *invoker per cluster* the invoker functionality is *de facto* offered as a centralized service. This model has the advantage of having the smallest possible impact on cluster resources, simplifying event distribution as all the events are routed to the centralized invoker services. This model forces a **strong coupling** with the specific orchestration technologies and protocols on which the FaaS platform is deployed e.g., Kubernetes, Apache Mesos, OpenStack, etc. Moreover, the communications between the invoker and functions could result in overall slower

execution times due to the centralization of the service and the introduction of network-based communications.

From the above considerations, the invoker per node model emerged for its advantageous trade-offs in terms of complexity, resource consumption, and performance. However, since many FaaS platforms are built on top of already consolidated resource management solutions and orchestration platforms, the choice of a model of deployment for the invoker should take into account the peculiarities and constraints of the running environment and not only business or performance logic. In particular, some solutions can inhibit resource management from third processes. In these cases, the adoption of the simplest and thinner solutions (invoker per node) or more integrated ones (invoker per cluster) can represent the more practical direction.

3.2 Function Composition

An appealing feature available on a rising number of platforms is the capability of combining the logic of multiple different functions into a single workflow. This feature, called *function composition*, not only further simplifies the development of new services and encourages greater modularity and reusability of code but also facilitates a greater **parallelization and distribution** of workflow business logic.

A particular type of Composition widely adopted in FaaS platforms is Function Chaining, a pattern inherited from functional programming, where the output resulting from the execution of a function is piped as ingress of another function, thus creating complex processing from the composition of a simple function. An

efficient mechanism of chain can encourage a fine-grained decomposition of logical functions in small actual functions, enhancing code modularity and reusability.

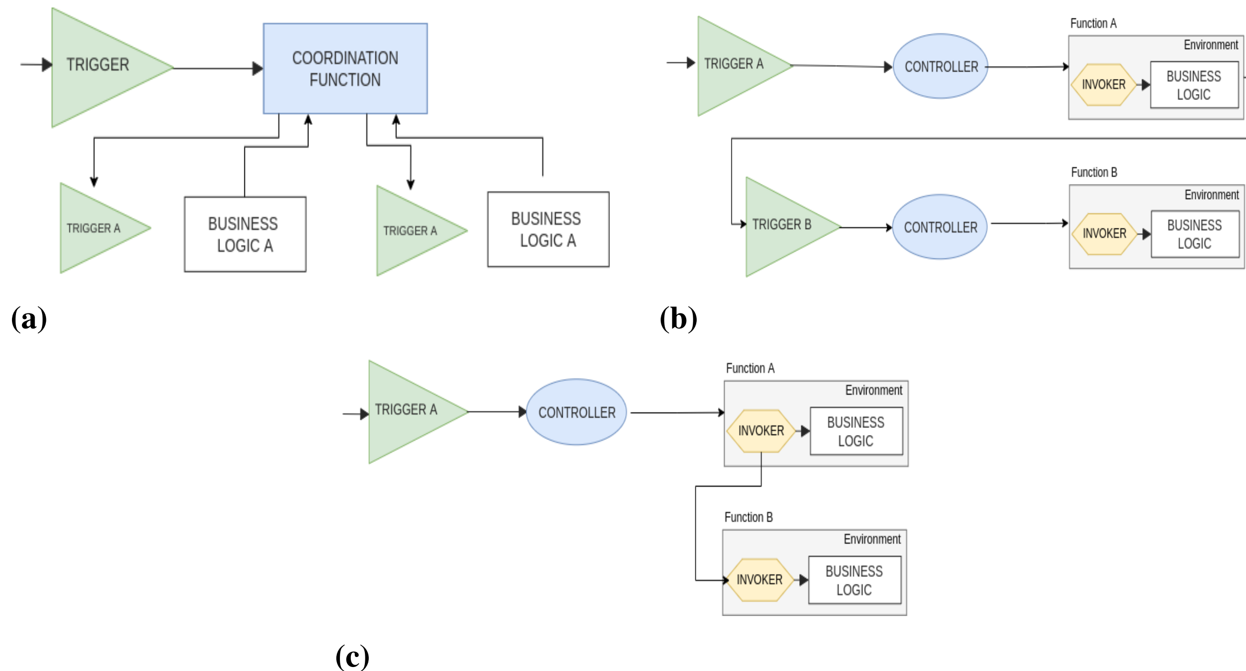


Figure 10: Function composition approaches. (a) Reflective invocation: a third entity coordinates the function invocation and forwarding of the result to the successive function in the chain. (b) Continuous Passing at the Business Layer: the business code directly invokes the next function in the chain through the associated trigger (c) Continuous Passing at the Infrastructural layer: the invoker is tasked to forward the output of the business logic to the next function in the chain.

From the literature, two main patterns emerged for implementing the composition logic in FaaS. The *reflective invocation* pattern relies on the use of a third entity to encapsulate the logic of function composition. The entity tasked with the function execution, waits for its result, forwarding it to the next function in the chain until all the processing units have been executed (Fig: 10a). In the *continuous passing* composition pattern, instead, the function can locate and name the

next function in the pipeline, invoking it by forwarding the necessary input (Fig 10b) [21].

FaaS platforms implement these two patterns either at the business layer or as a built-in capability at the infrastructural layer with advantages and disadvantages in both choices. The implementation at the *business layer* foresees that each function has at least one trigger associated, exposed to the outside, and addressable from other functions. In this setting, it is the responsibility of the developer to create both the logic of the invocation and implement the protocols needed to allow the composition of and message passing between functions (Figure 10b). While this approach can offer the best expressiveness and dynamicity, it also hinders function re-usability and modularity as the business logic is bound to a predetermined composition policy.

On the contrary, in the *infrastructure layer* approach, the business logic of each function is decoupled from the composition logic. In this case, upon function completion, it is up to the FaaS platform to decide whether and where to redirect the output, as well as the actual protocol used to forward it (Fig 10c). In this approach, there is a separation between policy and mechanism, leading to easier implementation and overall better performance. Moreover, some optimizations can be exploited by the infrastructure to improve the processing pipeline performance, e.g., function co-location, result caching, or optimized function-to-function communication protocols.

To conclude in the adoption of a particular approach for function composition many trade-offs emerge. In particular, the adoption of approaches based on third-entities, requiring often some form of **states** to execute, while providing a more complete and customizable tool for developers, can introduce constraints in

terms of scalability, performance, and availability of the mechanisms. Instead, approaches like continuous passing relying on asynchronous communication and the absence of state, guarantee better scalability and reliability of the system at the expense of mechanisms expressiveness. The integration of function composition at the infrastructural layer allows to exploit peculiarities of each platform providing advantages, such as performance, fault tolerance, and less resource consumption but they introduce a stronger coupling, with respect to solutions that rely on application-level coordinator, between the platform and the composition solution.

3.3 Cloud Continuum enabled FaaS platform

While the FaaS paradigm was originally conceived as a model primarily executed in centralized Cloud environments to exploit the huge availability of unused computational resources of modern data centers, it rapidly evolved to support deploy scenarios over the CC [22], [23]. The integration of these two technologies, in fact, promises to make available through abstractions of the FaaS model, performance improvements derived from the simultaneous exploitation of heterogeneous and distributed resources available on the Cloud Continuum [1]. The FaaS model can cover a wider spectrum of applications by exploiting at the same time virtually unlimited resources of Centralized Clouds and low latency and high throughput performance of Edge Computing.

In particular, in the Edge scenario, the FaaS paradigm providing fine-grained scaling and zero-scaling capabilities can provide huge advantages to those deployments with a high density of differentiated applications or with relatively limited resources such as for edge cloud nodes.

We believe that the research and development of standards, optimizations, and technologies enabling the efficient execution of FaaS function across resources of Cloud Continuum could benefit and accelerate the development of many fields such as Industry 5.0, Smart Cities, or smart mobility. Those scenarios, in fact, are often characterized by a strong dynamicity and heterogeneity in terms of offer and requirement. The application of the performance tailoring capability of the CC joint with fast and facilitated development of new services of FaaS can support in coping with the dynamicity of sectors.

4 Service and Data Integration in Highly Distributed Scenarios

Recent development in Cloud Computing towards the cloud continuum enabled unprecedented performance, low latency, and bandwidth availability. Many sectors, such as manufacturing, tourism, city management, and logistic operator are exploiting these novel capacities as a boost to the development of new distributed services. In fact, the availability of distributed computational resources combined with new network technology, such as 5G or WiFi6 is enabling the in-time processing of data coming from a rising number of devices connected.

The number of connected devices and sensors available in the territory and inside factories is constantly rising providing continuously update information. Cities administrations can then exploit sensors to acquire information, such as crowding, and noise to plan more targeted interventions. Manufacturers can leverage information coming from the Industrial Internet of Things and connected vehicles to create a more integrated supply chain with production reducing in this way delivery time and costs.

At the same time, new Service Models, such as PaaS or FaaS are opening the development and hosting of services and applications to a wider audience thanks to reduced deployment and management efforts. This unprecedented number of sources of information, partners, services, and device connected opens new forms of collaboration, optimizing processes and creating more involving and unified experiences for end-users. These integration, while appealing and expected to create great value for manufacturers, customers and communities are hampered by the heterogeneity in typology, representation, and protocol with which these are made

and exposed. This scenario is further complicated by the ever-increasing number of technology consuming this information and services and presenting them to end users in different ways, such as mobile devices, Virtual Reality headsets, and wearables.

In this context, the development of **integration architectures** emerged as a frequent pattern adopted by different industries and realities to respond to the exigence of interpolating information coming from different sources or combining services. These solutions however are characterized by a cost proportional to the number of integration as a consequence of the high heterogeneity of sources and requirements over the costs of management and orchestration. This often results in vertical integration with low interoperability among them and waiving on integrating all those sources are not cost-effective. The creation, promotion, and adoption of standards among partners could partially alleviate this problem by reducing heterogeneity. However, especially when considering integration among partners and data coming from different sectors the settling of a common set of standards satisfying the needs of all partners is often impracticable.

4.1 Smart Tourism Service and Data integration

The tourism field thanks to its intrinsic distribution and the natural presence of multiple heterogeneous actors shaping this context qualifies as one of our main reference scenarios during this study.

The rapid and pervasive evolution of digitalization covering all aspects of life has drastically changed the field of tourism [24] so to propose a new pervasive experience, more and more based on online services and information; moreover, that trend is expected to further grow in acceptance and offerings. In fact, we already

see a wide variety of services, datasets, and platforms concerning and supporting tourism in many of these new different forms. In the last decades, tourism acquired a key role in the development and economic growth of many countries, and so it is in Italy. As stated by Eurostat [25], in the European Union (EU) area tourism is the EU third largest socioeconomic activity, representing around 10% of the EU GDP (Gross Domestic Product). Moreover, five EU Member States are among the world's top ten tourist destinations worldwide.

In Italy, more significantly than in other countries, tourism is in continuous growth and represents a vital contribution to the wealth of the nation. In fact, the total contribution of tourism to the Italian economy in 2017 was 223.2 billion euros, equal to 13% of Italian GDP, and Italy was ranked the second destination for outbound trips made by EU residents within the EU, in terms of nights spent [25].

The fast increase in the tourism market is raising the need for "Smarter" Tourism (Smart Tourism, or ST for short) more able to personalize and adapt customer experiences while creating a more culturally rich and even more sustainable offer. To evidence the importance of ST in the sustainable development of a country, the European Commission launched the European Capital of Smart Tourism to stimulate the development and sharing of ST good practices. The European Capital of ST is an initiative to promote integrated offers and innovative, inclusive, culturally diverse, and sustainable practices for tourism development by European cities [26]. Within that project, the European Commission has defined the concept of Smart Tourism as the combination of properties:

- **Accessibility**: to enable barrier-free destinations and enable access, regardless of age, cultural background, and physical disability.

- **Sustainability**: to protect natural resources of a city, reduce seasonality, and include local communities.
- **Digitalization**: to use digital technologies to enhance all aspects of the whole tourism experience.
- **Cultural heritage and creativity**: to protect and capitalize on the local heritage for the benefit of all stakeholders: the destination actors, the industries, and tourists.

The always enlarging availability of information accessible through the network has modified the approach of visitors to the experience from an even structured and well-planned tourism offering to more dynamic and by-need ones. The development of ST will require not only a more personalized experience for tourists but a more dynamic service proposal as key factors of the whole experience. Examples of dynamicity are modern apps that exploit geo-localization to retrieve more suitable local services in the locality of tourists and in a by-need fashion. That dynamicity further promotes a more personalized experience by using an intelligent system recommender: today the whole information about previous historical data and profiling plays a key role in ST decision-making processes [27]. Moreover, in the last years, the recent global pandemic has stressed further the importance of smart and dynamic tourism services based on geographic positions [28].

Another important accelerator to drive information retrieval toward service quality is played by the pervasive diffusion of IoT and smart devices, connected with Social Sensing. In Social Sensing, the final customer can be actively and deeply involved in many ways, from contributing with her knowledge and sharing

data gathered with smartphone and personal wearables, to asking her to complete simple tasks while moving with her phone. Social Sensing extends an already widespread and well-established series of techniques called crowdsensing [29] already proposed to involve users in the process of data gathering [30]. Since initiatives of social sensing, crowdsensing, and crowdsourcing can play an essential role in the development of smarter tourism services, those initiatives are also coupled with incentivizing user participation via some forms of competition among users and via strategies of gamification.

Tourism gamification extends strategies from game design and involvement strategies in non-game contexts [31], so as to influence consumer engagement, customer loyalty, brand awareness, and user experience in tourism areas [32]. Examples of these initiatives include as an example the usage of a scoring system related to customer action undertaken also rewarded with forms digital or material incentives and rewards. As an example, in recent years many studies have proposed the use of Social Sensing to collect geo-tagged information and exploit them to identify Tourism Areas of Interest [33], map tourist behaviors [34], [35], compare and differentiate clusters of tourists [36], and discover and repropose noteworthy new places [37].

From a user perspective, the employment of ST technologies in combination with techniques of social sensing can achieve better information feeding to the tourists about the quality and accessibility of a place, depending on their specific interests, either long-term or defined on the spot, from the presence of barriers to the support of different languages in the service, depending on the current weather situation to the current mood of the entire group. In the city of Bologna, for example, there is an application to cancel barriers in both access to services and in

mobility, as demonstrated by projects like mPASS [38] or Kimap [39]. Additionally, we must add that the tourism area itself asks for deep integration with many other fields, such as Smart Cities [40], Smart Transport, Smart Wealth, and relative services and data sources as a few examples of connected areas.

To summarize, the integration and combination of ST information and services are expected to create great business value and enable the development of smarter tour-ism services and experiences, but unfortunately, the heterogeneity of formats and inter-actions protocols slow down the integration of multiple platforms and make difficult the creation and recognition of a unique and comprehensive standard, also because of the lack of regulation and the different stakeholders and organizations proposing ST services. As a clarifying example, one of the most challenging scenarios for its wide geographical distribution in the context of Smart Tourism is the business of Tourism paths (or ways or itineraries, sometimes pilgrim's ways), typically established very long ago to suggest routes to religious pilgrims(e.g. the Francigena way in Italy) in the Middle Age and to give advice in their ways toward their final destination. This novel and more re-quested type of tourism offers are characterized by the requirement of extreme personalization and dynamicity of target user experience; tourists can choose via the information given by their smart devices how to continue the experience, being always driven by current information got from their connection on demand. Of course, the same always-connected feature can be used by many other areas apart from ST, so presented architecture can be crucial in those too.

In Tourism paths, users intend to use ICT as an essential part of the experience and tend to be driven by the information they need to get either personally or as part of a group, and to feed information over the community depending on their

current experience. We consider that tourists can become prosumers (consumers and producers at the same time) of the experience of Tourism Paths. As an example, via ICT tools, the user can interact more dynamically and satisfactorily, by choosing to read personalized paths and calibrating languages and contents based on specific levels of learning [41]. It becomes essential to gather as much information as possible, so as to provide customers with the necessary details and to provide the customers with the best experience possible. That high dynamicity constitutes a challenge for providers of smart tourism services characterized by huge distances to be covered with a multitude of information to be gathered, with a high fluctuation in the number of users requesting those services, and with an important level of heterogeneity in partners to be involved.

4.2 APERTO5.0

To support a more open integration among the different partners of these scenarios we proposed APERTO5.0 (an Architecture for Personalization and Elaboration of services and data to Reshape Tourism Offers 5.0), in shorter APERTO, a layered integration architecture aiming at a whole integration and deep facilitation of service and information organization and blending, to enable the re-provisioning of novel services as advanced aggregates or re-elaborated ones [42]. APERTO5.0 was originally conceived to specifically support the field of Tourism but we claim the applicability of the same principles and technologies to many other fields, such as smart cities or Industry 5.0 seeking more open and dynamic cooperation with partners, customers, and devices cooperating in the creation of these extended business value chains.

APERTO is based on an organization that puts together, on the one hand, all possible information sources, and services toward better integration, on the other hand, the best proposition possible for the differentiated needs of all customers, either single or in different composition groups in number and interests. The proposed architecture has been designed driven in the middle of existing services and information providers and the possible requests and needs of customers (Figure 11). APERTO5.0 aims at becoming the reference for the development of new more innovative services and platforms while adding value both for Producers and Consumers based on its integration and augmentation capacity. In the context of APERTO5.0 architecture, we defined a Producer as every entity that provides information or services with a potential appeal, by including partners belonging to the public and private sector, open data, and any connected things spread in the interested region, like connected Transports, sensors, and user wearable devices. On the other side, Customers are users of the platform that can consume services and data resulting from the processes of augmentation, elaboration, and orchestration of information and services coming from Providers. We must stress that a Provider can also play the role of Customer and vice-versa, by creating a circular **Prod-Cons** pattern, where providers can interact with the proposed platform as customers to grow their services, and customers can improve their experience through personal contributions to APERTO5.0. This positive evolution can be opportunistically encouraged through initiatives, such as crowdsourcing campaigns and the creation of local relationship networks. We claim that the introduction of the proposed digital platform can encourage the creation of a network of partners that can also increase, monitor, and guarantee the value of data and services. These types of relationships can also foster the development of opportunities by

creating a mutual value such as the creation of an agreement of multiple actors interested in the development of a service that can benefit in different ways all the partners. The Network constituted the different partners in the territory supported by the proposed digital platform constitutes the target supply chain for customers.

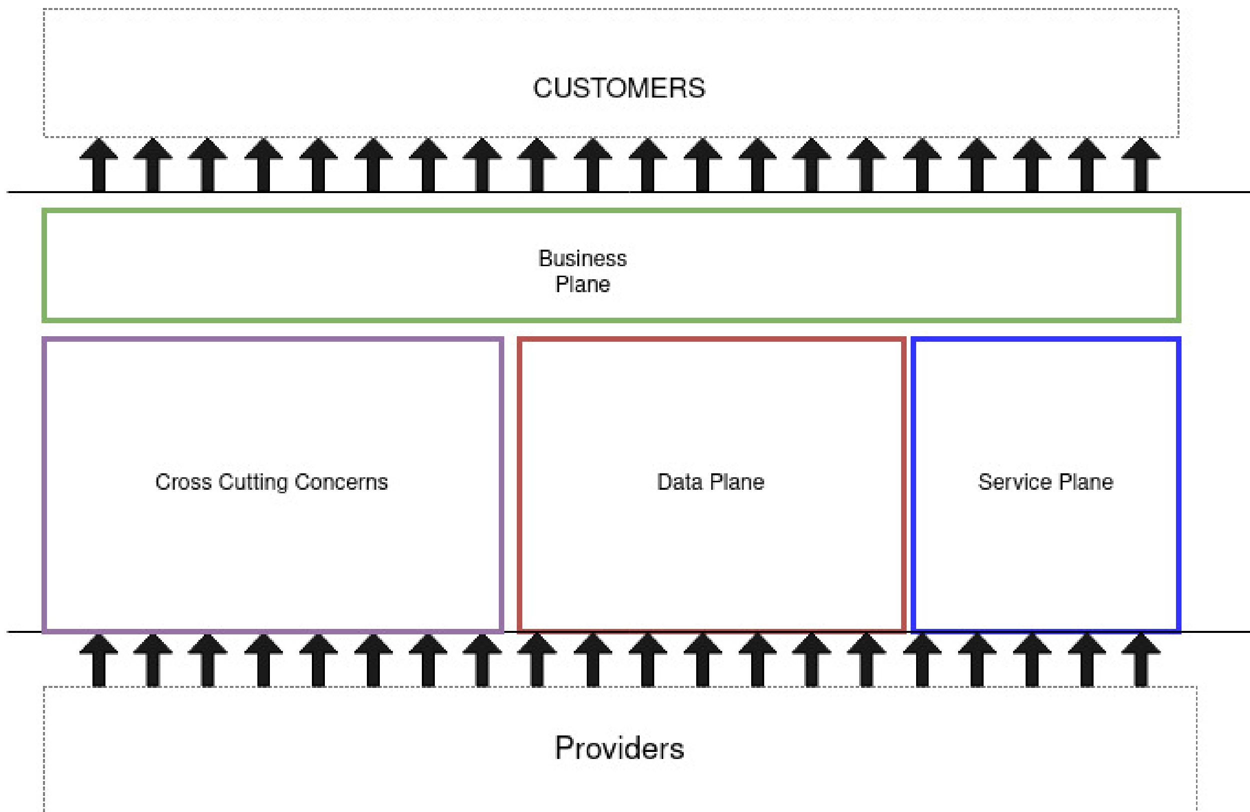


Figure 11: High-level vision of the APERTO5.0 architecture in terms of the layers connecting Providers to Customers.

The architecture of APERTO is based on three well-defined layers called planes: the higher-layer business plane is responsible for customer interaction; the other two lower-layer planes are responsible for all possible services (service plane) and information (data plane) arriving from interested providers. It is important to stress that APERTO5.0 can expose new services based on the available existing ones; another lower layer component is the crosscutting one, in charge of all managing and monitoring functions of the entire architecture.

We are now expanding the details for the above planes. The *Business plane* is the functional plane that addresses the complexity derived from interactions with customers with the main goal of providing a unique point of interaction, and, at the same time, hiding from the customers the complexity of distributed datasets and services. The Business plane has the main goal of uniforming access to the heterogeneity of services, protocols, and interactions arriving from the underline planes to compose a solution offer. This plane drives the composition, coordination, orchestration, and exposition of services and information coming from the data plane and the service plane. The Business Plane is capable of creating new synthetic proposals, starting from existing services and information. An example of a synthetic service in the context of the digital economy can be the commissioning of the creation of a product and the simultaneous search and booking of the delivery service. These features demand a high level of modularity and composability to adapt to the continuously evolving needs and interaction methods of customers, via tools like Dashboards, Apps, and APIs.

The *Data Plane* is the component of APERTO5.0 responsible for collecting, managing, and analyzing all the datasets and information collected from third parties-providers, realizing the augmentation and conformation of data. This process is an essential step in the creation of new services and platforms so that uniformity and standardization can reduce considerably the effort related to the management of different formats. The Data Plan can handle and process both data in motion and at rest (very static and very dynamic data, as extremes), enabling the exploitation of both historical data and fresh real-time information. To achieve good value from data both stored and processed, it is necessary the use **blending techniques**, over the data coming from multiple and diverse sources to merge

them and consolidate a network of partnerships in the territory that can provide feedback and support, so as to specifically verify the information. The Data Plane supports multiple types of representation and analytics, in order to handle the different needs of customers, including geographical and time-based queries, up to computationally intensive processing like graph algorithms and machine learning techniques.

These Planes, part of the proposed architecture, enable the representation, integration, and orchestration of the provider's services. Since the proposed solution does not replace or force migration of existing services but, on the contrary, focuses on empowering existing ones, the Service Plane has the goal of matching to each existing service one or more synthetic services representing it internally to the platform. Each synthetic service handles all the specificity of the target Producer service like protocol, billings, and authentications taking charge of all the necessary coordination with other planes present in the proposed solution, specifically the Cross Cutting Concern Plane. Moreover, the so-defined services can be further composed or decomposed to create new synthetic services so as to create offerings at **different granularities**, e.g., a service of transportation can be composed starting from a sharing mobility service and a public transport one. To enable these advanced techniques the Service Plane also introduces a series of composable categories applicable to services that can not only allow a simpler composition of services but also simplify the suggestion of alternatives to the final customer, e.g., to suggest alternative component providers, transportation to reach a point, or nearby hosting structures.

Finally, the proposal defines a *Cross Cutting Concern Plane (CCCP)* consisting of a series of components to implement all crosscutting concerns and sup-

porting other planes, in the whole management and interaction with internal and external services: we expect a continuous evolution of the CCCP, while dealing with new challenges and new scenarios, especially distributed across a heterogeneous territory like the one covered by a pervasive platform of tourism, smart manufactory, or smart city. The CCCP supports the resolution of problems not only addressed internally to proposed infrastructure but also directly in relation to Customers or Providers, e.g., health checks of provider services and endpoints. Some services belongings to the CCCP can not only support other services but can become themselves part of the offer of other planes, e.g., the authentication services that can be provided as a service directly to Customers behaving as part of the Business Plane. The components realized in the CCCP layer aim to operate transparently with other components of the infrastructure. In this way, the evolution and introduction of new features in the CCCP plane can benefit, with little effort, multiple components belonging to other Planes of the proposed solution.

4.3 APERTO5.0 Architecture Components Full Description

Going deep into a more detailed description of APERTO, we make zoom in on the presented solution that is fully partitioned into detailed layers, each one corresponding to a single business process (Figure 12). We describe here: the presentation layer that implements and proposes a unique view of all services within the Business Plane; the blending and the data layers inside the Data Plane that allow the input of all information needed by polishing and presenting, and also stored within the second component; the analytic and processing layer together with the integration layer constitute the Service Plane, where the former is capable of extracting any possible interesting service from the proposed available ones,

while the latter is capable of getting to all available services available for ST. In addition, the Auditing, Authentication, and Authorization (AAA) layer in combination with the monitoring layer constitutes the first two proposed modules that realize the Cross Cutting Concern Plane.

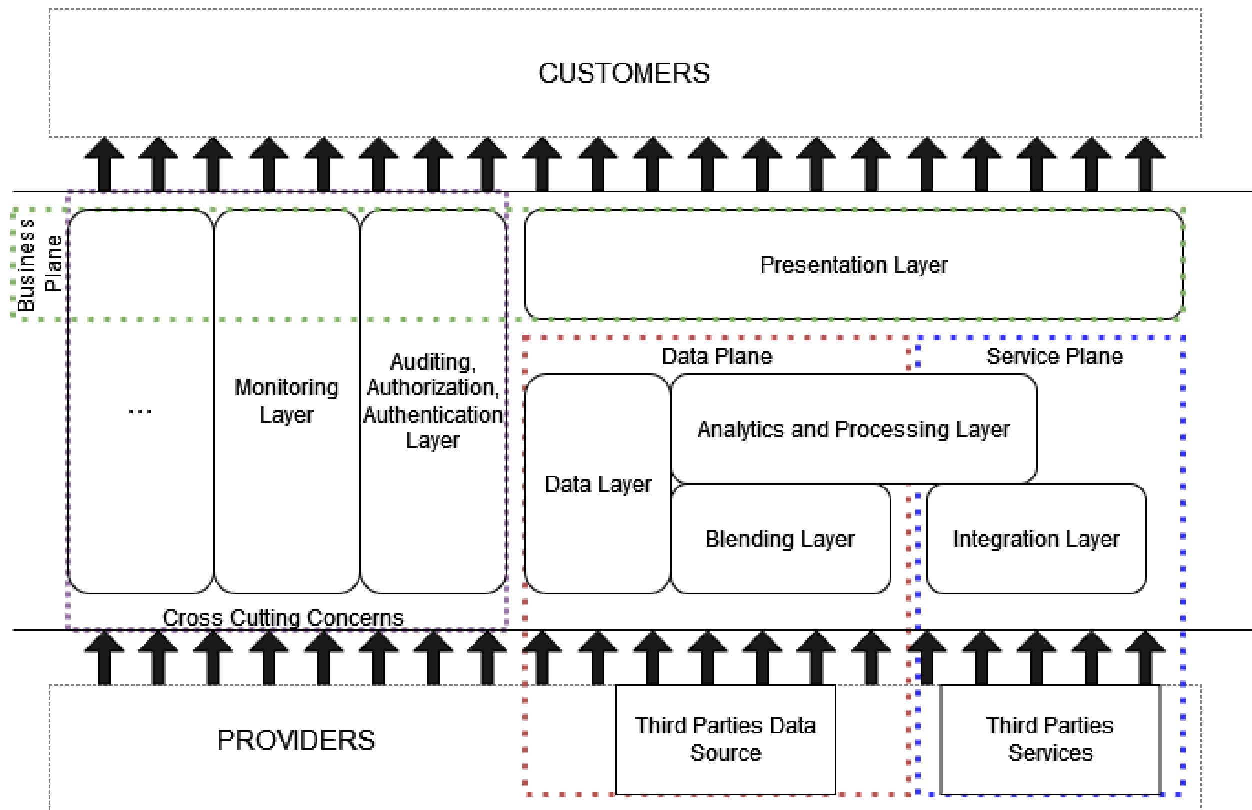


Figure 12: APERTO5.0 layers more detailed view. All components are put together for a more comprehensive effort of integration and synergy coordination.

The *Presentation* Layer constitutes the main component of the Business Plane to create and provide a unified view for customers and third actors, by combining information and services coming from the Data plane and Service Plane to provide new smarter services. Supported customer interaction can employ heterogeneous technical protocols, e.g., pub-sub, client/server, fire and forget, and advanced query languages.

The Data Plane is subdivided into two horizontal layers: the Blending Layer and the Analytics and Processing Layer (AP Layer) and one vertical layer: the Data one.

The *Blending* layer is responsible for gathering, cleaning, and adapting to a convenient format the information coming from third-party services, open data, and custom ad-hoc services. These data sets are then stored, according to pre-defined or dynamic policies on the different storage services composing the Data layer. This lower layer implements the integration logic with the different forms of interactions and queries exposed by external data sources. The interaction methods supported are massaged and adapted to require no changes on the Provider side and can support reactive interaction, such as event-based traffic information systems and scheduled/polling-based ones like information periodically published on a site.

The Blending layer widely exploits principles of modularity and composability, and any introduction of new data sources or data manipulations follows the plug-in logic with a minimum effort of development and instantiation, enabling in such way a sustainable growth of handled producers. This component implements a gathering approach direct from Producer sources, in this way enhancing diversity and customized experience. This approach differs significantly from more traditional ones such as in tourism aggregation portal as booking.com [43] where the hosting infrastructure must register and constantly update its own data in a third-party portal, so requiring a standard format a-priori.

As part of the Data Plane, the *Analytics and Processing* layer aggregates and analyzes the different data sources stored in the Data Layer and exposed by the Integration layer. This layer realizes the process of adding value to the information

coming from Producers, via the internal creation of new aggregate datasets and the discovery of new insights through advanced elaboration techniques, such as Big Data Processing, Data Mining, and AI (Artificial Intelligence) algorithms. This layer can also support real-time event-based and continuous-stream processing to enable advanced real-time queries and subscription mechanisms, as well as batch operations for heavier time-consuming analytics.

The *Data* layer is a vertical layer inside its Plane, since it cooperates with all other layers, by storing elaborated data, schemes, and metadata and by exposing them through advanced indexing and query languages. The Data layer handles the storing of fresh and past collected datasets and metadata, by enabling fast and advanced analytics and interactions through the exploitation of the data locality principle and advanced indexing, and by proposing customer-adapted viewing techniques. To support an effective memorization and query system the Data layer exploits the most convenient storage strategy, so supporting any different data format and memorization technology.

The *Integration* Layer cooperates with the Analytics and Processing Layer inside the Service Plane. The Integration Layer, in particular, is responsible for re-exposing in a convenient and optimized way the external services provided by third parties. The exploitation of many different categories enables the possibility to properly combine and substitute service calls to create more smarter and reactive services, e.g., joining actions of buying tickets for different services, such as theaters, cinemas, or public transportation. This wrapping mechanism enables hiding and abstracting from the peculiarities of each service, such as internal protocol, service call sequence, or rate-limiting, facilitating coordination and combination realized in the Analytics and Processing layer. Moreover, this layer interacts with

all external services and datasets to obtain data not directly available, since they are filtered away, such as real-time number of available tickets or positions and updated time of arrival of a transport. The Integration layer to support integration with different heterogeneous Providers implements many types of interaction including periodically, and reactively.

The *Auditing, Authentication and Authorization* (AAA) layer and the Monitoring layer constitute the core of the Cross Cutting Concerns Plane, available to all other components, to operate in conjunction with all layers in the proposed architecture. These two layers also form an important part of the Business Plane as they provide important services to the final customer e.g., authentication service or metric. In fact, the AAA layer is not only responsible for guaranteeing a proficient level of security to the infrastructure layer but also to provide a unique point of access, for final users, to the services covered and integrated into the platform. This allows to prevent registration and policy adaptation to any tourism service provider and enables a unique view for customers and third-party organizations.

The *Monitoring* layer provides useful insights into the service usage and the overall state of the platform to enable both elastic management of the infrastructure and significant added business value. In fact, from the Monitoring layer is possible to extract and underline trends in services usage with a geographical and temporal connotation and exploit them internally at the platform or supply 'as they are' to external organizations, think to the trends in ticket buyout of a public transport localized in a determined time or region. Moreover, this layer can control malfunctions and unavailability of services and information provided by producers, by generating alerts by-need to request automatic execution of recovery action.

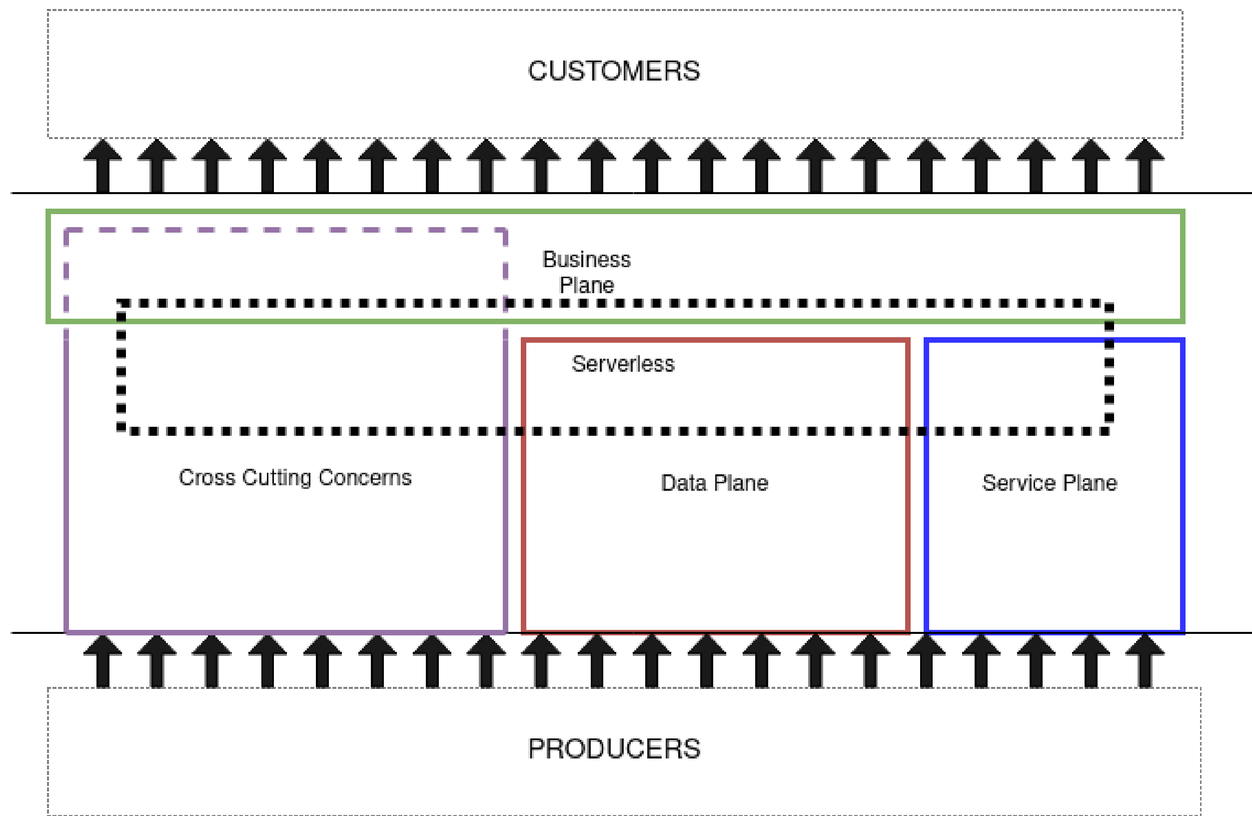


Figure 13: Serverless platform as enabling technology exploited in APERTO5.0 plane to speed up service and data integration .

The complexity and stratification of the proposed architecture derive from the exigence of addressing heterogeneity in highly distributed scenarios seeing the integration of information and services coming from many actors as well as differentiation in customer needs. Moreover, this complexity is expected to further increase with many additional layers and components while increasingly addressing more and more use cases with their intrinsic exigences. we claim, however, the validity of the proposed architecture as a base for the definition of a solution able to satisfy and integrate many scenarios.

As part of the APERTO proposal, we asses the relevance of adopting the FaaS cloud computing model to support scenarios characterized by high heterogeneity in protocols, format, and needs. For this reason, we pervasively integrate

the FaaS paradigm as a support to the development of services in all the layers of the proposed architectural solution.

To meet the complexity of interacting with evolving needs of Prosumer, the FaaS platform, thanks to the low configuration and setup, can significantly speed up prototyping, development, and deployment of new functions. Ease of creation of new functions can so mitigate the cost associated with the complexity of creating new adapters to services and data or the development of new customer-tailored services, such as ones of the presentation layer.

The strong separation among function and triggering protocols further speed-up the creation of new adaptation services thanks to the augmented modularity and reusability of code. Finally, the reduced management effort paired with fine-grained scaling and zero-scaling capabilities of FaaS platforms promises to lower the cost associated with the maintenance online of service opening to the integration of those sources and use-case that otherwise would not meet cost-effectiveness. The aforementioned capabilities of FaaS platform paired with function composition and function chaining support open also to the application of this cloud paradigm to the Analytics and Processing Layer providing parallel computing capabilities with native multi-language support.

5 APERTO FaaS

Herein we assess our APERTO FaaS platform proposal specifically developed to meet the needs of APERTO5.0 in terms of decentralization and easy extensibility. Current Open Source solutions such as OpenFaaS [18] or OpenWhisk, proposals are designed and developed with Centralized Cloud Computing as the target hosting environment. This led to architectures that leverage high-performance networks and virtually unlimited computational resources that could embrace classical synchronous client-server interaction. Moreover, the centralization of orchestration processes drastically simplifies the management of multiple nodes grouped in limited regions. However, the architecture so designed hardly fits in distributed scenarios where network performance varies significantly and the computational resources available in every site can scale from the big data center to resource-constrained devices such as Edge nodes, IoTs, and sensors.

In this context, the adoption of *Message Oriented Middleware* (MOM) is considered a best practice. In fact, the introduction of a Message Oriented Middleware provides **asynchronous** communication among components enabling better concurrency as processes are not requested to wait for the response in order to continue the computation. Moreover, embodying a communication protocol based on the use of a broker, the introduction of a MOM can support the creation of an abstraction layer over heterogeneous resources of distributed scenarios.

At the core of APERTO FaaS architecture, we integrated a Message Oriented Middleware embodying the Pub/Sub communication model and behaving as a backbone for all the communication among architectural components. The introduction of the MOM discharges single components from duties such as discovery,

load balancing, and implementation of delivery semantics which are completely deemed to the middleware.

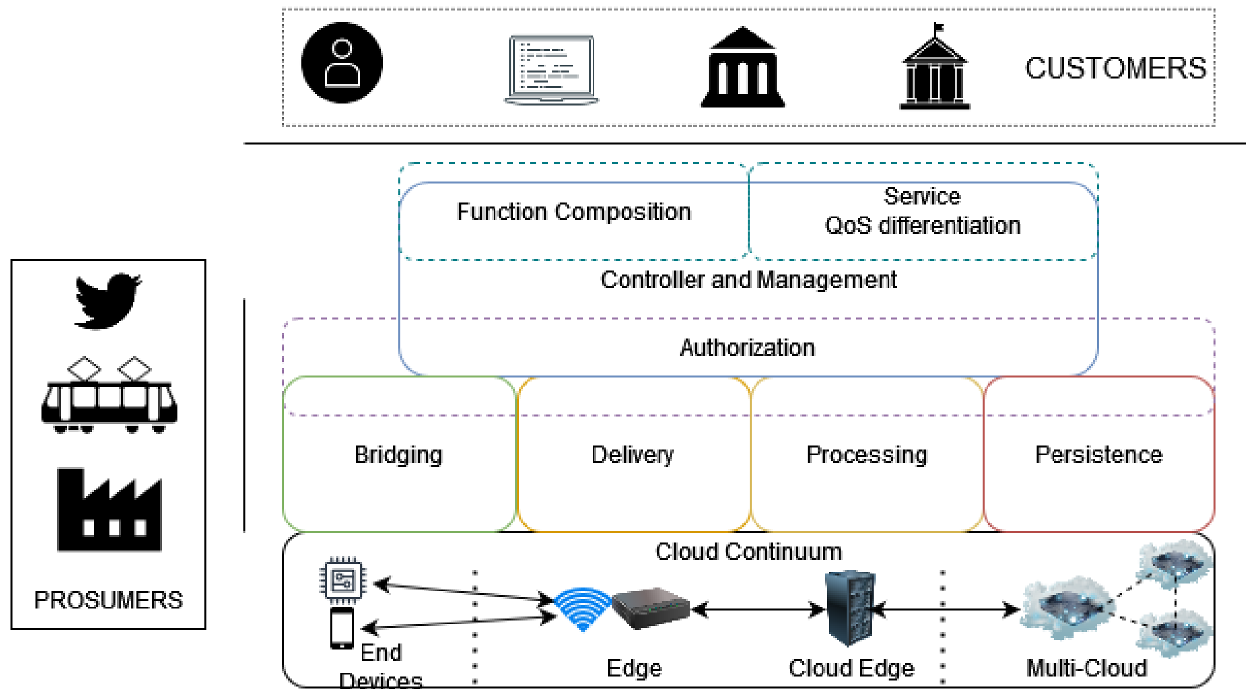


Figure 14: APERTO FaaS architecture partitioned in functional Layers.

The APERTO5.0 Serverless platform can be summarized into 6 functional Layers, as shown in figure 14, subdivided based on the functional role of each architectural component. In particular, the i) *Bridging Layer* is the layer of the platform interacting with the external world and sensing events can trigger the execution of a function, ii) the *Delivery Layer* groups all the processes in charge for the delivery of events, configurations, and metrics among the FaaS components, the *Processing Layer* is the layer responsible for the processing of sensed events through the execution of functions, the iv) Controller and Management Layer comprehends all those processes distributed across infrastructure nodes cooperating to the activation and responsiveness of FaaS workflows, the v) Access Control Layer regulating access to component configuration and provided to the developer at the

application layer to regulate access to functions, finally the vi) persistency layer creating a programming language independent layer of abstractions providing optimized access to heterogeneous and distributed data sources.

Resources and principles of the cloud continuum paradigm could provide benefits in terms of throughput, latency, and availability of the system, nevertheless, the process of orchestration is complicated by the need to simultaneously exploit resources with different characteristics and protocols. Leveraging on top of abstractions and distribution capabilities of APERTO FaaS we propose two orchestration services specifically thought to address the challenges of QoS service differentiation and task coordination when dealing with resources distributed over the Cloud Continuum. Without loss of generality, proposed orchestration approaches are then optimized for the APERTO FaaS architecture and exposed to developers following FaaS model principles and abstractions. We are now proceeding to describe in detail the architecture of APERTO FaaS organized in the aforementioned Layers.

5.1 Bridging Layer

The *Bridging Layer* is the abstraction responsible for unifying requests sent by external entities that want to utilize services distributed through the Serverless platform. The Bridging Layer employs components and mechanisms that transform external events into an internal representation of the FaaS platform so that they can be managed by other layers and transparently processed by functions. In particular, users, devices, and services interact with the workflows through the central component of the Bridging Layer, the *Trigger*. The main responsibility of

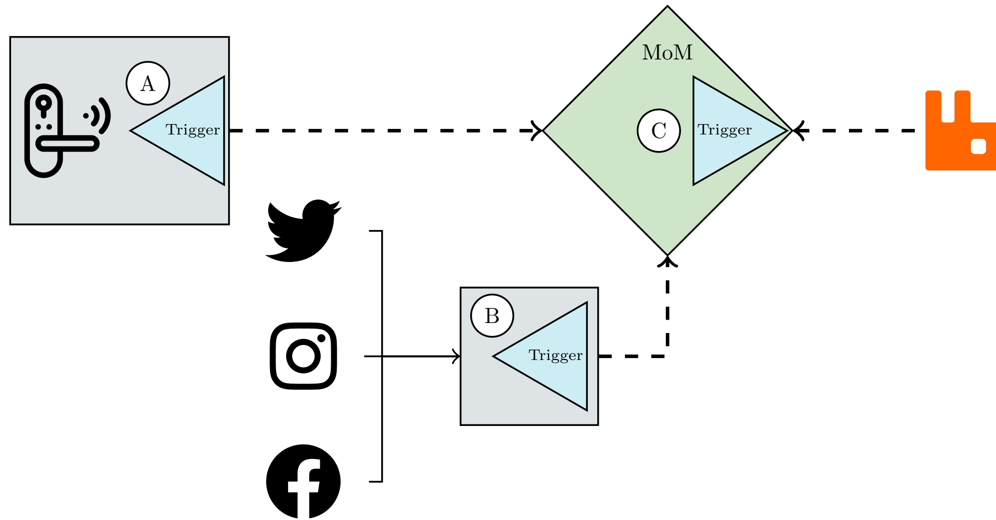


Figure 15: Different deployment options of Trigger: A) locally to source, B) in the middle between multiple sources and the MOM C) locally to the MOM as a bridge to other event systems

the Trigger is so to forward to MOM information sensed or received after having adapted and encapsulated them under the form of events.

Trigger behaves as a bridge between the external world and the FaaS platform, by adapting external protocols, representations, and QoS levels to internal ones. Thanks to the **location transparency** introduced by the MOM, the deployment of triggers can adapt to different scenarios and needs: in particular, we designed and implemented three deployment options for Trigger, depending on closeness to either the MOM or the external source.

In the first case (Fig. 15 case A), the trigger is co-located with the event source. This case allows us to simplify the support of delivery quality between the source and the trigger as they are co-located on the same host. As a drawback, this pattern prevents the simultaneous use of the trigger for multiple sources and also requires that the source device has enough resources to host the trigger execution.

In the second pattern, i.e., Fig. 15 case B, the trigger runs in the middle between external sources and the MOM. The trigger can be located in any node reachable by both MOM and sources and can behave also as a gateway between different networks. In this configuration Trigger is addressable by multiple sources, thus maximizing resource usage.

In the last case, Fig. 15 case C, the external source already exploits message-based communication. This scenario embraces different use case scenarios where the platform is integrated and deployed as part of an existing infrastructure that already leverages some form of event exchange, such as an Enterprise Service Bus infrastructure. In this context, Trigger is placed within the platform MOM and acts as a connector to external sources.

5.2 Delivery Layer

The *Delivery* Layer realizes event distribution among the platform components. As previously anticipated in APERTO FaaS we opted to rely on a MOM for the distribution of function-triggering events, configurations, and metrics among architectural components.

In this type of middleware, communication is typically asynchronous, meaning that the sender and receiver of a message do not need to be connected at the same time. When a sender sends a message, it is placed in a queue, and the receiver can retrieve it at a later time. Many MOM natively supports advanced semantics of delivery, fault tolerance mechanisms, and load-balancing distribution of messages among subscribers of a queue. The MOM introduces a transparency feature that allows adding and/or scaling dynamically the deployment of architectural components. The synergy between the MOM and Controller completely hides the inter-

nal complexity of our middleware from the application developers' perspective, thus achieving an essential feature of Serverless computational models.

The strong decoupling, in space and time, between architecture components introduced by the MOM facilitates the development of an architecture that could meet the needs of highly distributed scenarios such as the one of Smart Tourism, or Industry 5.0 and the distribution of platform components over a continuum of distributed resources. In fact, the **asynchronicity** of interactions between components enables better scalability of the platform by removing waits during interactions among components. The loose coupling between components interacting through the MOM enables also easy extensibility and scalability of the platform by removing points of synchronization among them. The removal of the strong constraint of co-presence in time of components in need to communicate facilitates the realization of zero-scaling capabilities of Serverless platforms as the service in charge of executing a task could be instantiated on demand when needed. This is totally transparent to the other components that interact with it as it is always available.

The separation of intra-component communication and consolidation in a unique common layer easiness the creation of communication optimizations that are inherited transparently by all the components of the platform. At the same time, the usage of a common protocol for communications simplifies developments and encourages software reuse.

The Delivery Layer works at both the Application and the Infrastructural layer providing support in the exchange for configuration and metrics among components and application events generated by the bridging layer. Every FaaS workflow, in APERTO Serverless, activates at least 1 queue accumulating events gener-

ated by triggers and waiting to be processed by functions in the processing layer. In this way, APERTO FaaS supports natively a many-to-many mapping between triggers and functions e.g. an event generated by an HTTP request can be enqueued in the same queue with one generated by a timed event and then be processed by the same function logic.

To allow for differentiated message distribution policies among the subscribers of a queue, we also inherited, from classical MOM solutions, the concept of *Subscription Groups*. In the classical pub-sub model, each subscriber of a topic receives all messages sent through it. Through the mechanism of Subscription Groups, each message is sent to only one of the subscribers in each group, thus realizing a load-balancing feature among the subscribers. Indeed, load balancing is an essential capability of a MOM in the context of FaaS platforms as it enables the distribution of workflow requests across multiple executor nodes.

5.3 Processing Layer

The *Processing Layer* is the abstraction Layer responsible for processing events forwarded by the Bridging Layer and then handled by the Delivery Layer. The Processing Layer takes the burden off the customer of knowing both the characteristics of the processing environment and the computing resources used to execute a specific workflow. This layer allows the customer to define both the business logic and QoS requirements, without knowing how the platform implements the support that can satisfy them. Specifically, the processing is done through user-defined business code uploaded in advance by customers.

The main component responsible for the Processing Layer behavior is the *Invoker*. The invoker is the terminal part of each MOM queue and waits for the

arrival of events to be processed by functions. At each event arrival, Invoker instantiates the associated function, set it up ahead of time through the user-provided configuration, and then takes care of forwarding the event to the function. Thanks to the abstractions and asynchronous interactions provided by the MOM, APERTO FaaS can integrate and execute at the same time all three architectural models detailed in section 3.1. In fact, none of the APERTO FaaS components, except the controller, are aware of how an event will be processed by a specific Invoker; Invoker is, therefore, the component in charge of managing the life cycle, the execution environment, and the invocation of the functions. APERTO FaaS Invokers can be specialized to exploit different function invocation methods and execution environments depending on deployment scenarios and achieving better performance or resource-saving.

This specialization can take advantage of the opportunistic composition of different technologies available either in the environment of execution of the Invoker, e.g., Operating System, Hypervisor, or realized by the Invoker itself. So, for its execution, the same workflow can exploit different technologies and optimizations at the same time, e.g. concurrent usage of an execution environment for 2 different functions or re-usage of the same function instance for subsequent requests.

5.4 Controller and Management Layer

The Controller and Management Layer is the set of processes accountable to orchestrate and configure architectural components in order to activate and maintain responsive FaaS workflow.

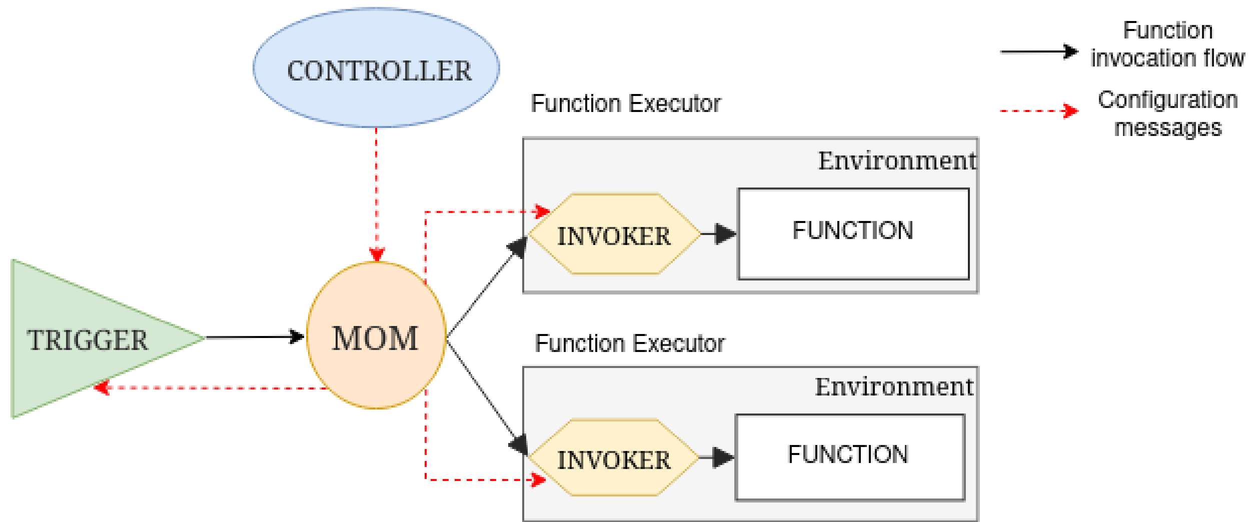


Figure 16: High-level APERTO FaaS architecture highlighting decentralization of controller process and removal from component interactions chain in workflow executions.

To reduce the presence of central points in the architecture which could potentially hamper platform scalability, we propose an evolution of the classical MOM-aided FaaS model by removing the Controller from the sequence of invocation of functions(Fig. 16). This reduces the number of processes actively involved in each request with consequent benefits in terms of resource usage and overall performance. Moreover, by removing **centralized decision** processing from the execution flow we remove a possible point of synchronicity and performance bottleneck which could hamper the scalability of the system. At the application layer, the routing of events towards the invoker components is done directly by the MOM with the configuration of queues, publisher, and subscriber made ahead of time at workflow activation. Eventual interventions of the controller on active flow, such as the scaling of the number of invokers, are made asynchronously taking advantage of the transparency provided by MOM.

At the architectural layer, the controller components follow the same principles adopted at the application layer. The controller interacts and configures other components asynchronously by writing configurations on the specific queue associated with the components it needs to target and receiving acknowledgment asynchronously through the MOM. Seamlessly, monitoring information and request from other components are received and processed by the controller asynchronously. The resulting strong decoupling and asynchronicity in interactions enable the **modularization** and **distribution** of the Controller over multiple processes, executed conveniently over the continuum of resources.

From a customer perspective, the proposed change of control model implemented by the controller is completely transparent with the developer uploading workflows definitions along with the code to the controller. The Controller then once elaborated on the configuration transmits the single configuration to the infrastructural components through the queue of the MOM. The infrastructural components subscribed to the queue once received the configuration executes the actions necessary to activate the workflow.

5.5 Data Persistence Layer

The Persistence Layer is the layer of APERTO FaaS providing abstraction to operate over heterogeneous data sources. Highly distributed scenarios such as the ones addressed by APERTO5.0 often require access to data stored on heterogeneous technologies and imply different protocols to access them. In the past years, many frameworks such as Hybernate [44], EclipseLink [45], or SQLAlchemy [46] established in the market, providing performance optimizations and abstractions to access data sources. However, those solutions were realized at the framework

level and so compatible with each one with a subset of programming languages hampering their applicability in modern cloud environments where multiple languages are exploited at the same time by different services and micro-services. Moreover, the rapid scalability required for modern cloud platforms, and in particular in FaaS ones, prevents an effective integration of **optimization techniques** based on the durability of the instances such as connection pooling or near computation caching.

Emergent workload management approaches rely on the containerized serverless paradigm, decoupling state and computation to gain scalability, with data and state being externalized, and when applicable, stored on cloud storage solutions [47] []. The **ephemeral** and lightweight nature of functions poses some challenges to data access and persistence solutions for serverless platforms. As a consequence, data stores are evolving, integrating new protocols and mechanisms to facilitate and improve data access performance from serverless platforms.

Cloudburst presented in [48] advocates for a stateful FaaS platform building on low-latency mutable state and communication. The proposal relies on the Anna key-value storage and a cache layer, retaining the autoscaling benefits of serverless computing co-located with function executors for data locality [49]. Cloudburst also provides a combination of lattice-encapsulated state and new protocols for distributed session consistency. While this proposal represents an experimentally viable solution to the problem of serverless data access, the migration of existing application/service data could be unfeasible due to underlying technological incompatibility of the data stores and/or as a consequence of a high data migration cost.

Different strategies and data access solutions have been proposed both in commercial and open-source serverless platforms. On the commercial front, we usually see the introduction of a third (durable) component acting as an intermediary between the data sources and business logic, and such are the solutions proposed by Microsoft Azure [50] and Amazon AWS [51]. In these settings, functions interact with the data store via durable components exploiting connectionless protocols such as ReST. The introduction of a third entity acting as an intermediary not only contributes to potential increases in latency due to network-based communication but can also lead to increased billing costs incurred by the service.

Shredder [52] proposes a novel approach integrating storage and a serverless computational layer with the objective of providing a low-latency, multi-tenant cloud storage solution. *Shredder* functions can take advantage of data locality and native acceleration mechanisms to provide access and processing capabilities over data stored in the platform. The approach advocated by Shredder is tightly coupled to the underlying data storage support. At the same time, the scalability of the platform is limited to the availability of resources allocated to the storage layer.

In [53] the authors propose *SONIC*, a data-passing manager that optimizes application performance by transparently selecting the optimal data-passing method for serverless workflows, and implementing a communication-aware function placement. The authors present a prototype of *SONIC* built over the OpenLambda platform, comparing the proposed solution against several others, demonstrating a reduction in response latency. While the approach taken in SONIC is shown to improve response latency, its design is limited by the underlying data store performance. Our approach, instead, is data store agnostic and aims at addressing

response and data access latency over pluggable datastore supports. Thanks to this design philosophy, our approach facilitates function reuse and composition by providing a common and well-defined interface to interact with the data layer.

The Persistence Layer is then specialized into Serverless Persistence Support (SPS) to better support Serverless/FaaS platforms. Is important to stress that the contribution proposed with the Persistence Layer, while specifically optimized for Serverless Computing, can be easily generalized as a general-purpose service for other Cloud Computing models such as PaaS or Container as a Service (CaaS). In particular, the architectural model proposed can provide the same performance benefits to all those cloud computing models adopting the **sidecar proxy pattern** for service communication such as service and event meshes.

The SPS addresses data source integration at an infrastructural level, as opposed to being provisioned as a platform function, allowing the developer to focus only on the adaptation and integration logic represented by the function. To this end, SPS evolves the Invoker architectural component, enabling rich interaction with the function it spawns, which in the classic FaaS architecture it is limited to the forwarding of the event which has triggered the function invocation and the return of an eventual result from the function.

In SPS [54], after function bootstrap, the Invoker opens a bidirectional communication channel with the latter over which access to data can be mediated. To this end, the invoker exposes two different interfaces. The first interface exposes the standard, datastore independent, Create, Read, Update and Delete (CRUD) operations. The second interface extends the first one with datastore-specific operations, such as exploiting advanced querying capabilities. This duality of behavior stimulates a strong decoupling from business logic and specific implementation

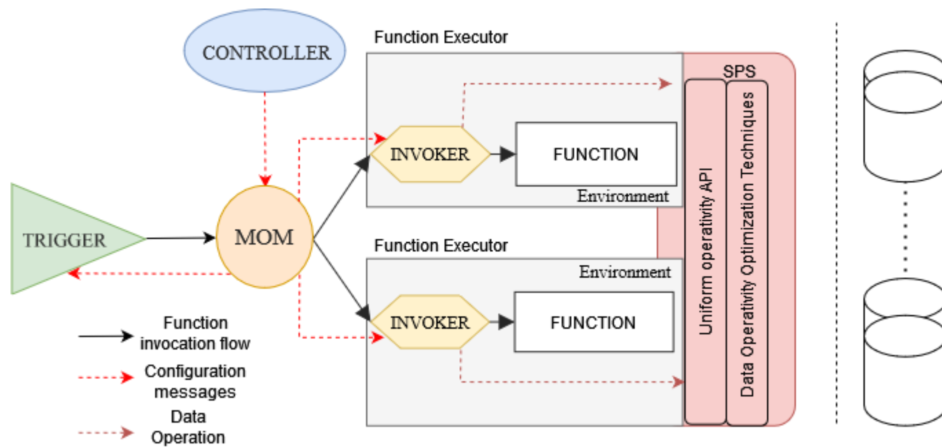


Figure 17: High-level architecture of persistence layer integration in APERTO FaaS showing the SPS operating as an abstraction and optimization layer for data operations requested in functions

of the data persistence layer, similar to what is already provided by widespread standards such as Java Persistence API (JPA). At the same time, the approach promotes the extensibility of the interface, providing more flexibility to the developer.

Once a function triggers an operation on the data, it is up to the invoker component to effectively execute the operation on the data store. Since the invoker is an architectural component, its lifecycle is handled by the infrastructure controller and can be assumed to have a **longer lifespan** w.r.t functions. SPS exploits this characteristic and the strong coupling between invoker and function to introduce several data access operation optimizations such as connection persistence, reuse, and pooling.

From the customer perspective, the activation of a workflow requires the user to (i) upload to the platform the code embodying the business logic of the function, (ii) upload a configuration file specifying the event that triggers its execution, and (iii) identify eventual data stores the function can access and perform operations. When the controller receives the configuration, it checks if data support is enabled

for that workflow and thereafter creates a container embedding the business logic and the invoker-specific implementation for the desired data store. Finally, the controller configures the trigger and MOM to activate the workflows and distribute the packed function over to suitable nodes.

5.6 Access Control Layer

APERTO5.0 proposes an architecture for service and data integration sourced by possibly multiple partners distributed over arbitrary regions. In this specific scenario, access control management represents a major concern and a challenging task for any development team. Access Control (AC) is a security technique that determines which resources/services can be accessed by a given entity in a computation environment [55]. This process is in charge of mediating every request for resources and data held by a system and deciding, according to a set of security policies, whether the request should be granted or denied.

A common approach consists in implementing the verification and enforcement of access rights within the business code of the services. However, this practice complicates the code maintainability and is considered to be an unsafe programming behavior that makes the application more prone to misconfigurations and errors. Moreover, implementing authorization at the source code level implies code modifications every time an update to authorization logic is required. Therefore, this approach deeply relies upon a careful manual configuration that becomes hardly viable in modern large-scale deployments. Due to the issues related to the tight coupling of authorization and service code, policy decision-making should always be decoupled from policy enforcement. The key advantage of this separation is to allow developers to dynamically modify their access control policies

without making any changes to the software. To overcome these issues, modern services usually rely upon a centralized trust entity that is responsible for the whole management and verification of access control policies.

Although this pattern brings several benefits such as decoupling software from authorization and enabling the sharing of common policies with other services, it still suffers from low reliability and poor scalability. Centralized designs are more vulnerable to attacks because jeopardizing the authorization server can lead to the compromise of the whole system. These problems are even more pronounced in cloud continuum computing due to its dynamic nature and low guarantee over inter-site connection availability.

In recent years, there has been a growing interest in providing more secure access control management for serverless platforms by both the cloud providers and the academic community. For example, in Amazon AWS, customers can use the *Lambda authorizer* [56] feature. When a client submits a request, the API Gateway invokes a lambda function that verifies whether the request should be granted or denied. Nevertheless, this approach is more oriented to user authorization than that of functions, in particular, it does not seem to be suitable for function-to-function communications where the API Gateway should not be directly involved. Moreover, authorization is achieved through an external centralized entity that is not integrated into the platform.

Alpernas et al. [57] proposed Trapeze, the first information control flow system for serverless applications. Trapeze wraps each serverless function through a security shim that intercepts all the possible interactions between the function and the external world. The shim is responsible for tracking information flow and enforcing a global security policy based on a combination of static and dynamic

security labeling. Trapeze forces developers to correctly define information flow policies and makes assumptions about the programming language of the serverless function.

Valve [58] is another serverless platform that assists developers in policy specification and employs a transparent coarser-grained (i.e., function-level) information flow model. It employs an agent inside each function instance aiming at monitoring the function's file and network behaviors. Valve agents sent information to a centralized controller that uses them to identify the flow paths of the application. These flows constitute a default security policy that can be further restricted by the workflow designer. Furthermore, it does not depend on the programming language of the serverless function while implying a lower overhead than Trapeze because the latter does not take into account serverless warm-start performance optimizations.

Sankaran et al. developed WILL.IAM [59], an access control framework that proactively verifies, at the point of ingress, if a web request will be accepted or denied. This access control model relies upon directed acyclic graphs that determine the permissions associated with roles (assigned to workflows) and the programmed workflows in the application. Thus, adopting this approach enables applications to detect and reject illegitimate requests as early as possible, saving computational costs and reducing the potential attack surface. Although these solutions allow decoupling policy evaluation from software and provide a more secure information flow control, they still rely upon a centralized access control enforcement point that degrades system performance and causes unavailability problems in multi-site distributed scenarios.

In order to practically address the aforementioned issues, we propose [60] the separation of the AC responsibility from the Controller component and the distribution of this process in order to create a performant and fully decentralized solution. Our proposal still **decouples** authorization decisions from software implementation, thus allowing dynamic policy updates, and does not rely upon a single centralized authorization service. The proposed approach is specialized to provide an effective and reliable solution for managing authorization in serverless environments without significantly impacting performance. However, the approach proposed maintains its validity as a solution for an effective Access Control service in all those cloud computing models enabling the distribution of computational resources over the cloud continuum. Moreover, the model applies seamlessly to all those modern cloud computing models implying the exploitation of a sidecar component for service communication such as in event and service meshes. Finally, it is also noteworthy that this layer can potentially behave like a **cross-cutting concern** not only for the application and architecture layers but also for all the other services (e.g., storage layers, container orchestrator platform) that can be coupled with a serverless platform. This proper nature of the authorization layer allows for significantly simplifying the management of access control to both providers and customers.

Before activating a workflow, some preliminary operations must be performed. Firstly, the customer provides a configuration to the platform specifying interactions and components involved in the activation of a specific workflow. In this process, the authorization layer is responsible for verifying whether the customer has proper rights to modify the configurations of each component involved in the

activation of the workflow. Then, the customer through the configuration can associate a set of access control rules to be checked at the invocation of the function.

Although the authorization mechanism is distributed across nodes in the cloud continuum, it will be perceived by clients as a centralized external service. This transparency provides users with a unified global perspective of the infrastructure that will further simplify the operations needed for managing the whole workflow. While still relying on the controller to configure and deploy new workflows, the verification of the access control rules can not be based on this component since in APERTO FaaS it does not behave anymore as an active participant in the workflow process. In order to pave the way to a better parallelization and distribution of access control verification, we decided to integrate the authorization verification within a process deployed in each **Executor Node**.

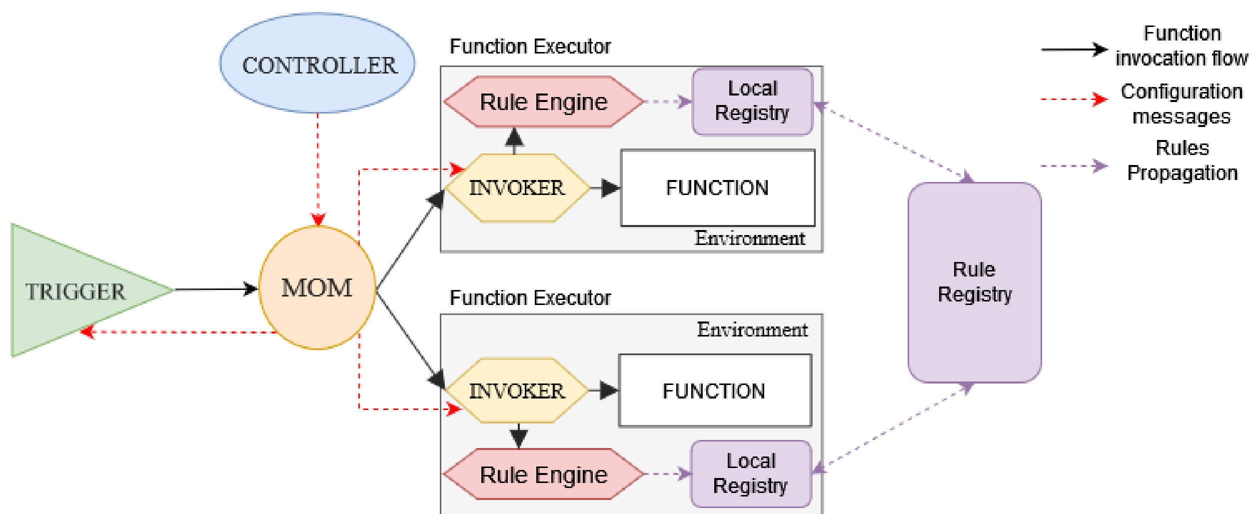


Figure 18: Our decentralized architecture for access control verification integrated into APERTO FaaS platform environment.

In particular, the proposed architecture sketched in Figure 18, introduces in the APERTO FaaS architecture three novel components: the Rule Engine, the Local Registry, and the Rule Registry.

At the reception of an event, the invoker firstly forewords the content of the event and the identifier of the function that should be executed to the co-located Rule Engine. The Rule Engine is a stateless process that starting from Rules expressed in a specific language and contextual information provided by the request and external systems, decides whether the submitted request should be guaranteed or not. One of the biggest issues of access control verification and evaluation in serverless platforms is its additional overhead that results in increased end-to-end latency. The decentralization of the process in charge of computing access rule jointly with the **co-localization** of the two components enables to minimization of overhead associated with the verification process.

The verification is based on data and policies, which are provided through heterogeneous sources (e.g., central registries, and users). Therefore, the management of such information is a trivial task that, according to the scenario, must meet different requirements spanning from strict consistency to high availability. Although this work mainly focuses on access control verification, we also introduce an *Local Registry* working as an offline copy and adaptation mechanism between the rule engine and the sources of rules and information exploited in the process of access verification. The Local Registry is distributed on each executor node similarly to what do with the Rule Engine providing the same benefits in terms of availability and low access overhead.

The introduction of the Local Registry solves the problem of momentary unavailability of the sources at the cost of a possible transitory inconsistency. Our system decides then to sacrifice the total consistency of the system in favor of a greater partition tolerance and availability of the overall system [61].

To provide the abstraction of a single source of truth our solution also introduces the Rule Registry component, a persistence service, logically centralized on which the user could upload and modify rules. The Rule Registry then acts as a master in the synchronization with Local Registries propagating updates and new rules guaranteeing in this way the long-term **consistency** of rules. The decoupling between the Local Registry abstracts into single implementations opens to fast evolution of the two components in order to meet constraints and needs of different deployment and application scenarios. The process of verification of access bases its function on external information constituted by data and policies. That information and policies can come from different and very **heterogeneous sources** such as central registries, the user itself, or the context in which the function is executed. Moreover, the management of this source of information it's not a trivial task and can require complex systems in order to meet very different constraints dictated by the scenario spanning from strict consistency to high availability.

Although this work mainly focuses on the process of access control verification, as part of our architecture we define the registry as the source of all rules and information. In order to cope with many different scenarios and requirements, we also introduce an adaptation layer between the Rule Engine and the Registry layer to enable concurrent access to different registry solutions.

5.7 End-to-end QoS Service Differentiation

In this section, we will propose Time-Effective Middleware for Priority Oriented Serverless (TEMPOS), an innovative middleware extending capabilities of APERTO FaaS Layers in order to enable the composition of the different technolo-

gies available in the cloud continuum to achieve an effective end-to-end Quality of Service (QoS) differentiation between services hosted.

The types of applications that are requested to run on cloud continuum infrastructures are very differentiated and with very differentiated requirements, from latency upper-bounds to maximum allowable downtime and reliability; moreover, the ICT infrastructures hosting them include very **heterogeneous resources** and tend to employ more and more cloud continuum virtualized resources, which are typically positioned close to IoT sensors and actuators, in the Far Edge, for greater efficiency [1]. In addition, in these scenarios, depending on the industrial sector and the specific kind of application, the severity of effects due to provisioning failures can range from negligible to critical.

In this context, the definition and efficient usage of prioritization mechanisms become necessary to meet the different QoS demands of different types of IoT applications, composed of multiple tasks competing for the same virtualized resources. In addition, single mechanisms are not sufficient: the coordination of different prioritization technologies, across the full invocation stack (possibly including virtualized processing, invocation messaging, and communications) is needed to meet constraints of end-to-end jitter, latency, and queuing delays [62]. However, such coordination and orchestration of distributed resource reservation and invocation prioritization, while maximizing the efficiency of resource utilization, is recognized as a challenging task. This is further exacerbated by the huge heterogeneity of devices, operating systems, communication mediums, and protocols present in Far Edge cloud-enabled IoT environments [63].

The need for control and compliance with QoS specifications in cloud-to-things environments for industrial manufacturing is widely recognized. But sim-

ilar needs are present more and more in other application domains, which are increasingly benefiting from IoT-empowered technologies. For example, in the Smart Hospitality domain, modern accommodation facilities are integrating more and more connected sensors and actuators to provide an increasingly digitized and personalized experience for their customers [64].

IoT devices, along with digital services, work alongside staff promising to help manage and create a more engaging experience, while also achieving accessibility and reduced environmental impact goals. Hence the need to provide differentiated QoS levels in the delivery and processing of information from different devices. For instance, IoT devices embedded in smart doors or management systems (e.g., SPA temperature controllers) require low latency and small variation in response time; at the same time, the growing number of AI and Virtual/Augmented Reality technologies embedded in both guest rooms and hotel gyms [65], while requiring high bandwidth and benefiting from localization of computational resources, can tolerate small performance degradation.

The opportunistic usage of edge cloud resources to improve latency and jitter has been extensively discussed in [66, 67] and also represents one of the key factors pushing for wide adoption of this computing model [68]. The coordination and coupling of different prioritization mechanisms is not a recent issue but, with the recent advent of next-generation networking, it has gained an increasing research interest. The need for concatenation of mechanisms present at different levels of the stack has been considered a primary problem since the earliest distributed systems. To tackle resource orchestration and partitioning while guaranteeing QoS levels at the edge, [69] proposes DRAGON: that paper describes some

implementation insights about DRAGON and evaluates its performance benefits if compared with traditional orchestration approaches.

The introduction of middleware for the concatenation of QoS-aware composition mechanisms is a frequent design pattern applied in the literature to reduce complexity. In [70] the authors propose a technique to couple priority and reservation-based OS and network QoS management mechanisms through Distributed Object Computing middleware, with adequate performance results.

In [71] the authors present a middleware built on CORBA for providing distributed soft real-time applications with a uniform API to reserve heterogeneous resources with real-time scheduling capabilities in a distributed environment: that solution introduced uniform interfaces to support the reservation of CPU, disk, and network bandwidth on Linux systems.

The application of FaaS Cloud Computing to Edge environments characterized by the limited availability of resources is considered an appealing direction for the peculiar capabilities of FaaS in terms of **zero** and **fine-grained** scaling. Notwithstanding the novelty of the FaaS computing paradigm, some platform improvements have already been proposed in the literature to achieve better FaaS performance and in particular latency reduction. Some papers have proposed the deployment of serverless platforms on edge nodes to achieve better QoS [72], such as in TEMPOS. The usage of different invocation methods to speed up function startup has been proposed as the exploitation of cross-compiling to achieve faster executables. For example, in [73] the authors propose *Faaslets*, an isolation abstraction that exploits WebAssembly to achieve good isolation and fast function startup; they also propose an additional optimization with a mechanism to restore

from already initialized snapshots, thus improving platform throughput and tail latency.

In the proposed project Catalyzer [74] the authors propose a serverless sandbox system to enhance function startup and isolation. To provide fast startup, Catalyzer exploits a checkpoint mechanism to skip initialization and a new OS primitive to reuse the state of the running sandbox; this results in a relevant reduction of the startup time of function invocations, up to less than 1 millisecond in the best cases. However, despite the relevant evaluations introduced in the FaaS ecosystem, opening the application of this paradigm to a rising number of scenarios, the proposed approaches do not solve the problem of providing coordination mechanisms in order to enable concurrent execution in the cloud continuum of different applications with different QoS needs.

To fill this relevant gap, we proposed TEMPOS [75]: a Time-Effective Middleware for Priority Oriented Serverless IoT applications in the cloud continuum. TEMPOS is a novel middleware, specifically designed and optimized for advanced QoS management in FaaS infrastructures, that hides the possible heterogeneity and complexity of edge deployment environments while providing a strong QoS separation among the workflows put into execution.

To this purpose, the TEMPOS orchestrator coordinates and composes different technologies of prioritization and reservation available across the full support stack associated with the different virtualization layers involved in FaaS infrastructures deployed over the edge cloud continuum. Thanks to its complexity hiding, TEMPOS can be exploited in multiple diverse scenarios such as Smart Tourism, Industry 5.0, or Smart Agriculture. TEMPOS abstractions require only the definition of the business logic in form of a workflow with an associated QoS level

(among the available ones). It is the TEMPOS middleware that asynchronously checks the QoS support of the targeted resources in the deployment environment components and updates the configuration of the single FaaS components accordingly.

TEMPOS integrates and extends the APERTO FaaS layers(Fig. 19) in order to create the needed end-to-end QoS differentiation behavior across all architectural components.

In the Controller and Management Layer, TEMPOS extends the controller in order to create an abstraction for developers to expose applicable configurations in a simplified and facilitated way. To this purpose, the Controller maps and matches the QoS requested by an application developer/deployer to the quality levels supported by the different layers and exposed through well-defined interfaces.

More specifically and practically, the Controller handles two different configuration steps. On the one hand, it receives the configuration for all components of the TEMPOS infrastructure from the developer/deployer, then remaps this configuration to specific commands sent synchronously to each of the different layers. The goal of this phase is to configure Channels and Topics with the different QoS levels offered by our middleware. On the other hand, the Controller receives the set of workflows initially defined by the developer/deployer, along with the QoS expressed with per-flow granularity (not for single invocation) and that will have to be mapped to the underlying infrastructure. The developer/deployer can also request the configuration of workflows at runtime, as the Controller exposes APIs for deploying new workflows or modifying existing ones. As in the other cases, these reconfiguration events are handled by the Controller, which interacts syn-

chronously with the TEMPOS layers to preserve the QoS required for the whole application.

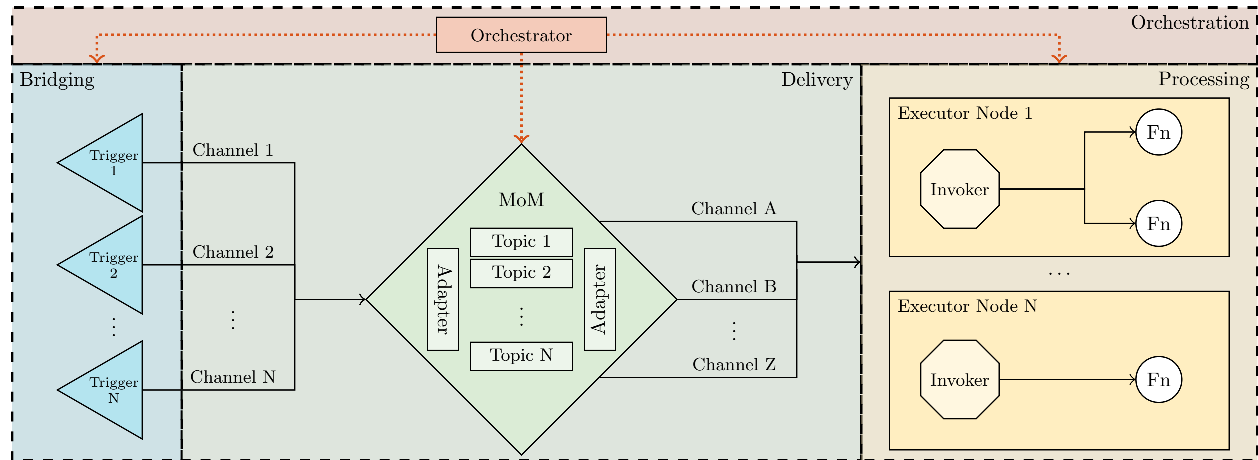


Figure 19: High-level vision of TEMPOS integration in APERTO FaaS architecture showing QoS differentiation based on TEMPOS channel composition .

In the Delivery Layer, TEMPOS extends event distribution by introducing a QoS-aware prioritization mechanism. TEMPOS event distribution process is achieved through the inter-working of different communication technologies and protocols, along with services in the duty of orchestrating and composing them. A series of abstractions are then introduced to easily extend the set of supported technologies and to provide developers/deployers with a simplified view. The core part of this layer is a novel Message-oriented Middleware (MOM) capable of dynamically exploiting different mechanisms and technologies to achieve QoS differentiation.

Application/middleware components can connect to our MOM either to send or receive a message, through the creation of a *Channel*. In fact, the TEMPOS Channel is the abstraction that we offer to define a connection between any pair of FaaS architectural components.

Since the Delivery Layer potentially covers several communication environments, a Channel is characterized by a specific communication protocol and, if supported, a prioritization or reservation technique. Therefore, to employ our delivery notion in highly heterogeneous contexts, the MOM adopts a mechanism based on the concept of Adaptor. Adapters allow to support a considerable number of Channels and interact with them seamlessly and simultaneously. Messages received by a specific Channel are processed in priority order through the use of "priority queues". Through these queues, the TEMPOS MOM processes events in parallel, prioritizing those associated with higher QoS.

To provide our middleware with a consistent end-to-end quality abstraction, we introduced the concept of *QoS-aware Topic* defined as:

$$T = \left(\begin{bmatrix} C_{in1} \\ C_{in2} \\ \vdots \\ C_{inN} \end{bmatrix}, Q \ , \begin{bmatrix} C_{eg1} \\ C_{eg2} \\ \vdots \\ C_{egN} \end{bmatrix} \right)$$

where

$$T = \text{Topic},\ Q = \text{Priority Queue},$$
$$C_{in} = \text{Channel Ingress},\ C_{eg} = \text{Channel Egress}$$

The topic is the reference construct in TEMPOS for coordinating and abstracting the different QoS levels made available by the channels and associated with the priority queues of the MOM. Each topic is then associated with a specific QoS, which can be derived from the performance of the two channels with the worst input and output performance, respectively, and the processing performance associated with the Processing Layer. Thanks to the Topic and Channel constructs, it is thus possible to provide application developers with a single and transparent

view, even if the platform is leveraging different QoS-sensitive technologies, such as TSN, 5G slicing, or Wi-Fi 6 prioritization.

The Bridging Layer plays a fundamental role in QoS differentiation. It is in this layer, indeed, that the first differentiation happens and QoS characteristics are applied to events.

Trigger behaves as a bridge between the external world and TEMPOS, by adapting external protocols, representations, and QoS levels to internal ones. Moreover, *Trigger* is the first TEMPOS component that can differentiate and characterize event quality by exposing a different endpoint for each supported QoS level. In the previous section 5.1 we claim the novelty introduced by our decentralized architecture is able to transparently integrate the deployment of the trigger in three different scenarios: trigger co-located with event source, trigger in the middle, and trigger as event adapter (sec. 3.1). Depending to the deployment scenario adopted, TEMPOS exploits different mechanisms in order to differentiate QoS.

In particular, in the first pattern saw the trigger is co-located with the event source (Fig. 15 case B). This scenario simplifies the support of delivery quality between the source and the trigger as they are co-located on the same host and so only inter-host prioritization mechanisms are exploited e.g. Operating System scheduler.

In the second pattern, i.e., Fig. 15 case B, the trigger runs in the middle between external sources and the MOM. In this scenario, we have the certainty that the delivery of information between the source and the trigger is feasible with adequate quality.

In this configuration Trigger is addressable by multiple sources, thus maximizing resource usage but also potentially causing conflicts. Of course, incoming

events belonging to the same quality class can incur conflicts in case of concurrent transmission; a fine-grained distribution and allocation of Triggers are so advisable to avoid situations of quality degradation.

In the last case, Fig. 15 case C, the external source already provides QoS concepts and exchanges information in the form of events. In this context, Trigger is placed within the TEMPOS MOM and acts as a connector to external sources remapping arguments, queues, and qualities of an external system onto internal ones.

Finally, in the Processing Layer, TEMPOS extends executor nodes behavior to enable a strong QoS differentiation and prioritization in the processing through functions of events delivered. This Layer allows the customer to define both the business logic and QoS requirements, without knowing how the platform implements the support that can satisfy them. Specifically, the processing is done through user-defined business code that is loaded in advance.

To preserve total transparency and Independence of the Invoker component from other architectural components even in TEMPOS none of the other platform components, except the controller, are aware of how an event will be processed by a specific Invoker; Invoker is, therefore, the component in charge of managing the life cycle, the execution environment, and the invocation of the functions in such a way as to reach the target QoS for that specific workflow.

Regarding QoS-aware processing, Invoker can employ both its internal techniques and the ones possibly present at the Executor Node, e.g, Operating System prioritization. Thanks to the Invoker abstraction, TEMPOS is capable of executing heterogeneous functions while employing different QoS mechanisms and policies, without causing side effects on other components or executor nodes.

5.8 Distributed Task Composition over Cloud Continuum Resources

The coordination of execution of processes and services is an appealing capability of distributed systems enabling the creation of complex workflows and analytics starting from simple resources and services available in the continuum of resources. These approaches result also in a greater appeal in those contexts involving many actors distributed in the territory. In APERTO5.0 the introduction of service composition could enable the creation of complex workflows starting from the many resources made available from the different partners leading to smarter and more integrated behaviors. As an example, a manufacturer at the delivery of a commission can automatize updates of the warehouse and of the different groups and machinery in charge of its production. In a tourism context, a tour operator can create booking packets that simultaneously exploit booking services from multiple partners, and create a more integrated experience.

The coordination of different services due to their intrinsic nature can really benefit from the adoption of a FaaS model, abstracting the heterogeneity and complexities of the distributed platform. In FaaS platforms, as already stated in section 3.2, the ability to compose (ready to use) functions to create application-specific processing pipeline(s) is typically called *function composition*. By decoupling complex functionalities into simpler ones, function composition enables smarter management of complex tasks and improved multiplexing capabilities. Moreover, function composability promotes reusability, thus further reducing the development burden, hence the time to market.

While the composition of services in cloud continuum environments already presents several challenges, due to the difficulty in coordination and the need to in-

tegrate heterogeneous protocols and resources, their execution on FaaS platforms adds challenges derived from the peculiarities of this model. The **opaqueness** of function execution, in fact, makes it more challenging to implement efficient mechanisms for **process coordination**, such as broadcast, aggregation, and shuffling, which are common communication primitives in distributed systems. This aspect is particularly relevant when considering machine learning and big data analytics workloads [76]. In addition, no current FaaS platform adopts any resource-aware optimizations to exploit the specificity of the underlying hardware and/or software resources. This is becoming an increasingly important feature when considering that functionalities can be deployed over a continuum of heterogeneous resources [77]. These optimizations do not only benefit single-host FaaS deployments but also distributed scenarios where one knows in advance the underlying resource capabilities and the application resource graph. In fact, this is not an uncommon situation, and the multi-host FaaS scheduler can be tasked to handle the resource-aware placement of functions [78].

Despite the relative novelty of FaaS platforms, several solutions, both commercial and academic proofs-of-concept, address the topic of function composition. Our survey is confined to proposals that present system-level novelties and optimizations, in line with our overall objective.

On the commercial front, Microsoft Azure Cloud has recently introduced Azure Durable Functions as an extension of Azure Functions [79]. This solution enables a user to define stateful workflows by writing special orchestration functions, whose state is managed by the platform. Amazon adopts a slightly different approach with their AWS Step Function offering, which behaves as a finite state machine controlling the execution of AWS Lambda functions composition [80].

AWS Step Function allows the definition of a series of checkpoints in the pipeline, used to enable fault tolerance capabilities, such as error handling and retry logic.

Both those commercial approach while providing a good level of expressiveness to customers breaks the fine and zero scaling capabilities of FaaS platforms as a function or another process, opportunely scaled, needs to be always present to initiate and coordinate the function composition leading in this way also to higher costs. Moreover, the necessity of gathering function result values in a unique point in order to effectuate subsequent invocations obviously lead to many more message exchange, and so more resource utilization and greater response latency, to realize the same function flows in respect to other approaches such as continuous-passing.

On the academic front, the authors of [21] identify some formal properties of function composition schemes, proposing a taxonomy of possible approaches. According to the proposed classification, they also discuss an infrastructural approach for function composition based on the OpenWhisk [20] platform.

The above offerings implement the function composition feature following the *reflective invocation* approach, which violates the fine-grained (zero) scaling feature, requiring an always-on entity to enact the composition logic. Moreover, the solutions seem to lack any form of optimization for inter-function communication, which is crucial for an efficient composition solution. On the contrary, we adopt a *continuous passing* composition pattern which does not require the instantiation of a third component and guarantees the best inter-function communication performance, avoiding intermediary entities.

SAND [81] is a serverless computing platform that combines a novel execution environment and fast inter-function communication. SAND promises lower

latency, better resource efficiency, and more elasticity than existing serverless platforms by leveraging an application-level sandbox and a two-level message bus. SAND makes use of a local communication bus that enables efficient function-to-function transfers. However, the proposal executes multiple functions of an application in a single container, violating a core property of FaaS, which is to run and manage code written in different languages and with different dependencies. Moreover, the local bus is implemented as a custom process, lacking a standard interface, interoperability features, and the extensibility and flexibility that DIFFUSE embodies.

FAASM [73] introduces *Faaslet*, an isolation abstraction based on WebAssembly that leverages shared memory regions for communication between functions in the same address space. Faaslets execute in the FAASM runtime, which takes care of isolating other system resources using standard Linux $cgroups$. FAASM achieves better performance compared to container-based solutions in terms of memory usage, function instantiation time, and overall throughput. Although FAASM could have been a good baseline for our work, its architecture does not provide any support for efficient function composition. Moreover, even though WebAssembly as an execution environment is an appealing research direction, especially if combined with other existing technologies, it still lacks stability [82] and some features are required by production-grade systems.

Moving a step forward, herein we present DIFFUSE [83] a DIstributed and decentralized platForm enabling Function composition in Serverless Environments. DIFFUSE expands APERTO FaaS architecture by introducing novel function composition mechanisms integrated into the architecture and an optimized MOM-based approach for function-to-function communication. Abstractions introduced

by DIFFUSE(Fig. 20) enable the **simultaneous integration** of multiple MOM over the cloud continuum, taking advantage of the peculiarities of each one and tailoring sections of the architecture to different needs and constraints. Deployment scenarios can then exploit resource-optimized middleware in edge resources and high throughput-optimized ones in cloud resources. Both can then be exploited at the same time by simply including functions belonging to the two sites in the same workflow definition.

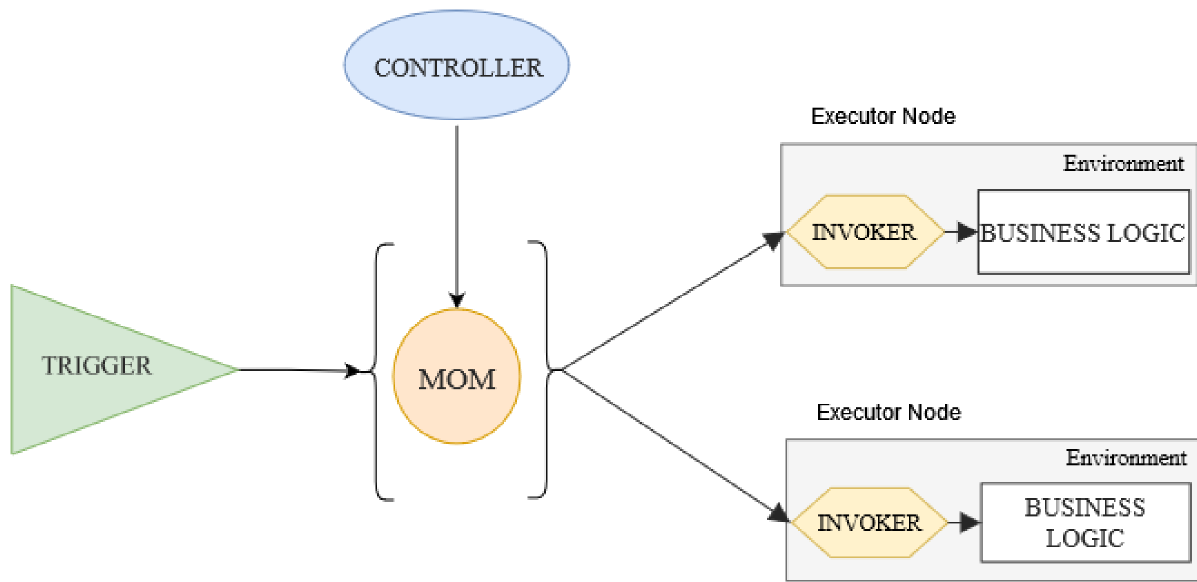


Figure 20: DIFFUSE relies on the peculiarities of multiple MOM solutions to provide enhanced function-to-function communication.

The composition mechanism comprises two layers: (i) the configuration and coordination layer instrumenting the FaaS platform components, and (ii) the function-to-function communication layer serving as a conveyor of messages between components.

Our proposal provides the user with the capability to define custom processing pipelines expressed via association rules, residing outside the functions' business logic. Association rules are shipped to the controller and allow the definition

of generic, graph-shaped processing pipelines whereby the pipeline continuation is determined by the output of the executed function. This also allows us to introduce run-time modifications and updates to the processing pipeline, adding to the flexibility of the approach.

The Controller then converts workflow definition in component-specific configurations forwarded to the designated components through MOM configuration queues. These asynchronous updates allow moving the FaaS composition controller outside the chain invocation mechanism, enabling us to reduce the function response times when compared to the approach where the controller is involved in each function invocation (*reflective invocation*. Our solution then doesn't rely on a single external or internal coordinator receiving results of each invocation and triggering subsequent phases of the composition. It is important to note that in contrast to the *reflective invocation* mechanism where the (control) burden is offloaded to the controller entity, in our approach the control logic is decentralized and distributed in each executor node. This approach follows the direction undertaken in APERTO FaaS to reduce points of centralization to achieve better performance, fault tolerance, and easier distribution over cloud continuum resources.

Without loss of generality, in Fig. 21 is depicted as an example of a processing pipeline that is composed of three functions, namely A, B, and C. In a hypothetical scenario, the execution of the pipeline is triggered because of a user-issued request, calling function A into execution. Upon function A termination, depending on its returned output inspected by the invoker component, the continuation of the pipeline will be either the execution of function B or C.

Once a pipeline configuration file is pushed to the platform, the components establish a series of communication queues (topics) used to exchange applications

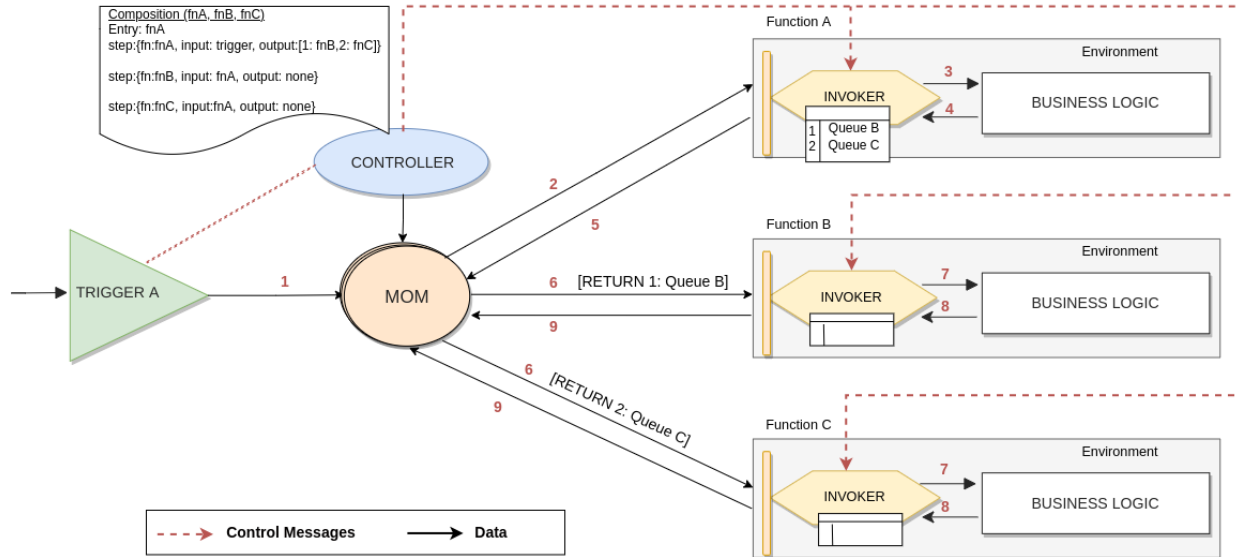


Figure 21: DIFFUSE relies on a MoM-based approach for function-to-function communication; invokers directly trigger execution of the next function by publishing function output on the corresponding topic.

and control data among them. In particular, the function-to-function communication mechanism relies on a MoM-based (Fig. 6) approach, adhering to a pub-/sub paradigm, enabling the **transparent invocation** of the next function in the pipeline. This approach not only enables the distribution of the composition controller exploiting location transparency provided by the MOM but also promotes taking advantage of single peculiarities of the different MOM. The introduction of acceleration techniques, as an example, can be done through its implementation in the Delivery Layer without, automatically benefiting all components.

Returning to our prior example, once function A terminates the execution, the invoker consults the output and depending on the value, forwards the output either to Queue A or Queue B, consequently triggering the execution of function B or C, respectively. This mechanism provides the ability to dynamically scale the number of *invoker* instances, thus increasing the level of parallelism.

In these settings, the MoM acts as a conveyor for all messages and events, hence it is of paramount importance that the solution is efficient and able to gracefully scale with the number of requests. At the same time, it is desirable the platform be **agnostic** and **decoupled** by the specificity of the underlying MoM solution, promoting portability and openness to future extensions. To this end, we have introduced an abstraction layer (vertical orange box near the Invoker and Controller entities, Fig.21) decoupling the components from the specific MoM APIs by implementing a series of high-level abstractions such as group (communication channel) creation, send and receive of messages, etc.

6 APERTO FaaS Reference Implementation

While preserving the general aspect of our study, and without loss of generality, we now present the current status of the implementation of the APERTO FaaS prototype. This prototype demonstrate the feasibility of our proposal and allowed us the creation of an extensive series of tests to demonstrate the validity of our proposal. For better readability, the structure of the chapters follows the high-level subdivision in Layers presented in the previous chapter, while presenting in detail techniques and implementations of the architectural components.

6.1 Bridging Layer

The Trigger is the TEMPOS component responsible for the forwarding of events issued by one or more sources to the MOM, thus defining the core part of the *Bridging Layer*. To provide a unique implementation for the different deployment scenarios presented in Section 5.1, we developed the trigger as an always-running Linux process. This process always implements two protocols, one for receiving requests and events from the extern and the other to send the message through a publish request to the MOM. Depending on the need we have implemented several protocols receiving requests from the extern ranging from HTTP, TCP, or UDP.

The trigger process at its startup first communicates its activation to the controller by publishing a registration message on a specific and unique queue. Then it subscribes to a specific queue waiting for configuration messages and communicated to the controller through the registration message. The configuration message that the trigger receives at the activation of a new workflow simply describes a

tuple containing the queue IDs at which the trigger has to forward a specific event. As an example, a configuration message for a trigger exposing the HTTP protocol can indicate that all the POST requests to a specific path should be forwarded to queues A and B, as another example a trigger exposing the TCP protocol can differentiate the queue destination depending on the port on which he received the request. Once received a request the trigger remaps the data received through the request into a data structure containing all needed information, then the data structure is serialized into a MessagePack [84] and sent to the mom. MessagePack is an efficient binary serialization format that enables to exchange of data among multiple languages in an optimized and compressed way.

To handle those cases where the final user requires a synchronous interaction with the platform, like during an HTTP GET, the trigger insert in the message the id of the queue on which it will listen for the response. All the interactions with the MOM and so with other architectural components of the platform are asynchronous thus maximizing the performance of the trigger process. Moreover, the stateless nature of this process joined with the transparency introduced by the pub-/sub protocols enables a simple scaling of this component by simply creating and executing another instance of it. The transparency introduced by the MOM enables the exploitation of the same trigger implementation for all three deployment scenarios analyzed. Thus, an event issued by a source (e.g., a sensor) is received by the trigger via the network, or if possible, taking advantage of an IPC mechanism. This can be applied to optimize communication in the case of the co-located deployment scenario.

6.2 Delivery Layer

During the development of APERTO FaaS, we integrated and experimented with different Message Oriented Middleware as the Delivery Layer of our platform. As mentioned before

It is worth mentioning that the Delivery Layer is not constituted only by the processes composing the broker but also by protocol implementations of publishers and subscribers can play important roles in message dispatching and management. As an example in the Kafka Broker [85], as better analyzed also in the next section 6.8, the publishers realize the load balancing feature of the middleware by synchronizing with the broker and writing the message to a partition of the Kafka Topic. For many experiments that we conducted and as a possible reference implementation we relied on NATS [86] messaging broker. NATS is a lightweight, high-performance messaging system that enables distributed systems and microservices to communicate with each other. It was designed to be fast, scalable, and easy to use, with a focus on providing a simple and intuitive API. NATS is often used as a ”message bus” or ”event bus” for connecting microservices and other distributed systems, and it supports a wide range of messaging patterns, including publish-subscribe, request-response, and streaming. NATS is written in the Go programming language and is available as open-source software under the Apache 2.0 license.

NATS natively supports the concept of **Queue groups** which allow multiple subscribers to load balance messages from a single publisher. When a message is published to a subject that has multiple queue group subscribers, NATS will distribute the message to a single subscriber in the queue group. This can be used to distribute workloads across a group of clients. As already mentioned, this

feature is exploited by APERTO FaaS in order to distribute the load generated by the same class of event among multiple invokers. Moreover, NATS natively implements the Request-reply. In this pattern, a client can send a request message to a subject, and NATS will deliver the request to a single subscriber/invoker. The invoker can then process the request and send a reply message back to the client. These primitives when available on the adopted MOM can support the Trigger in handling transparently synchronous end-user requests.

Finally, NATS implements different forms of **clustering** and distribution facilitating the creation of an infrastructure spanning the continuum of resources. In NATS, a cluster is a group of NATS servers that are connected together and work as a single entity. Clustering allows scaling the broker horizontally by adding more servers to the cluster. It also provides fault tolerance, as the cluster can continue to operate even if one or more servers fail. When a NATS server configured to join a cluster, starts up, it will connect to the other servers in the cluster and exchange information about the messages it has received. This allows the NATS cluster to route messages to the appropriate server and ensures that every message is delivered. The concept of clustering can be also extended to **multi-site deployment**. In a NATS multi-site deployment, you can set up NATS servers in multiple locations (e.g., data centers, edge resources, or cloud regions) and connect them to form a single, globally distributed cluster. This allows you to build distributed systems that can span multiple sites and operate even if one or more sites go offline. To set up a NATS multi-site deployment, you will need to configure each NATS server with the addresses of the other servers in the cluster, regardless of their location. NATS will use these connections to exchange information about messages and route them to the appropriate server.

6.3 Processing Layer

To cope with the specific needs of each situation and to also highlights specific characteristics of the proposal we have implemented the invoker as a Linux process executing with a different model of concurrency. In the first case, the simplest one the invoker runs as a synchronous blocking process, receiving an event from the MOM, executing the configured function, and waiting for its termination before receiving another event. This configuration facilitates a strong control over the concurrency with which the events are processed by augmenting the number of concurrent instances spawned by the system on executor nodes available. The second implementation exploits a multi-thread concurrent model where multiple functions are concurrently executed by the same invoker. In particular, the invoker reserves a thread as an event loop process waiting for events by the MOM and distributing their processing to the other thread available in the thread pool. This implementation, creating at the software layer a solution of concurrency is designed to be run in a single instance for each executor node. In the case the platform requires scaling resources associated with the processing of events coming from one queue the controller instantiates new executor nodes. Both the implementations are equipped with an additional feature that if enabled allows them to retain current functions already put into execution and treat them as long living processes. While this clashes with the zero-scaling principle of FaaS it also allows us to not pay the cost of Linux process instantiation and achieve in this way a reduction in response latency.

The problem of response latency in FaaS platforms is a well-known open challenge in the sector commonly known as *cold start* phenomena [87]. In fact, the recreation at each invocation of the execution environment and the process

running the function can cause an additional overhead even hampering the application of these models to all those use cases demanding low latency in interactions. Over the past years, many optimizations have been proposed in the literature and by cloud providers [88] to mitigate these issues ranging from the approach yet presented of retaining already instantiated function for process subsequent event to the proposal of new optimized execution environment and process startup. In the following, we present the current state of APERTO FaaS integrating 3 different methods of function startup and enabling their execution in many different execution environments. Without the sake of completeness, we believe that those methods can constitute a valid subset enabling flexible deployment of function on resources of the cloud continuum spanning from edge devices to big data centers.

The execution environment on which the functions are put into execution depends on the specific use case in which APERTO is being applied. At the moment of writing APERTO FaaS supports three different methods of function startup: i) the Dynamically Loaded Function(DLF), the WASM Function, and the Function Spawn(FSpawn). It is worth mentioning that thanks to the transparency introduced by the MOM and modern techniques of process isolation such as WASM, Linux Containers or Virtual Machine, it's possible to concurrently execute the different versions of the invoker exploiting different process instantiation methods on the same node and the resources available on the cloud continuum.

6.3.1 DLF (Dynamically Loaded Function)

We based our DLF mechanism on the *dynamic library loading* technique. This is generally used to combine several functions into a single unit shared by multiple processes at run-time, thus saving disk space and RAM. Although the

library code can be used by multiple processes at the same time, its variables remain isolated. Our DLF employs the POSIX standard APIs to handle the dynamic library loading [89]. When a function invocation occurs, Invoker opens (`dlopen`) the requested shared object file and, subsequently, loads the symbol (`dlsym`) related to the main library entry. For this to happen, the application developer must expose the function within the library with the name and arguments we expect. Consequently, the loaded function is first executed and finally unloaded (`dlclose`) after its termination. Due to the mechanism involved, this invocation method is suitable for executing functions with high performance and strict requirements, thus primarily aimed at meeting latency-sensitive application needs level [90].

6.3.2 WASMF (WASM Function)

The WASM invocation method is similar to DLF, as it adopts the same underlying loading mechanism. The TEMPOS invoker integrates a complete WASM engine, i.e., using the Wasmer library [91], initialized in the startup phase. When Invoker receives a request, the engine dynamically loads the shared library containing the requested function. To ensure a correct loading, the library must be compiled using a WASM code generator that translates a target-independent intermediate representation into executable machine code, e.g., Cranelift [92]. Once the function is loaded and executed, the engine removes the WASM code on its internal store. An advantage of this invocation method is the possibility of the application developers implementing functions in the programming language of their choice, still providing good results in terms of performance and levels of quality.

6.3.3 FSpawn (Function Spawn)

The last invocation mechanism, called Function Spawn (FS), follows the classic Unix idiom of `fork()` followed by `exec()` to execute a different program in a child process. If the invoker is deployed on a node supporting the `posix_spawn` API, the latter is used instead of the `fork()` and `exec()` scheme to achieve better performance in case of parent process with a larger size or memory layout [93]. Due to the flexibility and standard nature of the mechanism employed in this invocation method, it is possible to execute the function in arbitrary environments. In particular, as a first implementation, we leveraged FSpawn to execute functions, deployed in the form of an executable program, directly as a Linux user-space process. Alternatively, a function can be spawned inside an already started Docker container providing all the dependencies needed to execute the code.

6.4 Controller and Management Layer

In APERTO FaaS, thanks to the decoupling of architectural components provided by the MOM we opted to create a series of controllers, each one specifically designed to solve a single task. These processes can then be opportunistically and dynamically attached to the infrastructure by subscribing as publishers and subscribers to configuration queues.

In fact, our system can work in a static behavior without any controller process active by simply providing statics configurations in a queue. At the moment of writing, 3 controllers have been designed and implemented to support APERTO FaaS: the *Static Controller*, the *HTTP Controller*, and the *Node Configurator*. The Static Controller is a simple process that once activated reads a JSON or TOML

configuration file, requires the MOM the creation of queue described in the configuration, and writes architectural component configurations in the queue to then terminate its execution. This Controller was essentially created to help us in the configuration phase of those experimental settings requiring a static behavior of the platform without any scaling or reconfiguration action. The HTTP Controller, instead, after its startup activates an HTTP endpoint exposing REST API to configure APERTO FaaS dynamically while it is running. In particular, the HTTP controller accepts POST requests having JSON structured body and indicating configurations for elements or describing a FaaS workflow.

Finally, the Node Configurator is a process that helps the invoker in configuring the executor node in order to properly activate a workflow. Its duties can vary depending on the model of invoker adopted but actually implements: i) a download process to make available at a specific path the code of the function needed by a workflow, ii) a scaling process in charge of instantiating or destroying invoker process in the case of single thread model, a monitoring process in charge of watching node resource availability and signal to the controller, through the queue, the necessity of scaling resources associated with a specific workflow.

6.5 Data Persistence Layer

To enable uniform and performant data operativity from the function we create SPS as a process running inside the same execution context of the invoker component. While the strong decoupling and transparency guaranteed by the MoM can potentially enable the simultaneous exploitation of different invoker models and execution environments, the current implementation of SPS adopts a Docker-based *invoker per function* architectural pattern. This choice is mainly

driven by the simplicity that this model brings to the management and orchestration plane, enabling us to exploit already consolidated mechanisms for isolation, multi-tenancy, and container orchestration. Without loss of generality, the current proposal can be easily deployed over a *Kubernetes* cluster as a standard container and could leverage on *Containerd* and *Kubernetes* environment variables and secret management features.

From the customer perspective, the activation of a workflow requiring to operate on a data store requires only that the user indicate eventual data stores the function can access and perform operations. When the controller receives the configuration, it checks if data support is enabled for that workflow and thereafter creates a container embedding the business logic and the invoker-specific implementation for the desired data store. Finally, the controller configures the trigger and MoM to activate the workflows and distribute the packed function over to suitable nodes.

Once the workflow activation phase is terminated the specialized invoker opens receiving an ingress event, starts a new instance of the function, opens a bidirectional communication channel through stdin/out, and pipes the events to the function (22 step 1). The invoker then waits either for a message of termination containing the result of the execution of the function(22 step 6) or a specialized message containing information to execute an operation on the configured data store (22 step 2). If the invoker receives a data store operation, it unpacks and deserializes the message through the specific datastore driver and executes the desired operation (22 step 3 and 4). Upon termination, the result of the operation is serialized and returned to the function (22 step 5).

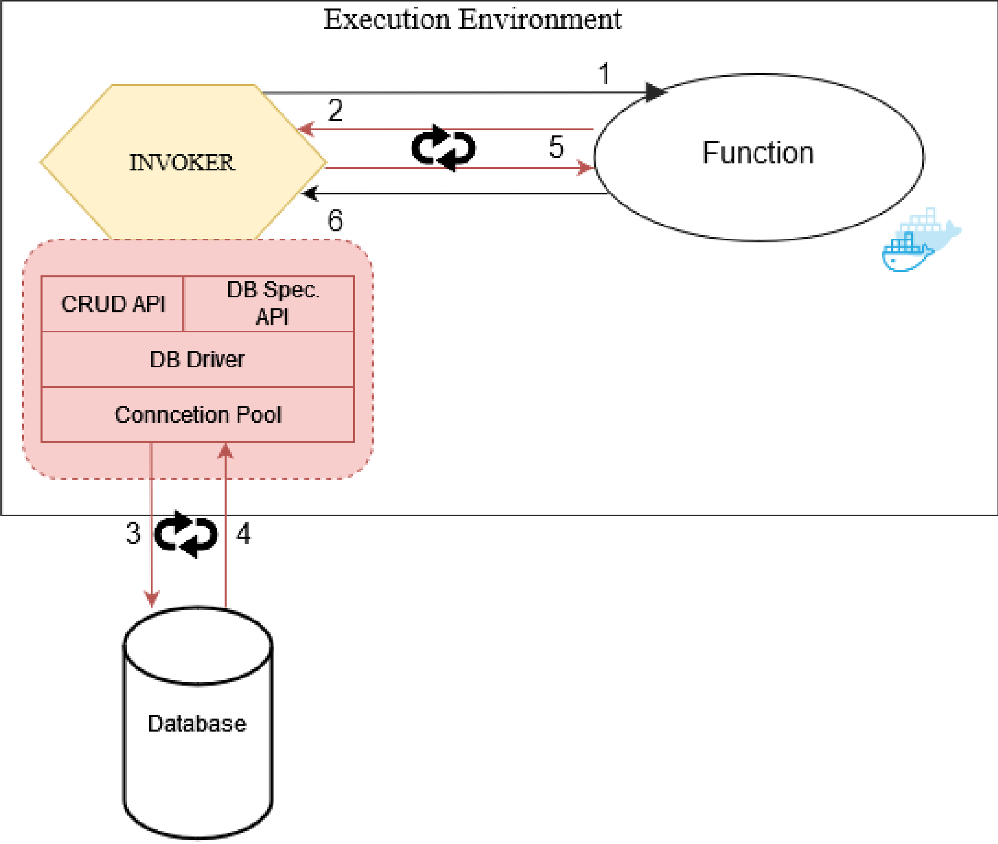


Figure 22: High-level architecture of SPS showing the interaction sequence from the function activation to the query of data storage and ending with function termination and return of result

During this last step, the SPS layer can take advantage of optimizations made in the invoker data access mechanism. In particular, the current implementation provides a **connection pooling** mechanism, allowing to retain persistent connection(s) to target databases and a (de)queuing mechanism that allows amortizing connection creation cost over multiple requests. It is important to note that business logic might require an arbitrary number of data operations during its lifetime.

The strong decoupling between the invoker and the functions it services allows amortizing the cost of development of a specific invoker for each datastore solution while providing a desirable degree of flexibility. In particular, in order

to offer the wider possible compatibility with existing languages and frameworks, we opted to implement invoker-function communication through *stdin* allowing asynchronous exchange. The (de)serialization process is handled *msgpack*, allowing us to achieve cross-language representation with a reduced overhead in both message size and computation time [94].

6.6 Authorization Layer

The Authorization layer verifies the rights associated with an event received by the invoker from the MOM before activating the function in charge of processing the event. In order to associate a set of rules with a specific workflow the customer uploads to the controller a set of rules referencing the id of the workflow specified. For each configuration received, the controller pushes rules in special topics. By following this approach and exploiting the communication capabilities provided by the MOM, different components of the architecture will be able to independently read them. Rules are then uploaded by the controller, through HTTP requests, directly to the master registry.

Finally, the invoker waits for events incoming from a specific topic. At each event arrival, the invoker: (i) deserializes it and extracts the fields necessary to issue the query to the rule engine, and (ii) if the rule engine evaluation is successful, it executes the code specified by the customer through the event provided as input argument, otherwise, an error is returned.

In the proposed architecture, the customer defines a set of policies described through a specific policy language named Rego[4]. For the sake of clarity, in the

[4]https://www.openpolicyagent.org/docs/latest/policy-language

snippet below, we report a simple example of a policy. In this example, a user will be granted to use a feature only if her/his role is authorized to invoke that function.

```
allow{
  some grant
  user_is_granted[grant]
  input.fnInvoked == grant.functions[i]
}
user_is_granted[grant]{
  some i, j
  role  := data.user_roles[input.user][i]
  grant := data.role_grants[role][j]
}
```

Listing 1: A Rego policy example granting access according to user roles.

As our rule engine, we employed Open Policy Agent (OPA) [5], a lightweight general-purpose policy engine service that decouples policy decision-making from policy enforcement.

To efficiently manage the distribution and update of policies while achieving the best performance in terms of fault tolerance and high availability, the registry was implemented as two-levels storage.

The first level is realized through CouchDB, an open-source document-oriented NoSQL database that exploits a Multi-Version Concurrency Control (MVCC) protocol to enable the synchronization and replication of documents over one or more instances to maintain *eventual consistency*, also supporting offline replication. It is the reference central point responsible for managing the consistency and persistence of access control policies, providing a global perspective of authorization management. The second level consists of caches that minimize the latency of accessing policies and data. It is implemented through the cache offered by OPA.

[5] https://www.openpolicyagent.org

A local process called *bundle server* subscribes to changes of access control rules memorized in CouchDB and consequently updates the cache. In this proposal, the bundle server represents the adaptation mechanism of our solution that enables the rule engine to transparently use multiple policy management systems.

This model of policy distribution enables us to achieve high availability and fault tolerance. Co-locating a replica of the rule engine, data, and policies, on each execcutor node, improves performance and makes each executor node totally independent from the others. It is worth outlining that in this approach, we embrace *eventual consistency* in favor of higher availability and reduced latency in accessing policies. In the future, as specified in Section 7.6, we plan to explore other consistency models.

6.7 End-to-end QoS Service differentiation

While preserving the general aspect of our study, and without loss of generality, in this section we present the current status of the implementation of a TEMPOS prototype, which primarily exploits Linux real-time scheduling, differentiated MOM priorities, and Time-Sensitive Networking (TSN) as the underlying system-level mechanisms to enforce the QoS-aware TEMPOS abstractions (TEMPOS QoS-aware topics) for QoS management. The section is structured by presenting how we realized the TEMPOS architecture described in Section 5.7, first focusing on the *QoS Level* and then on the *System Level* (Fig. 23).

The current implementation of the TEMPOS middleware provides application developers with two distinct QoS levels. On the one hand, we define a *Best-effort Quality (BQ)* and assume its use in case of communication and function invocation with no strong latency and jitter constraints. On the other hand, we

specify the *Strict Quality (SQ)* to support the execution of functions that require more stringent and soft real-time QoS.

For the initialization of a QoS-aware workflow, the application developer/deployer must simply provide a specific configuration file, currently based on the TOML configuration file format, containing: i) all the information needed by the Controller to interact with all other entities, i.e., MOM, Triggers, and Invokers, and ii) the specification of the requested QoS levels for the connection between components and function execution at each targeted node. Then, at the end of the configuration phase, the Controller waits for reconfiguration/management requests from the developer/deployer, thus making both the Controller and the entire middleware reconfigurable and modifiable at run-time. The current implementation exposes the Controller functionality through REST APIs.

In particular, every time the Controller receives a request, it performs the possibly needed reconfigurations by interacting with the entities involved in each layer. The latter, in turn, exposes specific management interfaces and manages these configuration requests in an ad-hoc process outside the interactions of the TEMPOS workflows defined by the developer/deployer. Finally, the Controller maintains an internal representation of all TEMPOS components, which is updated with each request, thus allowing a centralized view of the entire middleware deployment environment.

To enable the strong differentiation of QoS of workflows no current MOM implementation has been demonstrated to support the coordination of different prioritization mechanisms. We decide then to implement a prototype of the TEMPOS Message-oriented Middleware with two different queues, for SQ and BQ, respectively. We developed these two queues using two different network sockets and

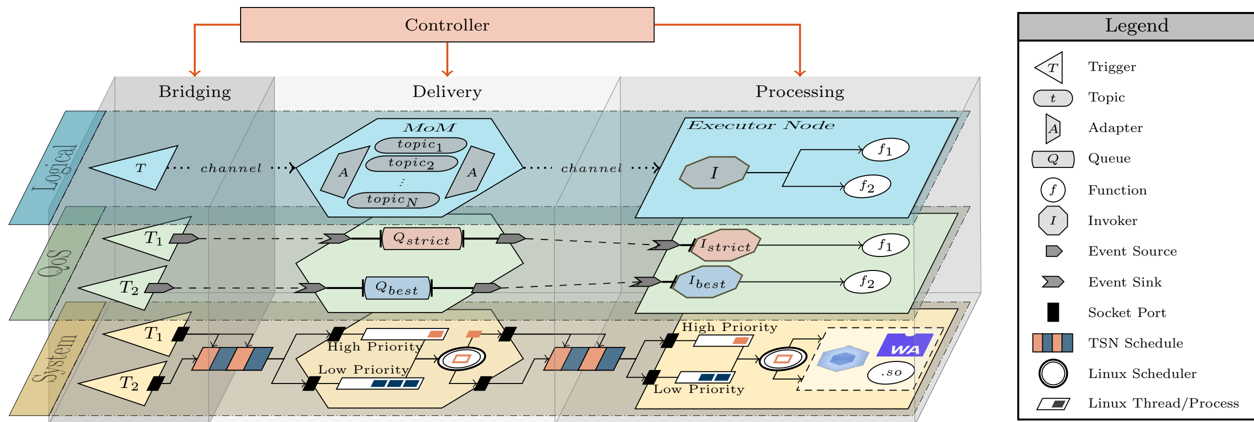


Figure 23: TEMPOS spanning different logical levels and orchestrating physical resources and component configuration in order to achieve QoS differentiation.

two threads. The sockets separate messages into two separate queues, while each thread acts as a priority queue processor since both are scheduled according to the **real-time Linux scheduler** [95]. The first thread handles all messages labeled with strict quality and runs with a higher priority than the best-effort thread. Since the priorities provided by the scheduler range from 0 to 99, as default, we use the lowest priority (0) for BQ, while we associate the highest priority (99) with SQ. Moreover, an application developer can specify to use of the Controller to choose the type of Linux real-time scheduler, e.g., Round Robin or FIFO, and set different values for the priorities of the threads associated with the queues. That makes the MOM more flexible and, in the future, opens up to the easy introduction of additional queues with intermediate quality.

A significant aspect of our MOM is its transparency of the protocols used by the underlying network; this property is achieved thanks to the introduction of TEMPOS middleware elements called Adapters (Section 5.7) and realized via a plugin-based mechanism within the MOM. A plugin represents a set of well-defined interfaces, which specify how to: i) open a connection, i.e., create a Chan-

nel, ii) configure the QoS level of a newly created connection, iii) send messages through the Channel and iv) safely close the connection. In addition, the association between one or more channels connected to the MOM and one of the queue processors realizes the concept that we name TEMPOS Topic. Since the TEMPOS components, including the MOM, are entirely implemented using a compiled language such as Rust, we based the plugin system on the dynamic library loading mechanism [96]. The realized plugins must be compiled and distributed as shared libraries, which are then loaded at run-time by the MOM according to the configuration received from the Controller.

At the time of writing, we completed the implementation of a TSN-based plug-in. Specifically, we based our implementation on the IEEE 802.1Qbv standard, which aims to support the combination of best-effort and **real-time traffic** within TSN networks [97]. The standard presents the notion of time-triggered communication windows, often called time-aware traffic windows, thus defining a mechanism to support different types of time-critical flows. In practice, a window is divided into multiple time slots associated with selected traffic classes and repeated cyclically. This makes it possible to minimize the interference of best-effort traffic with priority traffic (i.e., real-time traffic), which we refer to as strict communication QoS level. This mechanism is achieved by inserting a so-called guard band before the scheduled traffic window, which forces the buffering of packets belonging to traffic classes not to be transmitted. From an implementation perspective, windows and slots are expressed through a Gate Control List (GCL) that identifies the time instants in which packets can be transmitted on the medium [98].

To achieve a strong QoS end-to-end differentiation we have to recall that the trigger is a critical point of the architecture, creating the first differentiation of flux and eventually causing contention among concurrent requests coming from the outside of the platforms. We decided then to create a strong separation for the two priorities by executing a separate process for each trigger type and each designated priority. To provide a unique implementation for the different deployment scenarios presented in Section 5.1, we developed the trigger as an always-running Linux process listening on a network socket. Thus, an event issued by a source (e.g., a sensor) is received by the trigger via the network, or if possible, taking advantage of an IPC mechanism. This can be applied to optimize communication in the case of the co-located deployment scenario. Once executed, the Trigger first receives the configuration, containing the QoS level to be used, from the Controller, and then opens a second connection, i.e., the *Channel* used to communicate with the MOM. Different from the MOM, we consider the implementation of each Trigger as limited to a single protocol, be it TSN, Wi-Fi 6, or any other protocol providing a priority-based communication mechanism. At the moment, we have completed the implementation of the co-located trigger model by exploiting TSN-based communication.

To support the prioritization of the execution of functions we extended the invoker process by introducing the concept of prioritized execution. The current implementation exploits the concurrent execution of two threads one in charge of receiving configuration from the controller and the second of receiving events and spawning functions. Even in this case, as already done for the trigger, the separation among the two levels of QoS is done by executing a separate process for each function and each QoS level. If two or more invokers are concurrently

executing on the same node, the prioritization of the execution of the function is kept thanks to the Linux real-time scheduling. By executing the trigger as a process with different priorities in the Linux real-time scheduler, functions that are children of the invoker processes inherited the same execution prioritization.

6.8 Distributed task composition over cloud continuum resources

Thanks to DIFFUSE, APERTO FaaS is able to provide advanced function composition capabilities over resources in the cloud continuum. As already mentioned, our solution doesn't rely on a central coordinator for the composition of the single function but exploits MOM pub/sub decoupling and decentralization of the composition process that enables the **direct invocation** of the following function in the composition from the execution context of the function. From a customer perspective, composition workflows are shipped seamlessly to normal workflow through the controller REST API. Currently, the association rules are shipped to the controller in a JSON-based format and allow the definition of generic, graph-shaped processing pipelines whereby the pipeline continuation is determined by the output of the executed function. Once processed the workflow definition the controller configures through the specific queue the executor environment and the MOM by creating needed association rules and queues. Each invoker configured to run a function included in the workflow then receives a rule associating the execution of the function with a topic on which the result of this execution is published in order to trigger the following function in the composition. To introduce a level of flexibility in diffuse we also enabled the possibility of instrumenting multiple destination topics. In the current implementation, the decision on which topic the result is forwarded is based on a simple query evaluation on the exit code of the

function. The association between the function and the next topics is kept by the invoker in a hash table data structure for fast retrieval of correspondence.

To enable the invoker to communicate with different MOM middleware at the same time we implemented an adaptation layer exposing the standard primitives of the Pub/Sub model and remapping on specific API implementation opportunistically bundled with the invoker process as libraries.

In the following, we present the design of a distributed shared memory middleware, allowing us to exploit modern hardware, guaranteeing low latency and high-bandwidth communication. Next, we discuss the other MoM alternatives and overall characteristics, adding to the deployment spectrum of our platform. To support efficient function-to-function communication and thus reduce workflow execution latency we present a zero-copy transfer mechanism for use in distributed multi-host deployments. The Distributed Shared-Memory Queue (DSMQueue) exploits modern networking hardware to build a **zero-copy**, $delete-after-read$ data transfer mechanism embodying a similar semantic to its local counterpart, the Linux kernel *mqueue* primitive. This approach enables a transparent load balancing mechanism among subscribers (Invokers), whereby a read operation triggers the message removal from the queue. This way, the same function execution request is never executed more than once.

More in detail, DSMQueue is a distributed queue that exposes a *push* (send) and a *pop* (receive) operation. This queue, currently implemented as a ring buffer replicated among a group of processes (shared state), may be configured to offer different semantics, such as FIFO (default) or LIFO. The send/receive operations can be either blocking or unblocking. In a blocking configuration, when the queue is full, the process issuing a send (push) goes into a blocking state; when the queue

is empty, the process issuing a receive (pop) blocks on it. The specific configuration depends on the scenario requirements. For example, in a context where data freshness is important, an unblocking FIFO configuration allows the sender to overwrite the data in the buffer according to a specific queue management policy.

Considering we are dealing with a shared state in a distributed environment, one needs to rely on synchronization primitives to preserve consistency. To address this issue, we decided to adopt a well-known model for state sharing: the State Machine Replication (SMR) [99]. In SMR, every group member - the queue instance associated with processes representing the Invokers - holds a replica of the state, and all are bound to apply the same operations in the same order to maintain overall consistency. Guaranteeing a strong consistency model requires implementing a form of *atomic multicast*, which imposes that any message sent by any group member is broadcasted to the others and delivered in the same order to all the members (a total ordering) even in case of failures.

To fulfill all the above requirements, we implemented our solution on top of the Derecho open-source library [100]. The library enables point-to-point and multicast communication and supports total ordering, failure atomicity, and optional durable message logging. An optimal hardware mapping for RDMA enables Derecho to efficiently support even the strongest consistency properties, such as the SMR model while guaranteeing high performance in terms of data throughput and latency. At the same time, to preserve compatibility when suitable hardware is not available, Derecho can execute on top of the TCP/IP stack without requiring any modifications to existing applications.

More in detail, Derecho allows users to define distributed services as “replicated objects”, i.e., a set of state variables having an associated set of operations.

Table 1: MoM technologies with respective Delivery semantic, Delivery Order, and Load Balancing capabilities.

	Delivery semantic	Delivery Order	Load Balancing
Kafka	Exactly-Once, At-Least-Once, At-Most-Once	Within single partition total ordering	Producer-side: static, Round-Robin
Redis	At-Least-Once At-Most-Once	Total Ordering	Consumer-Side: First requesting First served
DQueue	Exactly-Once	Total Ordering	Consumer-Side: First requesting First served

Processes holding replicas of such an object will form a process group. Each state update can then be forwarded as an *atomic multicast* to all the group members and performed by all replicas. On an RDMA network, Derecho offers a zero-copy, lock-free critical path among remote applications, leading to ultra-low latency and exceptionally high bandwidth utilization. DSMQueue builds on Derecho and provides some higher-level blocking primitives to send/receive messages to/from, e.g., Invoker components. Specifically, we implement DSMQueue as a Derecho replicated object, where the shared state is the ring buffer. The operations we define on that state are API calls that allow components to create and subscribe to specific message queues, acting as conveyors of data among components.

Adding to the deployment spectrum of our proposal, we identified two other state-of-the-art MoM solutions, namely Apache Kafka and Redis Stream [85,101]. In specific, Kafka is a highly scalable, open-source event streaming platform, while Redis Stream is a streaming abstraction built on top of the widespread persistent Redis database.

Table 1 provides a summary of some characteristics the different MoM solutions embody. All the options offer advanced state replication and consistency mechanisms for improved load distribution and fault tolerance. In particular, Kafka exploits a multi-broker mechanism with a configurable level of topic (channel) replication, while Redis employs a classical Driver-Worker active replication scheme.

Similarly, our DSMQueue proposal replicates the queue state (data) exploiting RDMA to guarantee the highest possible performance. DSMQueue, which is based on the Derecho library, adopts the same active replication pattern of Redis, but the logic is completely decentralized, thus eliminating the need for a driver node on the critical data path. In this setting, all the nodes are equal peers that agree on the same shared state, thus achieving the maximum possible degree of parallelism.

Concerning the delivery semantics, DSMQueue offers an exactly-once semantic, while Redis offers an at-least-once embodying less overhead in synchronization when compared to DSMQueue. This behavior may lead to a lower use of the network resources but does not guarantee the consistency of the shared state in case of failure of one or more nodes, which DSMQueue is always able to guarantee. Kafka is the only one of the three solutions that, thanks to its deep integration with Apache Zookeeper, allows choosing among all the three delivery semantics at most once, at least-once, and exactly once at a topic granularity.

This chapter presented the state-of-the-art implementation of Aperto FaaS demonstrating the technical feasibility of the proposed approaches. Proposed implementations and integrated technologies represent some of the most widespread approaches adopted in cloud-native environments. Trade-offs, technological gaps,

and limits encountered, while narrowed to FaaS platforms can be easily extended also to other cloud computing models. Although the market is thriving with solutions, we denote the absence of MOMs proposals specifically designed to meet the FaaS platforms' needs, which has required their implementation from scratch. The prototype of Aperto FaaS enabled us to structure an extensive set of tests, presented in the next chapter and aiming at demonstrating quantitatively the goodness of approaches adopted and technological choices.

7 Experimental assessment and Test results

This chapter presents an experimental evaluation, assessing our proposal in its main contributions. During the development of the different components of APERTO FaaS we had the opportunity to collaborate with different partners and realities in the Emilia Romagna(IT) territory providing us with different use case scenarios belonging not only to the Tourism context. The experimental testbed takes into account different scenarios and consequently different execution environments and service offer constraints spanning from high-performance centralized cloud to low-power devices into the edge.

In particular, in the first section of the chapter, we show a performance brake down of the Processing Slices comparing the performance of the three invocation methods implemented in APERTO FaaS. Then we demonstrate the effectiveness of SPS architecture by accelerating data operations with different database technologies. In the third section, we evaluate the implications of different architectural implementations of the Authorization layer and the performance benefits introduced by our solution. The fourth section demonstrates TEMPOS capabilities of prioritizing workflow execution in each of the layers of the architecture. Then, we present tests and results demonstrating how end-to-end QoS is influenced by TEMPOS differentiation. In the last section, we present an extensive evaluation of DIFFUSE function composition mechanisms tested under different loading conditions and leveraging different MOM technologies.

Table 2: Specifications of the nodes used for the processing performance evaluation testbed.

Node Tag	Model	CPU	Memory
1	Custom Workstation	AMD Ryzen 3700X 8/16 CPU	32 GB
2	Dell Optiplex 3010	Intel Core i5-3470 4/4 CPU	10 GB
3	UP Core Plus board	Intel Atom E3950 4/4 CPU	8 GB

7.1 Processing startup methods Comparison

The first testbed section is focused on how different methods of invocation and execution environments perform when run over heterogeneous hardware. To this purpose we consecutively invoked the same function (Algorithm 1), programmed in a compiled language, for 2 minutes when invoked with the mechanism of i) *DLF*, ii) *WASMF* and iii) *FSpawn*. We next repeated the test with the *FSpawn* mechanism but with two different versions of the same function implemented in two different interpreted languages, i.e., Python and JavaScript. These tests are repeated on nodes A, B, and C (Table 2) as representative of three very different cases of resource availability on edge hosts.

All the results show(Fig. 24) that i) startup and execution times are sensibly influenced by the employed hardware and ii) latency minor than $1\,\mathrm{ms}$ is easily achievable on medium-top class hardware.*DLF* with execution duration near to $100\,\mu\mathrm{s}$ qualifies as the fastest mechanism to invoke functions; this opens up to the application of TEMPOS in many challenging and latency-sensitive use cases where sub-millisecond end-to-end latency is needed; however, DLF restricts the usable programming languages to the only ones compatible with the generation of shared libraries.

Algorithm 1 Pseudo code showing the operations performed by the function used in the tests: deserialization, count of occurrence in the text, and repetitions of operations of square root and power based on the index value.

```
1: function MAIN(e : Event)                          ▷ The function entry point
2:     message, pattern ← deserialize(e)
3:     occur ← count_occurence(message, pattern)
4:     res ← 0
5:     for i ← 0, occur do
6:         if i mod 2 = 0 then
7:             res ← res+ pow(i)
8:         else
9:             res ← res+ sqrt(i)
10:        end if
11:    end for
12:    output(res)
13: end function
```

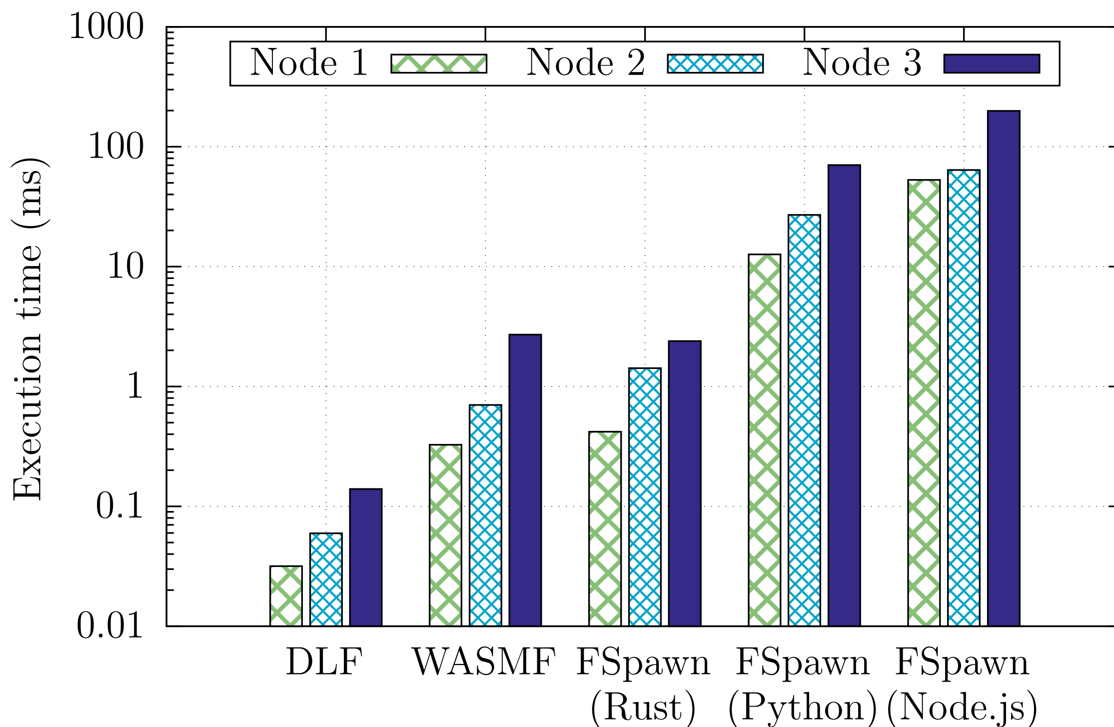


Figure 24: Mean execution times for the different invocation methods gathered in a run of 5 min. Each run repeated on nodes A, B, and C.

Table 3: Number of invocations executed by the different invocation methods during the processing test (5 min. run).

Invocation Mode	Node 1	Node 2	Node 3
DLF	1884×10^3	1004×10^3	428×10^3
WASMF	183×10^3	85×10^3	22×10^3
FSpawn (Rust)	142×10^3	42×10^3	25×10^3
FSpawn (Python)	5×10^3	2×10^3	855
FSpawn (Node.js)	1×10^3	940	303

The *FSpawn* execution, on the contrary, showed maximum flexibility, being able to run every language executable in a Linux environment. However, it exhibited the worst performance in terms of total execution time, with latency up to hundreds of milliseconds, in particular when running non-compiled languages (Figure 24). This qualifies FSpawn as a good mechanism to adopt in a FaaS platform given its flexibility, but its measured performance makes it infeasible to use in deployment scenarios where end-to-end latency needs to be below the $1\,\mathrm{ms}$ threshold. The execution through *WASMF* performed one order of magnitude worst than *DLF* and only slightly better than the execution of a compiled function with *FSpawn*, with an execution time of the order of $1\,\mathrm{ms}$. However, this mechanism showed the potentiality of sensibly reducing the execution and startup time of many non-compiled languages. Table 3 shows that the choice of the right invocation mechanisms also results from a trade-off between the freedom in implementation language selection and the number of executable functions on a given hardware infrastructure. Also, in this case, the results of this first test work also as a baseline for the successive results because the first test was conducted without concurrency among workflows.

7.2 Data Persistence Performance

Herein, we assess the performance advantages that derive from exploiting the architectural optimizations proposed in SPS. To this end, we conduct a comparison of two different approaches as follows: (i) the Native approach where the function embeds in its business logic the creation of a connection to a database and executes operations on it, and (ii) the SPS approach where data operations are mediated by the invoker.

We evaluate the different schemes under two representative synthetic workloads: (i) a constant-rate stream of requests, and (ii) a stream of incoming requests issued at an increasing rate. The first workload aims to assess the properties of the proposal in a steady regime, while the second scenario reproduces a typical traffic pattern that can represent a transitional state of a system subject to an increasing number of users requesting a particular service.

The proposed evaluations are conducted over pluggable data store layers, namely: (i) MySQL a classical, widely popular SQL relational database, and (ii) MongoDB a connection-based NoSQL document-based DB. The choice of these two candidates was made not only taking into consideration the characteristics of those solutions but also their wide diffusion on the market.

To assess the different configurations, we employ a lightweight, short-lived business logic coded in Rust with execution times of about 4 ms. Upon termination, each invoker logs into *Unix Syslog* three timestamps associated with the reception of a particular event, the issuing of an operation to the data store, and the transmission of the result. Those timestamps are later on used to compute the different metrics. Response times are measured as the time-lapse between the mo-

ment the request is issued and the moment when the result of the computation is returned to the trigger.

To better highlight the performance impact of the different approaches, we also fix the number of concurrent functions that the FaaS platform can perform in parallel, removing the time variations due to the scaling mechanism.

The experiments are conducted on four identical nodes, each equipped with a 4-core i5-3470 CPU @ 3.20GHz, 12GB RAM, running Ubuntu 20.04. One node hosts the stress script used to simulate different patterns of traffic, a second node hosts the various DBs (one at a time), and the other node hosts the FaaS platform. Once a request is issued and received by the trigger, the latter forwards it to the invokers via the MoM.

In this first experiment, we would like to investigate the system behavior under a steady regime. Hence, the trigger issues a fixed number of requests at a constant rate of 20 requests/second, and the experiment is run by varying the underlying data store support and mechanism used to access the data.

Focusing on the read performance shown in Fig 25(a), we observe that MySQL mediated by SPS configuration exhibits the best performance, while the native MySQL approach results in the worst-performing configuration. In the case of MongoDB, the introduction of SPS optimization accelerates not only the function execution but also the data access metric component. When comparing the performance of the Read to the Create one, the scenario changes significantly. MySQL, as expected, has the worst performance and this is to be attributed to its transactional nature and ACID properties. Even in this configuration, the benefits introduced by SPS optimizations are considerable, obtaining a 28% reduction of end latency in the case of MySQL and a 55% reduction for MongoDB.

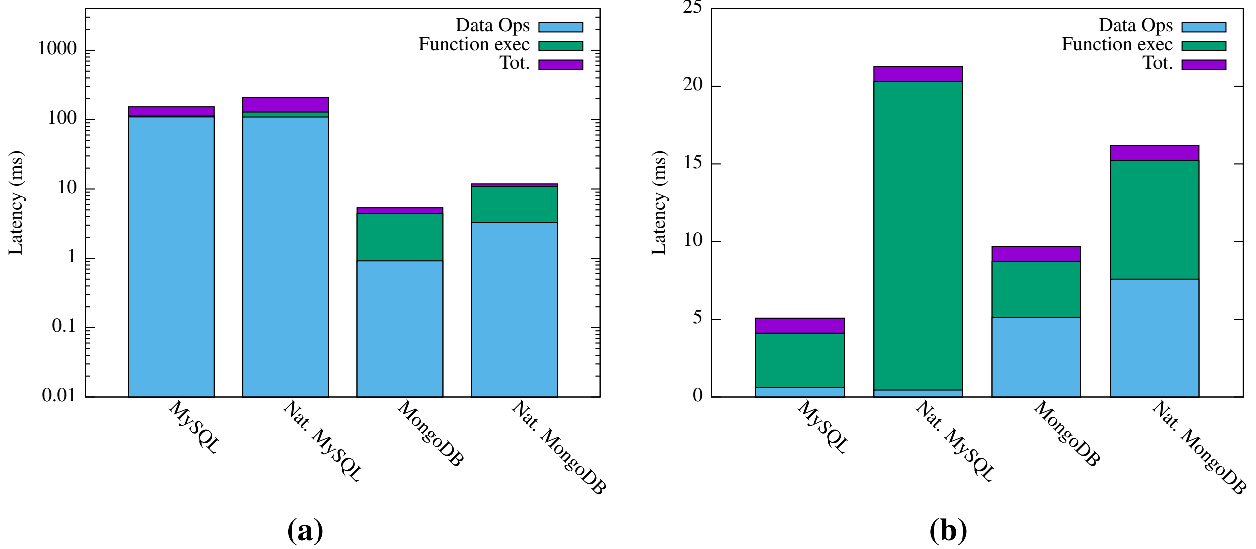


Figure 25: Average end-to-end response latency, function execution time, and database operation latency of the various configurations. The system is subjected to a synthetic, constant load of 20 requests/second, assessing the performance of an (a) read operation and a (b) create operation (log scale).

In this experiment, we aim to showcase that our solution not only guarantees better use of computational resources but also that the connection reuse feature, when employed, can contribute to an improved workload throughput and better parallelism in data store interactions.

Fig. 26 shows the performance comparison of the different configurations. For both the Read and Create operations, the benefits of SPS are evident; overall, the system can digest the load gracefully, working at a faster pace when compared to the Native approach. This behavior becomes evident in the Create operation, where SPS-mediated access is capable of absorbing 8 times the load of the Native configuration while still achieving lower end-to-end latency. In the Read operation, the SPS-mediated MySQL services almost 4 times more concurrent requests before exhibiting a performance degradation, while SPS-mediated MongoDB can

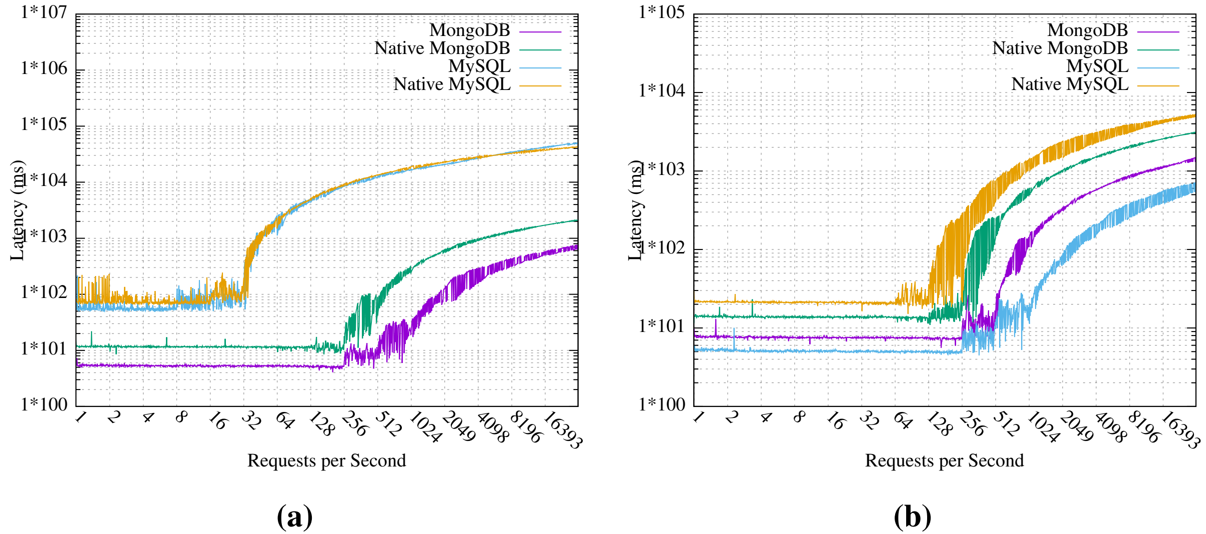


Figure 26: Log scale representation of the end-to-end response latency of the different solutions assessed when subjecting the system to an increasing rate of (a) read operation and a (b) create operation requests starting from 1 and reaching 16400.

tolerate almost 2 times more concurrent requests. Overall, SPS performance exhibits lower variability when compared to the Native solution.

7.3 Authorization Performance

In order to assess the applicability of the proposed architecture in serverless environments, we tested it in different load conditions. All experiments were conducted on a local cluster composed of 6 nodes equipped with an Intel(R) Core(TM) i5-3470 CPU running at 3.20GHz and 12 GB of RAM. The node in charge of submitting requests was configured to linearly generate from 0 to 1000 concurrent requests in 300 seconds to obtain more samples under the different load conditions. To compare the performance of the decentralized approach with that of the centralized one, the cluster was configured in both modes. In the centralized de-

ployment, we have a single node that hosts the trigger, the controller, and the rule engine, while the invokers and the functions are deployed on each of the remaining nodes. On the other hand, as far as the decentralized design is concerned, the trigger and the controller are hosted on the same node, while the invokers, the functions, and the rule engine are deployed on each of the remaining nodes.

7.3.1 Component Performance

We first evaluated in which component (trigger, invoker, and function) of the architecture the authorization verification should be integrated. This experiment was only conducted with the centralized approach due to the impracticability of decentralizing authorization in the trigger that would have significantly favored other components. In Figure 27, for each of the components under study, we report the trend of end-to-end latency as the number of requests increases. In order to evaluate as many scenarios as possible, we considered both authorized (Figure 27(a)) and unauthorized (Figure 27(b)) requests.

Results show that the novel approach proposed for enforcing access control verification in the invoker performs better than the others for both authorized and unauthorized requests. Verifying access control at this level prevents the actual function invocation if the request is not authorized. Moreover, this component will not be susceptible to congestion because our architecture foresees more instances. This choice enables instantiating resources proportionally to the effective number of requests as demonstrated by the graphs where its end-to-end latency scales gracefully. As well as for the invoker, verifying policies at the trigger level avoids instantiating a function for requests that will be denied. Indeed, it per-

forms quite well for unauthorized requests even though its performance tends to get worse as the number of requests increases.

Since every type of external event is managed by a different trigger, integrating verification inside this component requires the replication of authorization mechanisms on each trigger type. Thus, each trigger will constitute a bottleneck for the architecture since triggers are not replicated. Furthermore, this approach is not adequate for function invocations where the trigger is not directly involved, such as in function-to-function communications. Despite we have already mentioned the disadvantages of verifying access control within the source code and the reasons for not using it, for the sake of completeness, we also evaluated the access control verification at the function level. By adopting this approach, even though the request may be denied, the access rights will be evaluated only after the function has been instantiated resulting in additional network latency and higher billing.

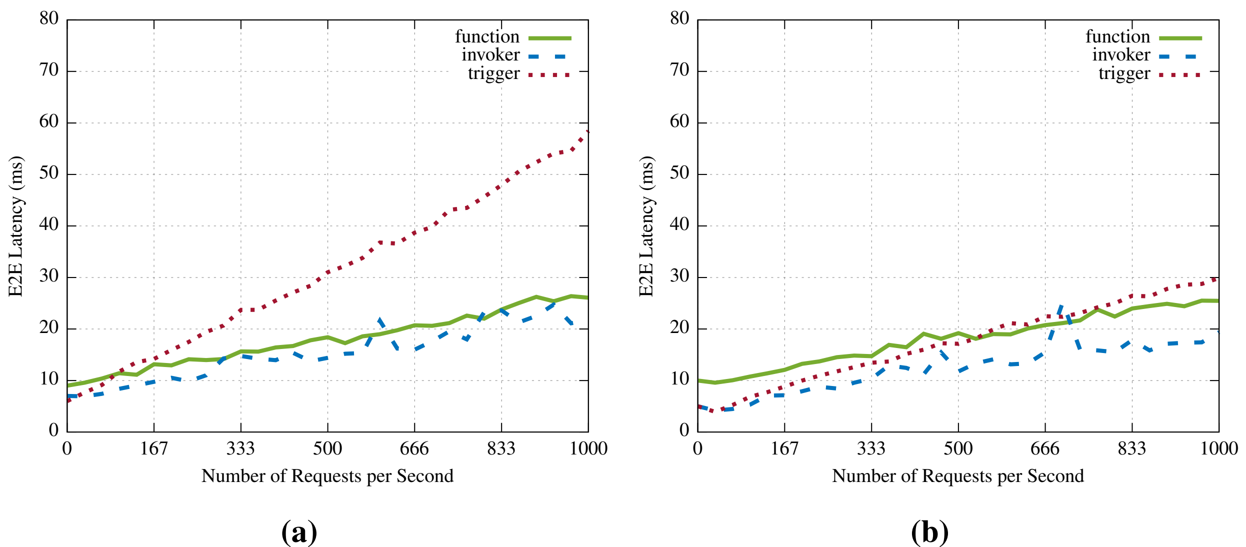


Figure 27: End-to-end latency of access verification at the (i) trigger, (ii) invoker, and (iii) function level when the requests are granted (a) and denied (b).

7.3.2 Decentralized Performance vs Centralized Performance

Then, we compared the performance of the decentralized approach with that of the centralized one to evaluate if our architecture can be actually adopted. In Figure 28, we report, for both designs, the trends of end-to-end latency as the number of requests increases and scaling the number of the executor nodes. With a low number of nodes employed for executing functions, the performance of the two approaches is almost indistinguishable, while with only one executor node, the centralized deployment performs slightly better. However, with about 700 requests per second, the centralized design requires more than 50 milliseconds to satisfy the requests. This exponential increase in response latency, shown in Figure 28, is caused by the increasing queuing of requests due to resource saturation. On the other hand, as the number of executor nodes increases, the decentralized approach starts performing better. This can be clearly observed with four executor nodes. When having available more than one node for executing functions, the decentralized approach should be always adopted since it guarantees to achieve better performance.

7.3.3 Policy Performance

Finally, we tested both approaches using policies progressively more complex. Indeed, according to the system and the use case scenario, policies can be really different from each other. In complex systems which involve many users, several organizations, and dynamic environmental factors, policies tend to be much more complex compared to those employed to address authorization in small networks. As a matter of fact, the time needed to evaluate policies is proportional to their complexity. Hence, to evaluate the potential applicability of our proposal to

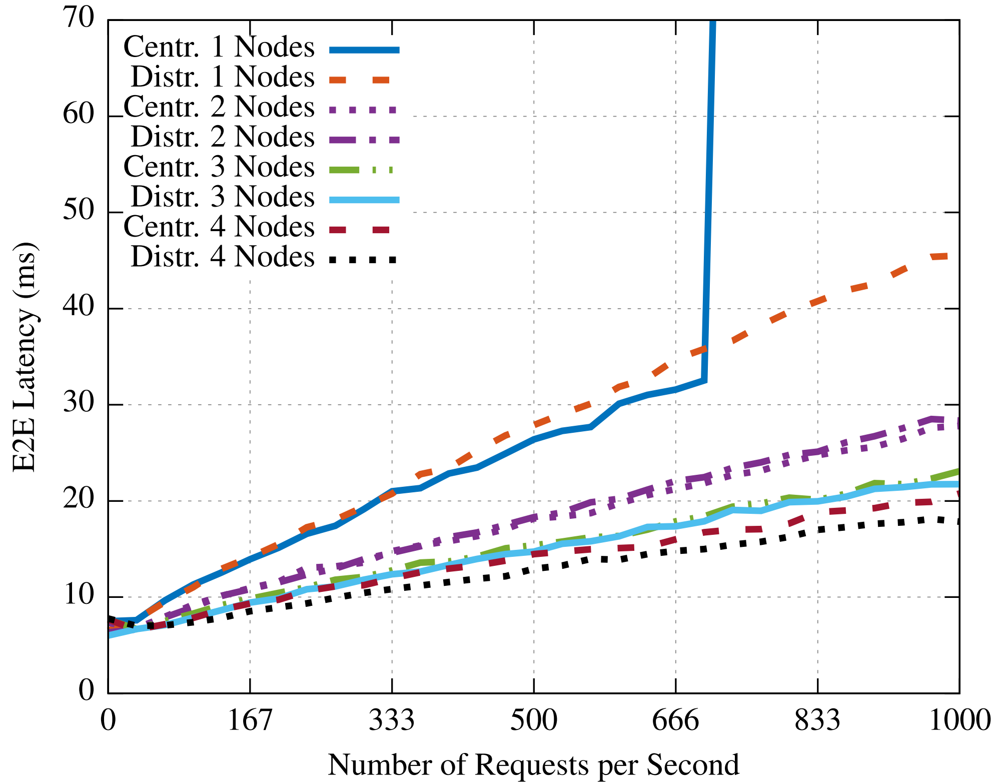


Figure 28: Comparison of end-to-end latency performance between centralized and decentralized access verification when scaling the number of executor nodes from 1 to 4.

actual scenarios, we tested our architecture against increasingly complex policies classified as *Easy*, *Mid*, and *Complex*. The policy reported in Listing 1 is the Easy policy employed in our evaluation. Mid and Complex policies were obtained by complicating the Easy one with respectively 10 and 25 random integer comparisons.

As shown in Figure 29, our decentralized approach outperforms centralized ones already when evaluating simple policies. The performance difference between the two approaches becomes more and more evident with the increase in rule complexity. This can be observed when evaluating complex policies where the end-to-end latency of our decentralized proposal reaches about two orders of

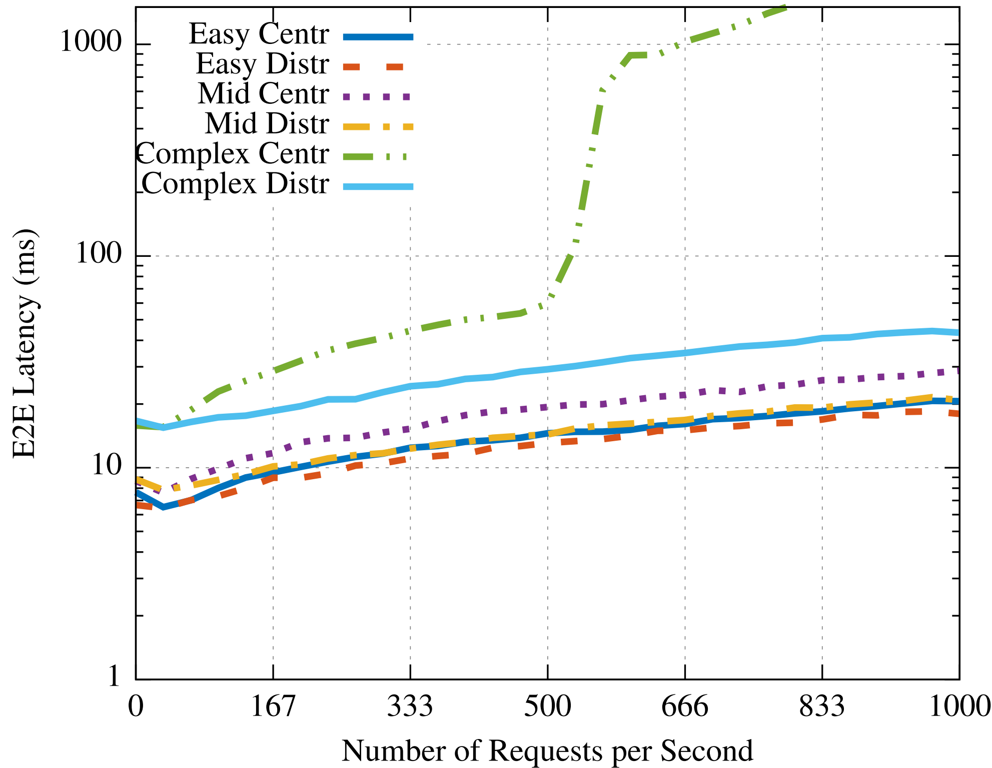


Figure 29: Comparison of end-to-end latency performance on the *log scale* between centralized and decentralized access control verification of increasingly complex policies.

magnitude lower than its centralized counterpart due to network congestion related to access control verification.

7.4 QoS Service differentiation evaluation

To quantitatively evaluate and validate the effectiveness of TEMPOS, we developed a series of testbeds to analyze the behaviors of several of its primary components. The tests described below aim at demonstrating that the TEMPOS middleware can support differentiated end-to-end QoS levels while offering application developers a simplified interaction and instrumentation interface. Special

attention was given to demonstrating the ability of TEMPOS to orchestrate and compose different mechanisms to achieve highly differentiated QoS for different workflows. Nevertheless noteworthy, these testbeds also show the validity and feasibility of the achieved TEMPOS implementation stack under very different load conditions.

Table 4: Specifications of the nodes used for the evaluation testbed.

Node Tag	Model	CPU	Memory	TSN driver
A	Custom Workstation	AMD Ryzen 3700X 8/16 CPU	32 GB	1 × Intel I211
B	Dell Optiplex 3010	Intel Core i5-3470 4/4 CPU	10 GB	-
C	UP Core Plus board	Intel Atom E3950 4/4 CPU	8 GB	4 × Intel I210
D,E	UP Xtreme board	Intel Core i3-8145UE 2/4 CPU	8 GB	4 × Intel I210

Of course, the performance results achievable by TEMPOS in absolute terms depend on the characteristics of resources in the targeted deployment environment. Therefore, the following series of testbeds can also constitute the first step to calibrating resources in target deployment scenarios with similar technological stacks. Our testbeds are designed to simulate a worst-case where the number of concurrent requests is putting under stress the TEMPOS middleware. In particular, we organized our testbeds in three cases, aiming each one to stress alternatively the event-delivery process, the event processing, or the overall middleware.

In the first case, we test the behavior of TEMPOS event delivery under different load conditions, thus emulating diverse resource competition scenarios of workflows. The specific goal is to demonstrate the ability of our middleware to chain different QoS mechanisms while maintaining guarantees about latency and jitter. The second case aims at demonstrating the TEMPOS ability to hide heterogeneity while still providing a strong differentiation of QoS. For this reason, in this testbed case, we trigger the execution of a complex and computationally heavy function, representative of many common workloads (Algorithm 1), while

employing all the different function invocation methods currently supported in our TEMPOS prototype.

In the last testbed case, instead, we aim at verifying the ability of TEMPOS of composing mechanisms for QoS at different TEMPOS slices, to achieve configurable and complete end-to-end QoS over different workflows.

We have implemented and configured two workflows, invoking the same function (Algorithm 1) and configured one with BQ level and the other with SQ. All tests foresee an increasing number of requests for each workflow, to show the TEMPOS behavior in the presence of challenging dynamicity in the supported service load. The reported results are discussed and analyzed by presenting the overhead quotas introduced by single TEMPOS components.

To assess and validate TEMPOS feasibility over edge cloud deployment environments, we have decided to conduct our test on nodes with limited computational resources (Table 4). As the edge hosts, we have employed three TSN-enabled nodes. In particular, Node A and Node B have been introduced and exploited only during the second testbed case to verify how invocation methods performance would variate in correlation with node performance.

We opted to co-locate Triggers and data Producers on Node E, thus emulating a practical case where two edge nodes cannot communicate by employing differentiated QoS mechanisms. The choice of assigning one of the two resource-rich nodes to these TEMPOS components is mainly due to the need to generate high and precise loads to stress our middleware. The second most performant board, node D, hosts an instance of the TEMPOS MOM. In addition, we decided to deploy all the invokers on the node with fewer resources to emphasize concurrent

resource requests and potential QoS conflicts in the processing slice, which is a practically recurrent situation.

The three nodes of our testbed are connected through a Relyum RELY-TSN-BRIDGE Ethernet switch, configured with a TSN setup realizing differentiated QoS channels for the ingress/egress of topics. To set up the selected QoS mechanism for TEMPOS best effort and strict effort levels, we use two new *qdiscs* queuing disciplines built into the Linux kernel: i) *taprio* (Time-Aware Priority Shaper) implementing a simplified version of the scheduling defined by IEEE 802.1Qbv and ii) *etf* (Earliest TxTime First) qdisc that allows applications to set a transmission time for each packet (this information is then used by the scheduler to de-queue the packet and forward it over TSN). Note that applications based on the IEEE 802.1Qbv standard must rely on a single time reference: to this purpose, an autonomous standard called IEEE 802.1AS is specified in the TSN context, which defines a specific profile of the IEEE 1588 standard by extending the Precision Time Protocol (PTP). This extension, called generic Precision Time Protocol (gPTP), defines two main entities, namely the Clock Master (CM) and the Clock Slave (CS), associated with the devices in the network [97].

About our testbed synchronization, each TEMPOS node participates to elect a controlling entity, determined by the Best Master Clock Algorithm (BMCA): this controlling node is called the PTP grandmaster [98]; the grandmaster sends clock information to each of the Clock Slaves connected to it; once all TEMPOS devices are synchronized, we have what is effectively a time-aware network of nodes, i.e., a ready-to-use gPTP domain.

In our testbed, we created two time-aware TSN windows of $1\,\mathrm{ms}$, i.e., between Trigger and the MOM, and between the MOM and Invoker. Each window

is divided into two time slots, one for SQ and one for BQ, each of $500\,\mu s$; the first SQ slot is scheduled in the first half of the first window, where the second SQ slot is skewed of $300\,\mu s$ concerning the starting time of the MOM-invoker window; this configuration enables strict TEMPOS traffic to find the gate open at each step, with no additional delays. Finally, to gather monitoring statistics and to evaluate TEMPOS performance, in each TEMPOS component in the testbed we introduced the logging of any received event id, associated with its synchronized timestamp; those logs are collected and analyzed only offline at the end of tests, not to perturb the performance of workflow execution.

7.4.1 Event Delivery QoS Differentiation

In the first testbed case, we aim at demonstrating the TEMPOS ability to prioritize event delivery based on workflow QoS. In particular, in the first test, we submitted a constant rate of 1000 events per second to Trigger for a time-lapse of 5 minutes. Then, we measured the difference between the timestamp corresponding to event creation at Trigger and the one reported at its delivery. We alternate the activation of SQ and BQ workflows to observe the behaviors of the two in a scenario with no perturbation due to concurrency. The results in Figure 30 show that the events belonging to the SQ workflow are characterized by a lower end-to-end latency and jitter when compared with those of the BQ workflow.

In particular, end-to-end latency for SQ workflow events settles to $501\,\mu s$ on average, thus showing that TEMPOS is compatible with very challenging contexts that call for less than 1 ms response time, like soft real-time ones.

Note that a clear differentiation between TEMPOS QoS levels is possible thanks to the combined exploitation of prioritization mechanisms acting at the net-

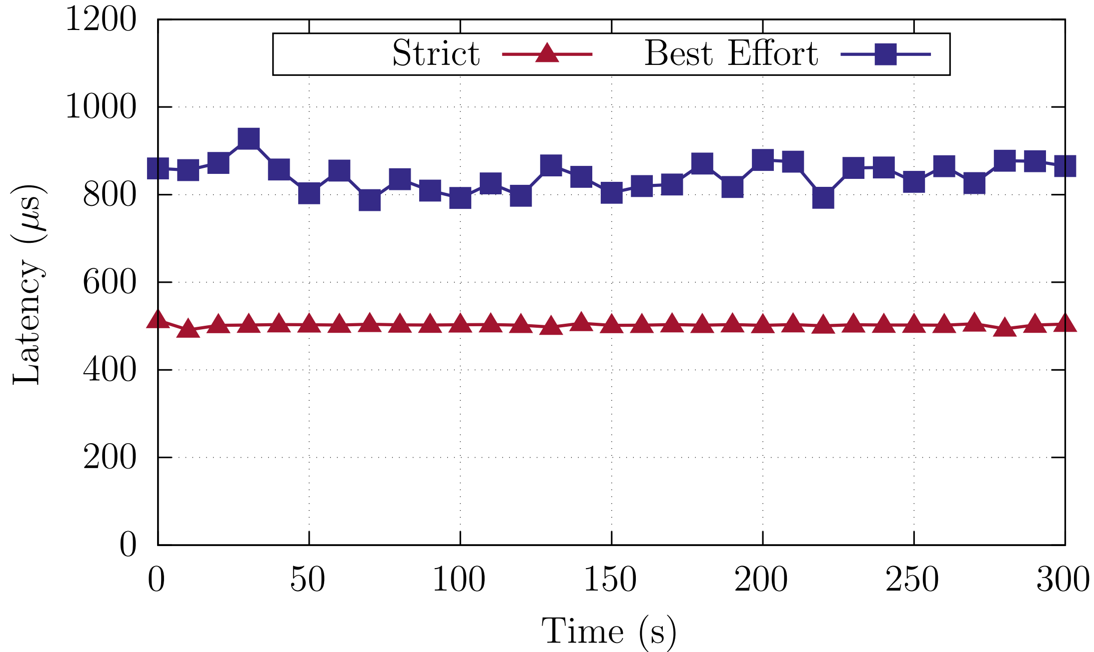


Figure 30: First testbed section showing performance of Delivery slice. Average end-to-end latency of best effort and Strict effort traffic when executed in separated environments.

work layer and the event processing layer. In particular, the lower jitter is mainly due to the strict scheduling of events and the synchronization of TSN windows in ingress and egress of the topic. In fact, the maximum latency that we measured throughout all tests for each hop is 223 µs for the delivery of one event to the TEMPOS MOM, 57 µs for event processing, and 299 µs for event delivery to Invoker. Overall, once transmitted by Trigger, a packet reaches Invoker in no more than 700 µs, in full compliance with what is configured as the QoS request in the testbed setup.

Finally, let us observe that the high priority assigned to the queue processor for SQ events prevents other applications in the user space running at the edge node (such as the MOM control thread) to steal resources from the event processing; of course, this does not happen for BQ. Note also that these measurements

could be considered as the baselines for event delivery by TEMPOS in the ideal case of absence of perturbations.

In the second test of this testbed case, we investigate how the TEMPOS event-delivery mechanisms behave when multiple workflows are active and in competition for resources. This test consists of two rounds: i) 1 best QoS and 1 strict QoS workflows are concurrently active and ii) the number of active best QoS workflows is increased to 3. We decided to increase only the number of best-QoS workflows in this test because the configuration of the current testbed makes impossible the simultaneous sending of more than 1000 strict messages per second, moreover, in most practical scenarios, most events tend to belong to the Best-quality type. We submitted for 5 minutes a constant rate of 1000 events per second per each active flow and again we measured the time-lapse between the creation of the event in Trigger and its delivery at Invoker.

The results in Figure 31 demonstrate that the strict-quality latency is not penalized by the concurrent execution of one or more best-effort workflows even when compared with the baselines of Figure 30. In both rounds, the latency constantly remains under the threshold of 600 µs. As a second-level observation, note that in the first round, despite the concurrent presence of two active flows of event delivery, the jitter is negligible but a noticeable increase is observable in the second round: this is mainly due to our usage of new API (NAPI), a device driver packet processing extension to improve the networking performance; in particular, NAPI implements a mechanism of interrupt mitigation for network devices. This mechanism allows the network card driver to exploit two different packet reception modes: i) interrupt request (IRQ) issued for each incoming packet and ii) a polling [102] based mechanism. Since the IRQ-based implementation can be

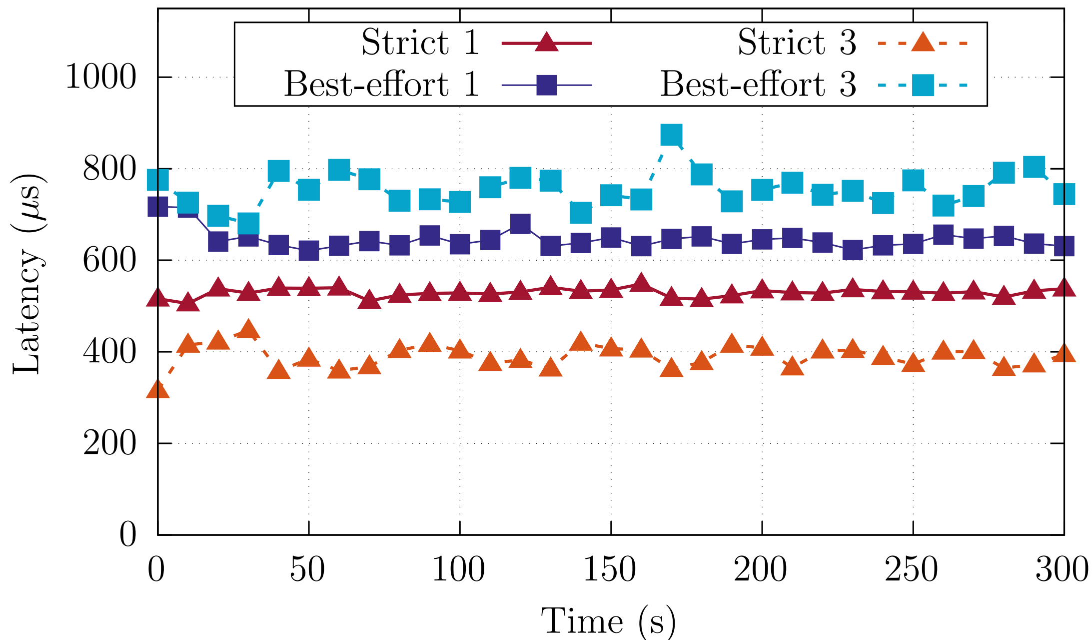


Figure 31: First testbed section showing performance of Delivery slice. Average end-to-end latency of best effort and Strict effort traffic with 1 and 3 concurrent best effort producers and 1 strict effort.

very inefficient in high-speed networks as it constantly interrupts the kernel, NAPI introduces the polling mechanism that allows the kernel to periodically check incoming network packets without being interrupted. When the incoming packet data rate is sufficiently high for NAPI, then it automatically switches to polling-based mode, thus motivating the observed behavior [103].

The periodic activation of the NAPI polling mode allows us to achieve a substantial acceleration in terms of latency, at the expense of jitter and CPU utilization. In the case of workloads more sensitive to jitter than latency, the disabling of this feature is recommendable; TEMPOS can perform this disabling transparently for application developers thanks to its abstractions.

7.4.2 Processing Prioritization

After validating the QoS-constrained delivery features of TEMPOS, here we present a series of tests to show TEMPOS performance in terms of event processing. The following tests are therefore implemented by considering the Processing Layer only, with local-to-nodes function triggering. In this test, we separately experimented again with the three invocation mechanisms (i.e., DLF, WASMF, FSpawn), but this time with concurrent invocations of 6 functions, 2 executed with SQ setup, and 4 with BQ. The level of parallelism selected is motivated by the number of cores available on the used nodes (Node C Table 4) and the need of creating challenging resource conflicts among workloads in our tests.

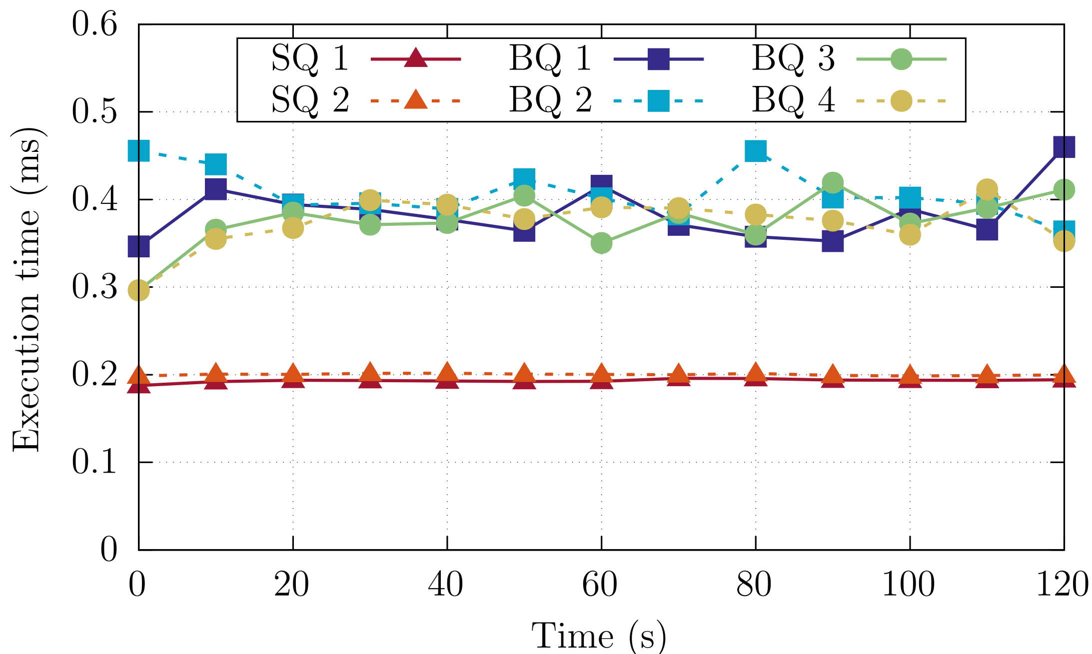


Figure 32: Testbed results of concurrent invocation of functions configured with different QoS using the *Dynamic Loaded Functions* invocation model

As shown in Figures 32, 33, 34 our queuing mechanism can well prioritize strict quality when resource conflicts occur: the time of execution of SQ functions

is almost half of the BQ ones. In other words, Invoker demonstrates to be capable of correctly applying the requested prioritization even with heterogeneous mechanisms and in different execution environments.

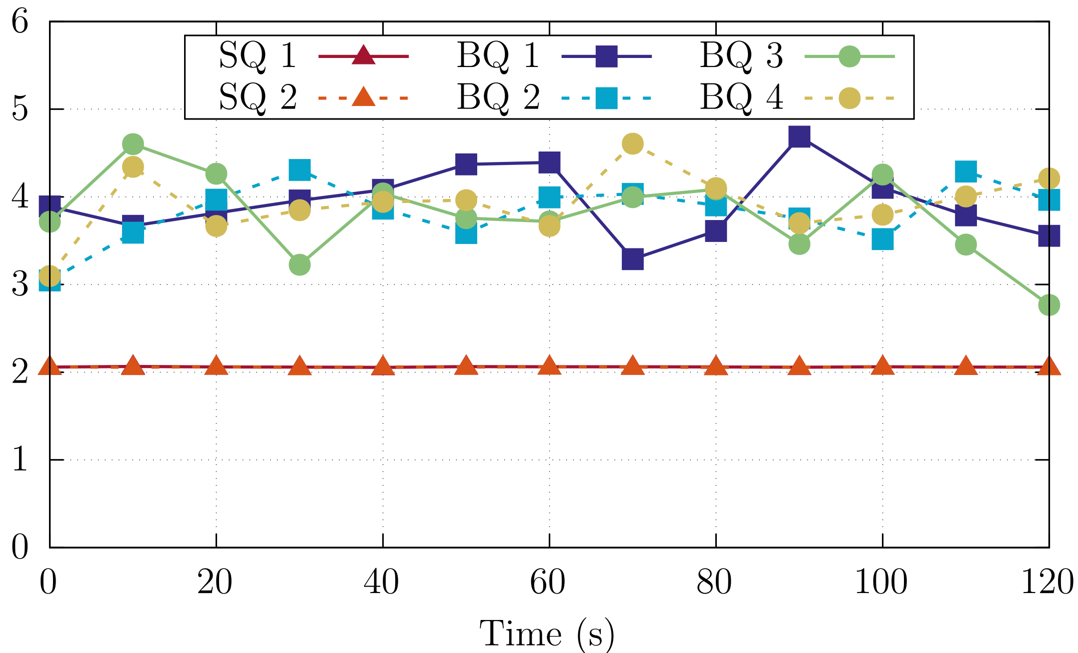


Figure 33: Testbed results of concurrent invocation of functions configured with different QoS using *Function Spawn* invocation model.

In addition, the reported results highlight a negligible variability in execution time in the case of SQ functions, as opposed to BQ. BQ functions showed a significant variation of the execution time of the order of hundreds to thousands of ms depending on the method used; therefore, the usage of SQ functions enables, not only to achieve a further reduced latency but also stricter predictability of processing time. Invoker showed to be capable of transparently executing heterogeneous workloads while exploiting diverse technologies present in infrastructure nodes.

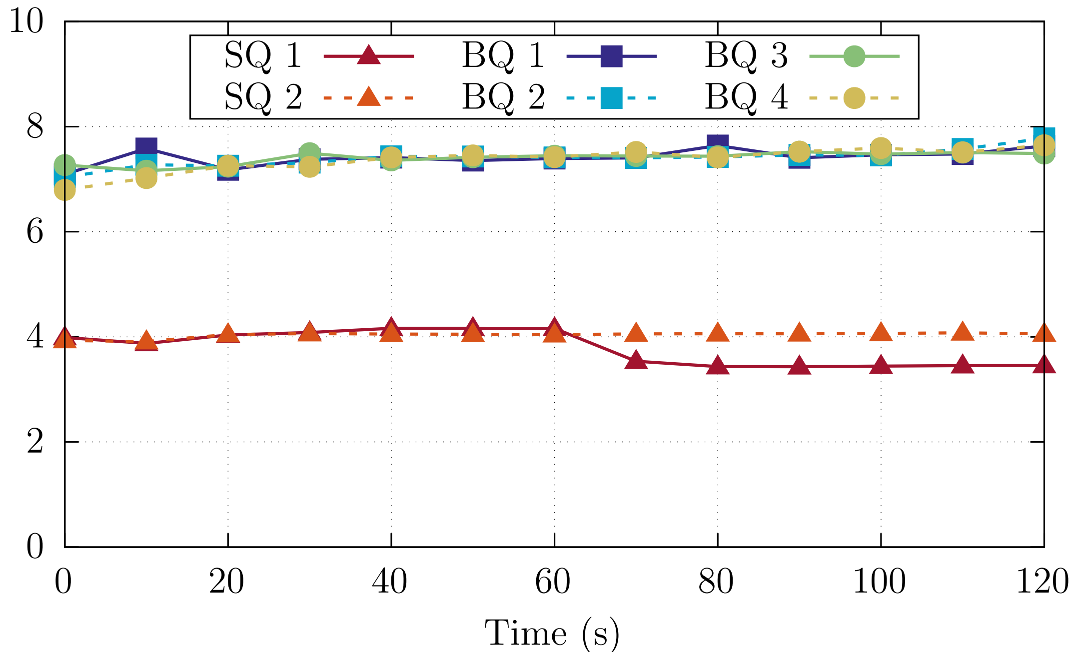


Figure 34: Testbed results of concurrent invocation of functions configured with different QoS, using *WASM Function* invocation model.

7.4.3 Full Stack

This section report results about the TEMPOS ability to coordinate and concatenate different QoS mechanisms available in each slice to achieve the targeted end-to-end quality for the workflows. We deployed on node E two data producers and two triggers configured with the two BQ and SQ levels; on node B, instead, we deployed 3 invokers with SQ configuration and 3 with BQ.

We then create and deployed two workflows executing the same function and triggered by the same event, but configured one with BQ and one with SQ. Next, we linearly increased the number of events submitted to the triggers until reaching 1000 events per second for each workflow. The experiment is repeated firstly with only one active workflow, then with both workflows concurrently active.

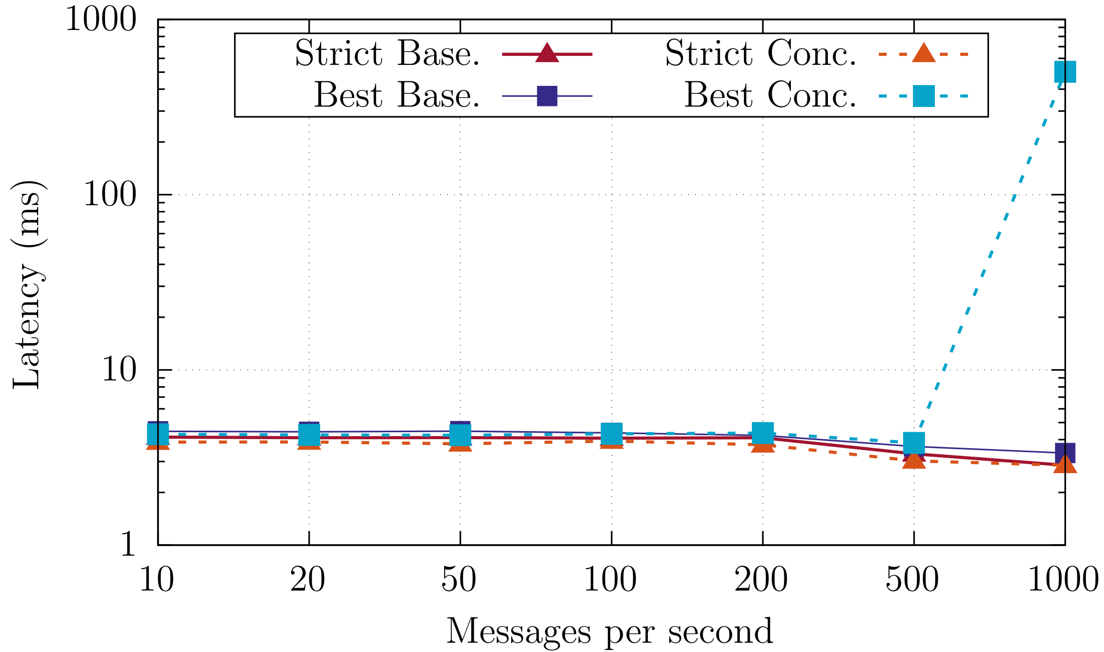


Figure 35: End-to-end test performance of the TEMPOS platform operated in two different scenarios: isolated execution of workflows with different QoS (1 BQ and 1 SQ), and concurrent execution of them. Comparison of end-to-end latency averages for BQ and SQ traffic executed both separately and simultaneously with an increasing number of messages/seconds (from 10 to 1000).

As predictable from the results of the previous sub-section, in an isolation case with only one workflow active per time, the SQ end-to-end latency is considerably better, with an average of $3.34\,\mathrm{ms}$ than BQ, which settled to an average of $3.96\,\mathrm{ms}$, as also shown in Figure 35. It is also noteworthy that this behavior is maintained for the entire duration of the test, with different rates of requests, thus demonstrating the elasticity of the TEMPOS middleware. In the concurrent scenario, with both workflows active and competing for resources, the two workflows coexist and do not affect each other's performance until reaching the critical threshold of 500 messages per second. Until this threshold, we can also observe

that both workflows behave similarly as in the previous experiments where they were executed separately. Over the critical threshold, we can observe that conflicts among workflows become critical and the BQ workflows progressively degrade their performance. Note that the latency performance of SQ workflows remains consistently approximately $3.1\,\mathrm{ms}$ despite the constrained hardware adopted and the concurrency with other workflows.

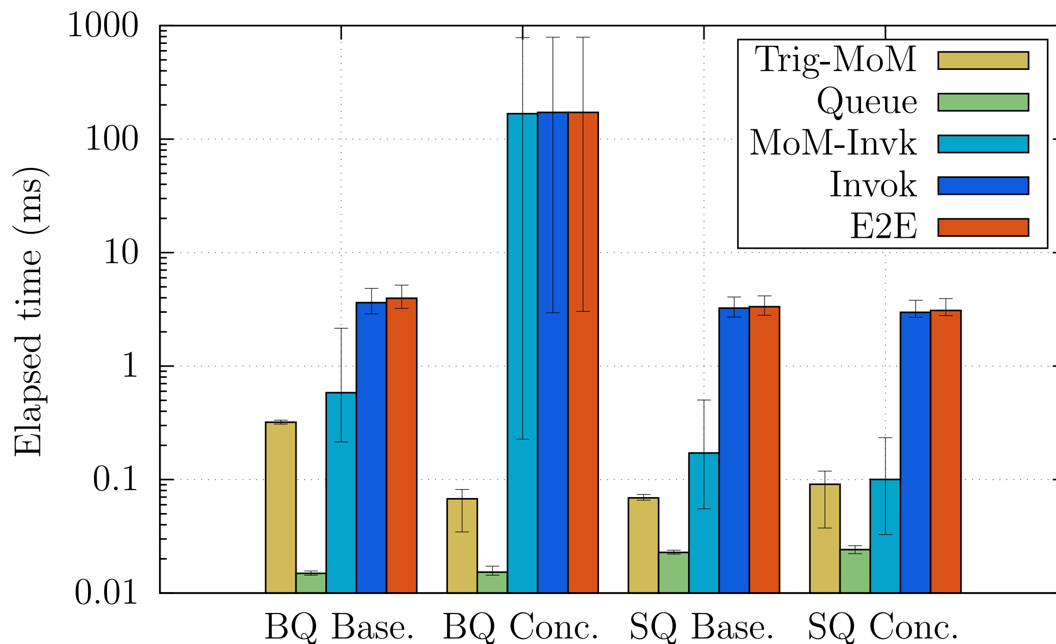


Figure 36: End-to-end test performance of the TEMPOS platform operated in two different scenarios: isolated execution of workflows with different QoS (1 BQ and 1 SQ), and concurrent execution of them. Zoom-in on end-to-end latency results showing single contributions of TEMPOS components (execution time) to the overall response times

Zooming in on the performance behavior of some single TEMPOS components, we can observe (Figure 36) how QoS mechanisms are correctly applied across all the hops of the technological stack. In fact, we can observe how, in each trait of the invocation stack, SQ performs almost identically when executed

in concurrency with other workflows, while BQ workflows degrade their performance when competing with other active workflows. Let us finally note that in Figure 36 the "Best Conc." MOM-Invk bar is almost the same as the Invok bar because the time is taken as the difference between the invoker function invocation instant and the sending message instant from the MOM: given that the invoker reception is sync-blocking, that message anyway waits in the invoker socket until the previous invocation is completed.

7.5 Function Chaining performance comparison

This section presents an experimental evaluation, assessing our proposal as-a-whole while varying the underlying MoM support. In particular, we evaluate and compare the capabilities of DSMQueue with the other traditional MoMs, identifying possible deployment tradeoffs.

To this end, the three emerging configurations are evaluated under three representative workloads: (i) a constant-rate stream of requests, (ii) a stream of incoming requests issued at an increasing rate, and (iii) a large batch of requests submitted to the system in a small amount of time. The first workload aims to assess the properties of our serverless platform in a steady regime of incoming requests, whereas the second and the third scenarios reproduce a typical traffic pattern that arises when a high number of concurrent events need to be processed in batch (e.g., process all the tweets with a specific hashtag).

For this evaluation, we employ a lightweight, short-lived business logic with an execution time of about $60\,\mu s$. Upon termination, the last function of each pipeline appends a timestamp to its output, later on, used to compute the different metrics. Response times are measured as the time-lapse between the moment

the request is issued and the termination timestamp of that function (*end-to-end* latency). We define as *throughput* the number of satisfied requests per unit of time. We examine how these metrics, as well as the total execution time, vary under an increasing composition length. In this assessment, we vary the number of composable functions from 2 to 5, and each function is packaged as a distinct container, although embodying the same business logic. Also, the application graphic is a linear path, hence no branching logic is considered.

Finally, addressing the issue of the cold start phenomena [104], that is minimizing function bootstrap time whose effects are further exacerbated in a composition setting, the invoker adopts a simple pluggable optimization. In the current implementation, the provisioned strategy does not reclaim the resources allocated to the function whenever the request inter-arrival time is lower than the average function bootstrap and execution time. In the experimental setting, this simple, yet effective, logic removes any potential bias in the measurements of the different system configurations

The experiments are conducted on two identical machines, each equipped with a 4-core i5-3470 CPU @ 3.20GHz, 10GB RAM, running Ubuntu 20.04. The two nodes are directly interconnected by a 100Gbps Mellanox ConnectX-6 DX NIC, which supports both standard Ethernet traffic and RDMA networking. On each machine, we run a single instance of the function invoker, which has access to the code of the function to be executed in the experiment. On one of the nodes, we also run a traffic generator process, which we use as a trigger to simulate different ingress traffic patterns: the trigger forwards the invocation requests to the invokers using the MoM.

We configure Kafka with at least-once semantics to avoid the overhead introduced by transactions in an exactly-once mode, and for the same reason, the number of partitions in the topic is set equal to the number of nodes with a replication factor of 1. Redis was deployed as a single instance in one of the two nodes and set up in order to create a Redis Stream with one single active group, i.e., function invokers cooperate to consume a different portion of the same stream of messages. Finally, we configure DSMQueue to replicate the shared-memory queue across a group of three processes, the trigger, and the two invokers. We configure the underlying Derecho library to enforce strong consistency across the replicas, and to keep the shared state in volatile memory, with no persistence support.

7.5.1 Constant-rate stream of incoming requests

In this first experiment, we would like to investigate the system behavior under a steady regime. Hence, the trigger issues a fixed number of requests at a constant rate of 1.000 requests/second, and the experiment is run by varying the length of the function composition from 2 to 5.

Figure 37 shows the end-to-end latency of each execution as a function of the composition length. Note the logarithmic scale in the y-axis, which magnifies the length of the whiskers for the smaller values. We can observe two important trends. First, as expected, the latency increases with the composition length. This increment is generally attributed to the time taken to execute more functions, and the time spent in the (de)queuing operations. At this request rate, the MoMs can sustain the traffic with little to no queueing effects, hence the delay contribution is mainly to be attributed to networking and synchronization of concurrent requests.

In all configurations, the latency increment is linear, but there are important differences. For DSMQeueue, the median latency shows a 2x increment when switching from 2 to 5 functions: much of it is the function execution time, whereas only 33% is caused by additional middleware operations. This increment is more evident in Redis, which demonstrates a higher (3x) latency increment between 2 and 5 functions. As the function execution time is constant, the additional latency time is caused by the middleware operations, which in this case account for 86% of the total increment. Kafka exhibits similar behavior, but with an even higher increment factor (4x).

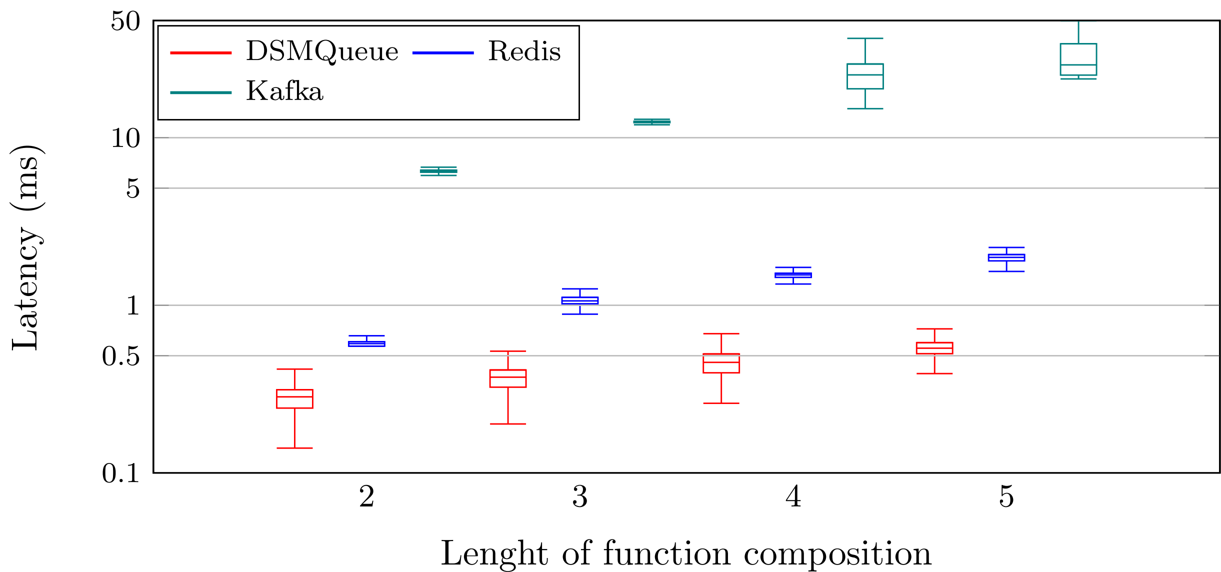


Figure 37: End-to-end latency at a steady regime.

The second consideration regards the relative performance of the different middleware solutions. For the simplest case of two functions in the composition, DSMQueue shows the best median latency (284 µs). While Redis is able to keep up (2x slower), Kafka demonstrates an order of magnitude higher latency (22x slower). The amount of these latency gaps increases as the composition length

increases: for a composition of length 5, Redis is 3.2x worse than DSMQueue (554 µs), and Kafka is again out of scale (49x higher latency).

In conclusion, DMSQueue outperforms the other alternatives, which rely on traditional TCP/IP networking and higher-layer constructs to implement advanced capabilities. Kafka adopts a similar semantic to the other MoMs (see Section 6.8), yet it shows the worse performance by far, and this is to be attributed to its default message/topic persistency support. DSMQueue and Redis have a more similar architecture, but Redis is between two and three times slower in this context.

7.5.2 Incremental rate stream of incoming requests

Herein, we would like to assess system scalability by subjecting the platform to an increased rate of incoming requests, varying from 1 to 65K requests/second. In this scenario, the composition length is kept constant at 3, representing a common option in real-world scenarios e.g., simple map-reduce operations, etc. Figure 38 shows the end-to-end throughput and latency (y-axis in log scale) of the proposal under a varying rate of incoming requests.

Figure 38 shows that the different configurations gracefully scale up the resources to keep up with demand, and throughput increases linearly up to a certain inflection point before starting to decline. This critical point corresponds to the maximum input rate that middleware can sustain without queuing any request: after a threshold, new requests begin to queue up, competing with the existing invocation requests, using up the available resources. The critical rate is similar for DSMQueue and Redis, which start to queue requests between 8K and 12K requests/second, whereas this behavior emerges much earlier in Kafka, at about 240 requests/second.

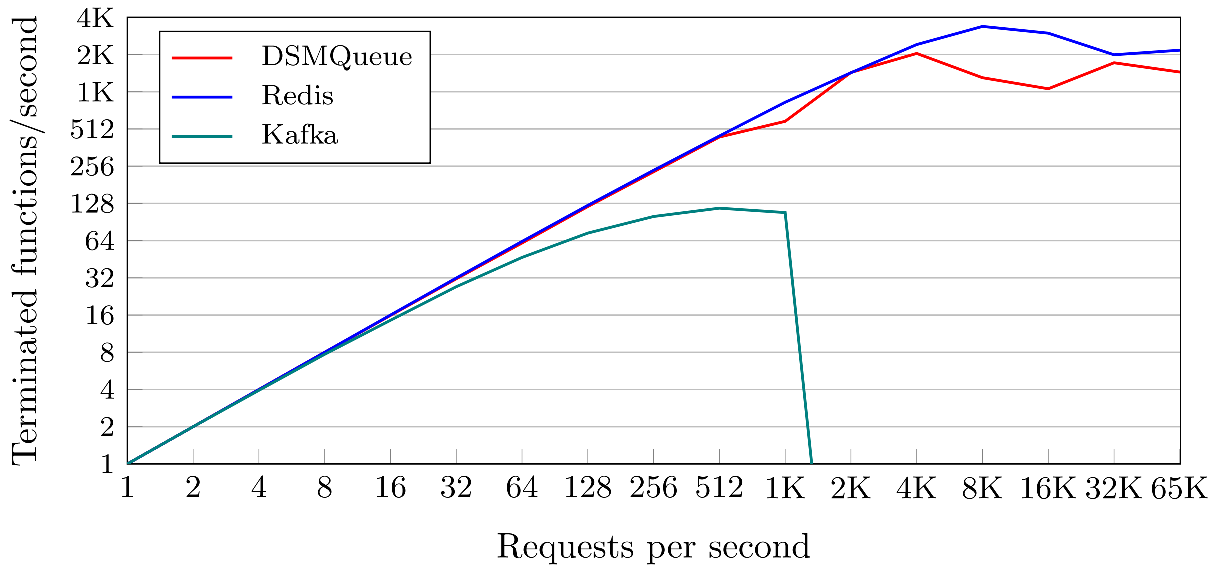


Figure 38: End-to-end latency and throughput with a varying rate of ingress traffic..

As one may expect, the competition for resources between incoming and enqueued requests has a direct effect on latency (Figure 39). Up to the critical input rate, DMSQueue shows a better end-to-end latency than Redis, averaging about $500\,\mu s$ versus $1\,ms$. Shortly after the critical rate, DSMQueue shows an increasingly oscillatory effect, whereas, surprisingly, Redis exhibits a decline in latency. Finally, between 8K (Redis) and 16K (DSMQueue) requests/second, performance degrades rapidly and reaches a similar regime of much higher latency (tens of ms), although DSMQueue still demonstrates a much better behavior. Finally, we observe that Kafka, even in low request regimes, demonstrates an order of magnitude higher latency than both Redis (10x slower) and DSMQueue (20x slower).

Overall, DSMQueue performs well in terms of latency (2x) and has comparable throughput to Redis. This behavior remains constant up to a critical ingress rate, as well as for the highest input rates, while the behavior of both systems becomes unstable during the transition between those two phases. In addition

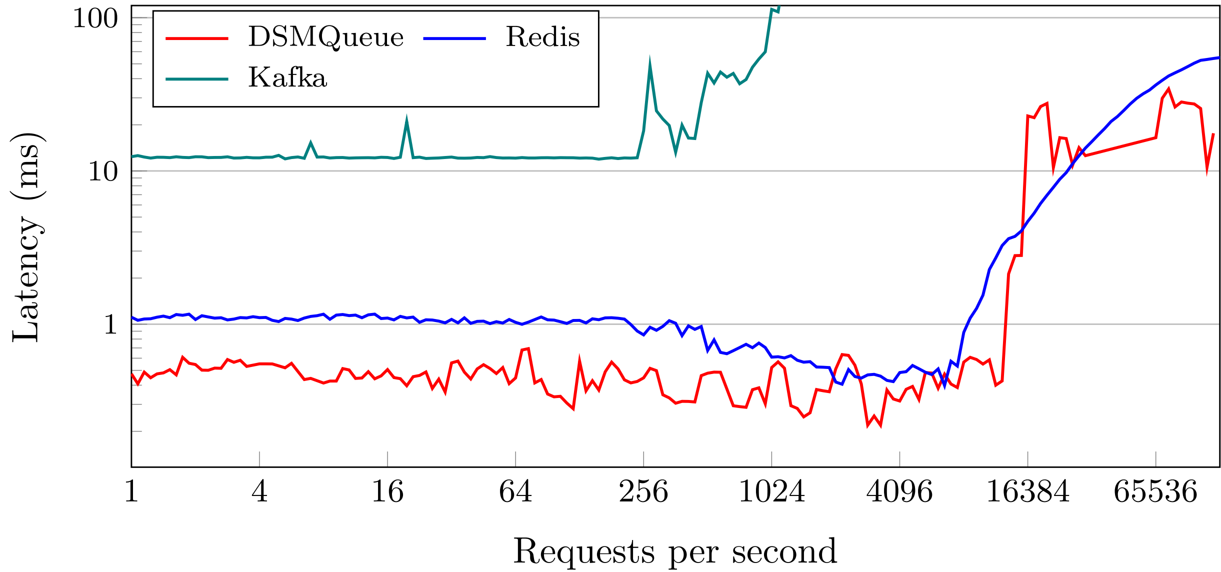


Figure 39: End-to-end latency and throughput with a varying rate of ingress traffic..

to the motivations provided in Sec. 7.5.1, it is noteworthy to point out that the DSMQueue zero-copy approach fully manifests its benefits as the message size grows. This leads to extra spare time, not spent on copying data [100].

On the other hand, the poor performance of Kafka is to be attributed to the MoMs consistency mechanism used to maintain a distributed, structured, and durable commit log of events: any request - ingress data to functional components of the chain - must be acknowledged prior to serving successive ones. Considering the high ingress load and the additional load generated by intermediate results of function executions, the topics acquire an increasing backlog of requests (events) subject to the dynamics of the commit log. As a consequence, the invoker entities tasked with the execution of functions and output serialization to Kafka are subjected to ever-increasing waiting times, expecting an acknowledgment from the broker. This in turn results in a lower end-to-end chain throughput with a respective spike in terms of latency. The specific interval where the phenomenon

manifests itself is tied to the current testbed characteristics (CPU, RAM, etc.): to mitigate it, one could rely on the topic partitioning feature of Kafka, distributing the load among cluster nodes according to design-time criteria. It is noteworthy to point out that in our current setting, Kafka is configured with an "at-least-one" semantic, more optimistic in terms of performance with respect to an "at most-one" semantic.

7.5.3 Burst of incoming requests

In this experiment, we assess the behavior of the platform when subjected to a sudden burst of concurrent requests. To this end, our trigger produces a burst of 10K invocation requests at the highest possible sending rate. This way, we intentionally exacerbate the queuing effect described in Section 7.5.2: the message queue will fill up with invocation requests, as the invokers will not be able to consume them at the same rate. We keep the composition length constant to 3 functions: this further stresses the queue, as per our architecture each pipeline execution requires the invokers to produce and consume new requests to and from the queue.

In this setting, we are interested in the total time the system takes to consume the entire batch of concurrent requests. Figure 40 breaks down the total execution time by plotting the Cumulative Distribution Function (CDF) of the pipeline execution time.

In this case, the different behavior of the considered systems depends on the different waiting times between the execution of two consecutive functions. Such waiting time is determined by the different communication overhead introduced by each solution, which directly affects the speed at which they process the backlog

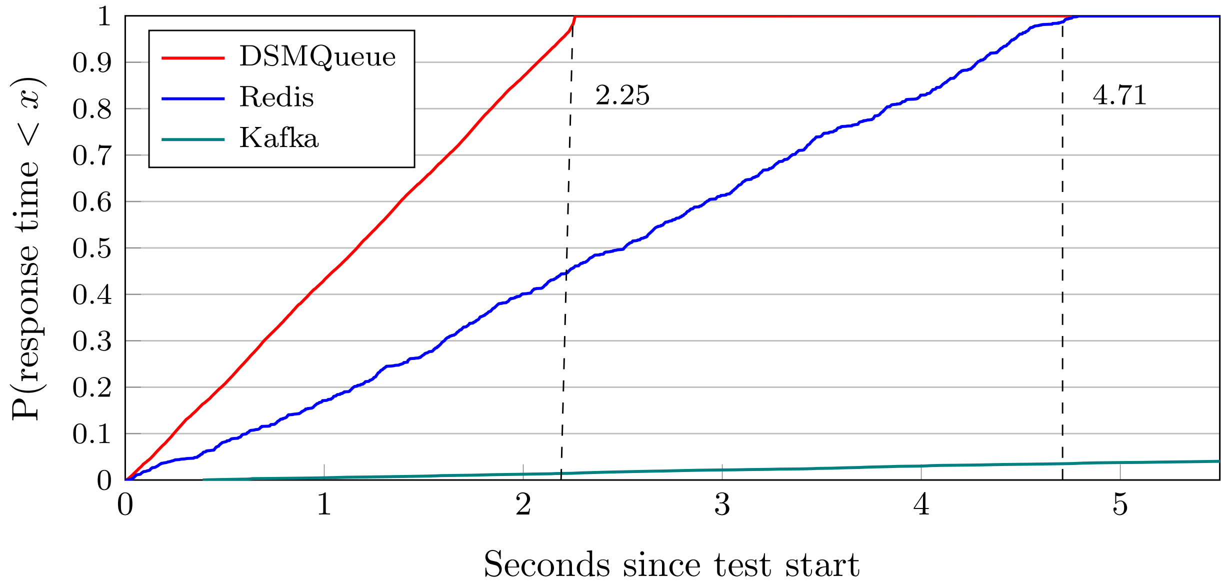


Figure 40: Response time CDF MoM comparison for a composition of length 3.

of requests. In particular, the trend that we observe is the same as we described in the previous experiment. The shared memory approach of DSMQueue is the fastest in processing the request batch (2.25 seconds), Redis takes about twice that time (4.71 seconds), and Kafka is an order of magnitude slower (86.06 seconds) as shown in Figure 41.

7.5.4 MoM enabled load balancing

In this last experiment, we investigate how the different properties of the three MoMs (see Sec. 6.8) impact the distribution of the pipeline workload across the available nodes. Indeed, one driving motivation for this work is to enable DIFFUSE to scale across a varying number of hosts and to efficiently use all the available resources. To effectively measure such efficiency, we are interested in how the workload is distributed when the available machines are subject to different load conditions.

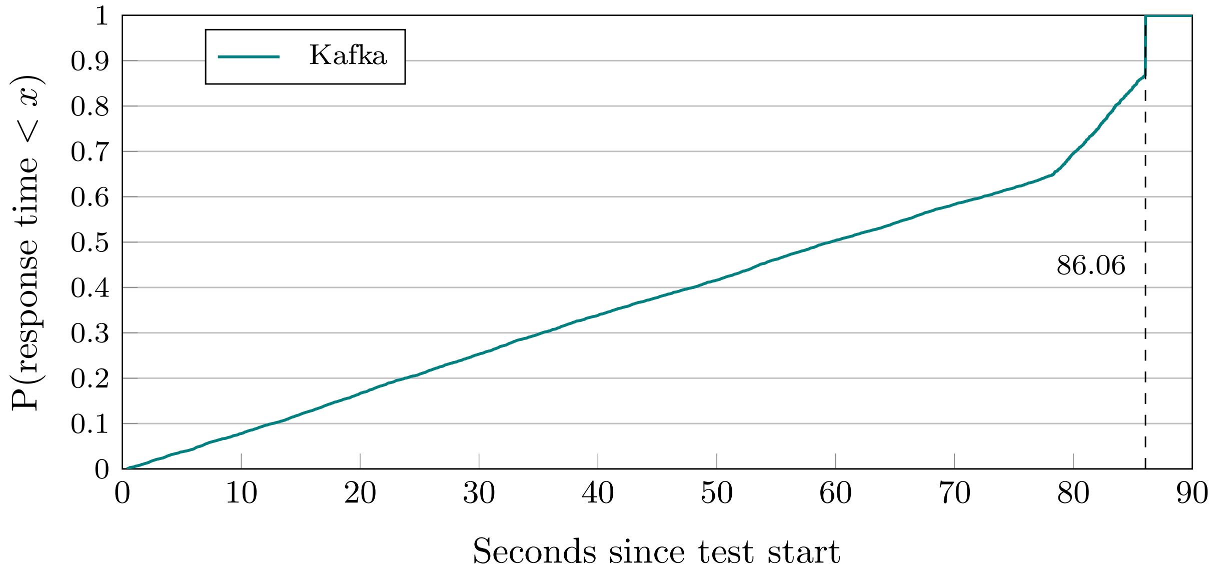
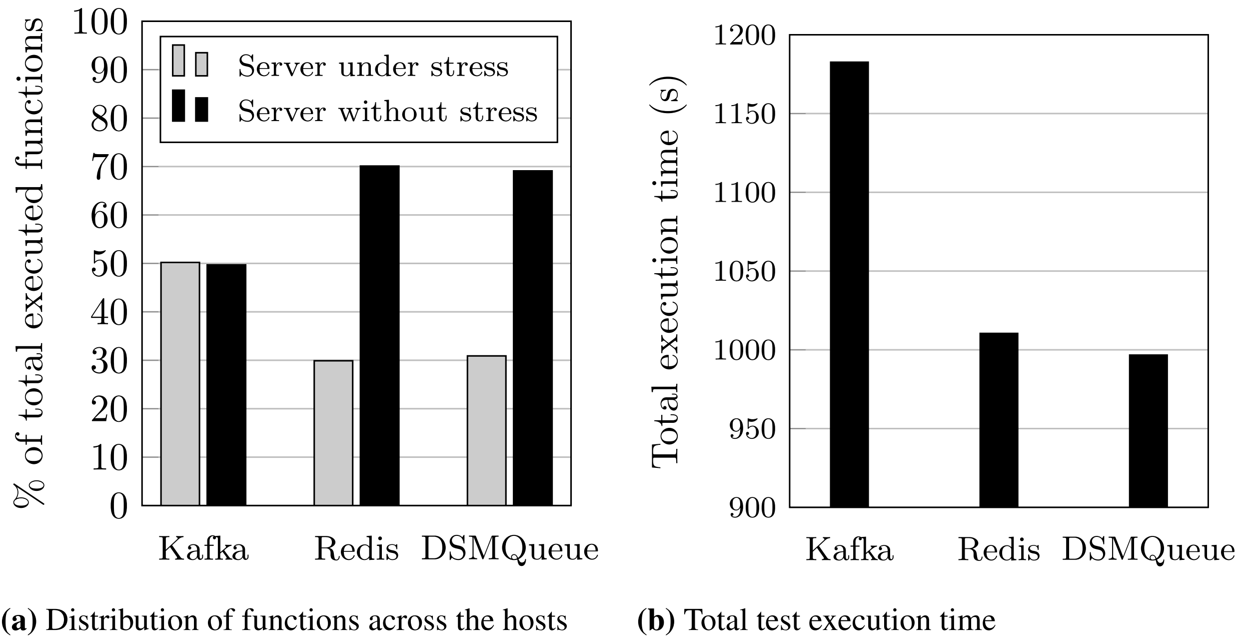


Figure 41: Response time CDF Kafka performance in isolation for a composition of length 3

To this end, we run the same experiment discussed in Sec. 7.5.3, but this time the adopted function is more computationally expensive than the one in the prior experiments, totaling an average execution time of $60\,\mathrm{ms}$. Also, to better highlight the differences between the MoMs, one of the two available hosts executes a background application, saturating its computing, memory, and disk resources. This way, we expect that the same function will take a different execution time depending on the host it is executing on in one case, the function will compete with the background application to acquire the necessary resources, whereas those will be immediately available on the other host. We want to understand if and how each MoM takes the server load condition into account when deciding, transparently to the user, how to distribute the incoming workload.

Figure 42a shows the results. As expected, the same function on the two hosts takes a significantly different amount of time to complete: on average, $35\,\mathrm{ms}$ on the idle one, and more than twice, about $85\,\mathrm{ms}$, on the other. We observe that



(a) Distribution of functions across the hosts **(b)** Total test execution time

Figure 42: Load balancing behavior of the different MoMs

Redis and DMSQueue execute about 30% of the workload on the saturated server, leaving almost 70% of it to the idle machine. On the contrary, Kafka assigns the same number of functions to both hosts. This different behavior is directly linked to the way each MoM implements the queue abstraction (Sec. 6.8). In DSMQueue and Redis Stream, the queue is a (logically) single FIFO buffer that processes compete to access, either when producing or consuming new data. In our setting, these processes correspond to the two invokers. Since the function execution on the idle host takes approximately half of the time taken on the saturated one, the invoker on the idle host ends up consuming more than twice the number of functions than the invoker on the saturated host: an indirect form of load balancing induced by the load on each node. Kafka, instead, blindly follows a round-robin scheme, assigning an equal number of functions to each host. While this approach eliminates the need for coordination among the invokers - which no longer need to compete to access the queue - it also does not take the actual server load into

account. As a consequence, many more functions are scheduled on the saturated host, leaving unused resources on the idle host: this is clearly inefficient, and it results in a significant increase in the time needed by Kafka to process the function batch (Figure 42b).

Even though Redis and DSMQueue already provide an implicit form of load balancing, an explicit mechanism could lead to faster function execution times, and, as a consequence, to improvements in the overall system throughput. The development of a new load-balancing mechanism requires the introduction of an observability layer, providing real-time information on resource usage.

7.6 Experimental assessment summary considerations

In conclusion, this section presented experimental assessments and performance evaluations conducted on our APERTO FaaS proposal. The exploitation of a MOM to interconnect architectural components of APERTO FaaS enabled our solution to leverage heterogeneous computational resources and integrated seamlessly different technologies. Moreover, the asynchronicity in intra-component communications has facilitated and improved the parallelization of workflow executions.

The integration of different methods of *function invocation* and relative isolation technologies enable APERTO FaaS to exploit the more convenient methods depending on the needs of the use case and the characteristics of the hosting infrastructure. From our experiment emerged a strong tradeoff between the flexibility of the solution and the startup overhead of the function. The FSPAWN, based on the classical Linux fork startup procedure, while providing a more flexible execution environment allowing the creation of functions with arbitrary programming

languages and leveraging on different isolation technologies such as VM or Linux container demonstrated the worst performance when compared with the two other methods. On the opposite, the DLF method showed the best performance in each scenario of execution with performances in function execution of different orders of magnitude faster than the other two methods. This performance improvement can be critical in particular when leveraging on limited computational resources as enablers to support greater loads (Table 3). In future work, we plan to indagate in more depth the potentiality of DLF methods, possible further performance improvements, and the porting of this technology to more process isolation technologies. Finally, the WASMF method, based on the WebAssembly emerging technology showed good performance while still providing good support to the execution of functions programmed in many programming languages and executed in a promising isolation environment.

In the next few years, we expect a rising number of technologies and improvements in existing ones specifically addressing the problem of **process startup** while guaranteeing different levels of **isolation**. Function as a Service in fact has shifted part of the research attention not only to execution performance and acceleration of programs but even to their performance startup when executed in different isolation and virtualization environments. Experiments conducted in this regard aim not only to provide an overall perspective on the performance of the different methods but aim also to give a first technical reference to address tradeoffs driven by the choice of the different technologies depending on the application context. The expansion of this technical reference is actually under consideration in order to provide a more general and exhaustive perspective on methods, technologies, performance, and consequent tradeoffs over these technologies.

The *Persistence* and the *Authorization* Layers demonstrated the feasibility of creating a layer addressing cross-cutting concerns and **decoupling** business logic with protocols and configurations of single workflows. The creation of FaaS-specific optimization in these two layers also allows us to demonstrate that integration at the architectural level following distribution and asynchronicity principles is to be preferred with respect to the centralized and synchronous one.

In particular, results from the test conducted over the persistence layer show that the proposal improves end-to-end response latency, leading to better usage of resources and graceful scaling of resources. Experiments conducted over the Authorization layer demonstrated that the invoker is the most appropriate component to integrate access control verification in a serverless platform realizing a complete decentralization of the process. Our evaluations demonstrate, in fact, that the distributed approach outperforms the centralized one under different load conditions and with increasingly complex policies, also delivering better performance when lower computational resources are available.

In future work, we plan to extend the capabilities of SPS and integrate novel optimizations for further accelerating data operations performance On this front, we would like to investigate mechanisms for shared connection pooling and distributed data caching, speeding up the performance of some operators. We also aim to extend SPS to other invoker model architectures and innovative data mesh-oriented solutions, expanding the deployment options of our proposal over a continuum of resources made available by the development of 5G, NVF, and cloud-edge technologies. Concerning the authorization mechanisms, we plan to extend our solution to as many heterogeneous scenarios as possible in the cloud-to-thing continuum, considering different distribution and consistency models such as tan-

gle [105], and blockchain for managing policies across distinct distributed regions. Moreover, we also aim at integrating context-aware access control mechanisms in our authorization architecture.

The **queue abstraction** introduced by the MOM in APERTO FaaS radically changed orchestration and coordination by providing an abstraction for end-user over different prioritization technologies, and a point of indirection and optimization for function-to-function communication.

In particular, we proposed TEMPOS a QoS-aware middleware for serverless platforms which employs and coordinates different QoS mechanisms provided by individual technologies. Leveraging on virtualized FaaS invocation stack in the cloud continuum, TEMPOS is capable of properly managing end-to-end QoS in terms of jitter, latency, and en-queuing time. Therefore, to evaluate the validity of TEMPOS, we presented a series of real testbeds to extensively assess the state-of-the-art implementation of a TEMPOS prototype. The latter mainly exploit Linux real-time scheduling, a novel MOM with differential priorities, and Time-Sensitive Networking (TSN) protocols as underlying system-level mechanisms to apply TEMPOS QoS-aware abstractions (TEMPOS QoS-aware topics) for QoS management.

The results show that TEMPOS strongly differentiates workflows based on the assigned QoS level. Specifically, TEMPOS allows the SQ workflow to maintain an end-to-end latency that is 1 millisecond lower than the BQ workflow, throughout the isolated test. In addition, the SQ flow maintains a stable latency of 3 ms even during concurrent testing, while the BQ averages $600\,\mathrm{ms}$ under heavier workloads. QoS awareness is preserved across the entire invocation stack with the delivery layer able to achieve nearly twice the performance for event deliv-

ery leveraging SQ workflows compared to BQ workflows, even under concurrent execution. Finally, the TEMPOS processing slice leverages multiple invocation methods seamlessly, ensuring that higher priority (SQ) workflows execute twice as fast as lower priority (BQ) workflows.

As future work, we are planning to integrate TEMPOS with a novel resource orchestrator for the full cloud continuum chain, e.g., up to 5G micro-datacenters and traditional geographically distant cloud datacenters, able to fully handle both network [106] and computing [87] resources. In addition, we aim to introduce new levels of QoS considering not only latency and jitter differentiation but also the semantics of delivery and throughput while expanding support to resources not considered in this work such as storage or hardware accelerators.

To support efficient function composition, we presented DIFFUSE: a DIstributed and decentralized platForm enabling Function composition in Serverless Environments. The proposal relies on pluggable middleware solutions serving as conveyors of messages among the platform components. To this aim, different middleware configurations were presented and assessed under different scenarios, highlighting their strengths and weaknesses. Results show that networking techniques like RDMA may bring significant performance advantages and enhanced QoS guarantees. At the same time, systems based on standard networking interfaces represent a valid alternative in environments with more conventional settings or specific constraints on the development, deployment, or scale of the infrastructure.

DIFFUSE is under active development and in future work, we aim to introduce support for hybrid MoM deployments, exploiting also the in-host shared memory optimizations, spanning heterogeneous resources on the edge-to-cloud

continuum. This feature allows for exploiting the most convenient medium for function-to-function communication, depending on the local environmental characteristics. Moreover, we are working on the introduction of a resource management mechanism able to handle the intelligent placement, scaling, replication, and coordination of platform components and functions. At the core of this capability is the introduction of an **observability** layer providing real-time operational data on resource usage, data locations, etc.

8 Conclusion and Future Works

Continuous development and introduction of new technologies in connectivity, devices, and ICT services are opening wider and more interconnected business value chains. The growth of these new business frontiers is continuously stimulated by an ever-increasing number of partners in willingness to cooperate and integrate among themselves and with devices, services, and data available over the internet and in the territory. In many sectors is, in fact, expected an expansion from the usual confinement of single businesses to more integrated and interconnected ones involving partners and information with valuable potential. Industry 5.0, Smart Cities, and Smart Logistics received a lot of attention in the last years from industry and research as valuable examples of fields that are demanding more connected and open integration, but many more are expected to rise in the next future.

In particular, the ever-increasing availability of connected devices and the seeking of customers for more personalized and involving experiences is fastly evolving also the sector of tourism is taking into great relevance also emerging aspects of sustainability and inclusivity. Many regions in Europe and in Italy are currently investing in the progressive digitalization and interconnection of partners in the tourism value chain.

However, **deep fragmentation** in services and technologies adopted by different actors in tourism, as well as in other sectors, that characterize also the whole information provided by customer sensing and IoTs heterogeneity deeply clash with an effective organization of smarter services.

To support a rapid integration of service and data coming from heterogeneous sources we proposed APERTO5.0 a modular solution aiming to address the

problem of heterogeneity by providing a unifying view in which any item (data, service, and agents) can become part of an integration mosaic capable of accommodating any new possible element.

As a practical solution for service and data integration, blending, and augmentation we proposed APERTO FaaS a FaaS platform specifically developed to execute on the cloud continuum overcoming its complexities and taking advantage of its potentiality. In fact, APERTO FaaS MOM-centric architecture promotes **asynchronous** and **decoupled** interactions among architectural components enabling it to seamlessly exploit peculiar characteristics of different computational resources available in the cloud continuum. At the same time, the adoption of the FaaS model abstracts complexity to customers with a consequently reduced development effort of new services and integrations.

Through an extensive test bed, we demonstrated that the layered architecture of APERTO FaaS fosters a separation between business logic and specificities introduced by addressed use cases, such as data formats, technologies, or protocols adopted. This **strong separation** of responsibilities joined with the distributed capacity of the architecture allowed us the integration of performance optimizations targeting cross-cutting concerns in cloud-oriented systems and more specifically in FaaS platforms. In particular thanks to the integration of novel mechanisms for data operation and service authorization APERTO FaaS achieved significant latency reduction, up to 55% in data operations and 80% in authorization verifications, leading also to better usage of infrastructural resources. Testing conducted on these two levels also revealed that the right strategies for decentralization of operations involved in processing user requests can support the system in achieving better concurrency and parallelization of requests.

A key feature of APERTO FaaS is the flexibility to exploit and integrate different resources and devices available in the Cloud Continuum. As part of the APERTO FaaS proposal, TEMPOS realized this flexibility by **integrating heterogeneous prioritization technologies** and creating in this way an effective mechanism for the QoS differentiation of workflows execution. Experimental results showed that TEMPOS is able to differentiate workflows execution, even in conditions of contentions of computational resources, guaranteeing a lower latency and jitter for those configured with a higher prioritization.

Asynchronous communication and strong decoupling among architecture components enabling APERTO FaaS to exploit the capabilities of different technologies lead to the proposal of DIFFUSE a framework for function composition. DIFFUSE queue abstraction for function-to-function communication and the **distribution of composition logic among** infrastructural nodes enabled the creation of optimized workflows composing the execution of multiple functions. DIFFUSE, also, demonstrated that local optimizations, such as the exploitation of shared memory technologies, can accelerate complex function workflow execution and increase overall platform throughput. Those optimizations are offered in a completely transparent way to the developer which can create distributed workflows spanning multiple sites.

We believe that the acceleration in the pervasive adoption of ICT technologies will push different domains to wider and more open cooperation among actors, enriching value chains and supporting the achievement of shared and communitary objectives such as **sustainability**, **inclusivity**, and **resiliency**. We sustain that innovation and concepts introduced by APERTO and APERTO FaaS can effectively be employed to support the integration and interconnection of services

and data in many of these emerging use cases. Through the cooperation with many realities, we then plan to research the challenges and benefits of adopting APERTO proposal in different domains, such as Smart Transportation, Smart Cities, and Industry 5.0.

Considering the contribution of APERTO5.0 to the tourism management field, owing to the cooperation with realities of the territory we plan a deeper exploration of the potentiality of a pervasive application of concepts and possibilities opened by APERTO in the many possible tourism facets and in particular in tourism management deserves better exploration by field experts.

The distribution of services over resources of the *Cloud Continuum* is expected to assume ever-increasing importance to cope with the rising demand for contextualized and ever-updated information coming also from the ever-increasing number of devices and services connected. In this context, the development and consolidation of new cloud computing service models **abstracting the complexity** of distributed and heterogeneous environments is a key factor to open a wider adoption of these distribution models. In particular, the FaaS service model is expected to achieve greater importance in the market thanks also to performance improvements introduced by research opening to the application of this paradigm to more use case scenarios characterized by constraints such as low-latency or low power consumption. During our research path, many new challenges emerged (Section 7.6) in this direction and we plan to extend our APERTO proposal in order to expand its capabilities and adaptability to address an ever-increasing number of scenarios.

The study and development of new layers abstracting and optimizing complexities of distributed and heterogeneous environments, while presenting a sim-

plified view to developers, is a promising research direction. In particular, the ever-increasing attention to the theme of **sustainability** of IT infrastructure demands the creation of orchestration mechanisms able to make developers conscious of environmental impacts while still abstracting from implementation complexities. In this direction, we plan to expand APERTO offer by integrating novel low-power technologies, abstractions, and optimizations to differentiate APERTO services in terms of performance and power consumption. Experimentation and integration of novel Cloud Computing models, such as event and service mesh, could further complete the APERTO offer by enabling and conveniently proposing a more sustainable model suiting customer needs. Finally, we consider an appealing direction of research the study and creation of novel orchestration mechanisms able to create new forms of **federated** and more **resilient** collaboration among different sparse resources and partners in the cloud continuum. The creation of federations could lead to further decentralization of APERTO ecosystem encouraging cooperation among partners in terms of both resources and services.

Acronyms

AAA Auditing, Authentication, and Authorization. 51, 54

AC Access Control. 71

APERTO Architecture for Personalization and Elaboration of services and data to Reshape Tourism Offers 5.0. 2, 3, 47, 50, 55, 164

API Application Programming Interface. 14, 48, 82

AWS Amazon Web Services. 14

BaaS Backend as a Service. 22

CaaS Container as a Service. 69

CC *Cloud Continuum*. 1, 3, 6, 19

CCCP Cross Cutting Concern Plane. 49–51

CRM Customer Relationship Management. 23

DNS Domain Name System. 23

ERP Enterprise Resource Planning. 23

EU European Union. 41

FaaS Function as a Service. 1, 3–7, 24, 25, 39, 67, 164

GCP Google Cloud Platform. 14

GDPR General Data Protection Regulation. 17

gRPC Google Remote Procedure Call. 30

HTTP Hypertext Transfer Protocol. 26, 30

IaaS Infrastructure as a Service. 21

ICT Information and Communication Technologies. 4, 8, 16, 44, 78, 161

IoE Internet of Everything. 15

IoT Internet of Things. 1, 15, 79, 161

IT Information Technology. 8, 23

JPA Java Persistence API. 70

JSON JavaScript Object Notation. 30

MaaS Metal as a Service. 20

MOM Message Oriented Middleware. 30, 71, 84, 162

PaaS Platform as a Service. 22, 39, 69

QoS Quality of Service. 3, 4, 6, 78, 79, 85

SaaS Software as a Service. 22

SPS Serverless Persistence Support. 69

ST Smart Tourism. 41

TCP Transmission Control Protocol. 26

TEMPOS Time-Effective Middleware for Priority Oriented Serverless. 77

TOML Tom's Obvious Minimal Language. 30

TSN Time Sensitive Networking. 85

XaaS Everything as a Service. 23

XML Extensible Markup Language. 30

References

[1] L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, and O. Rana, “The Internet of Things, Fog and Cloud continuum: Integration and challenges,” *Internet of Things*, vol. 3-4, pp. 134–155, 10 2018.

[2] “The NIST Definition of Cloud Computing,” in *Application Performance Management (APM) in the Digital Enterprise*, 2017.

[3] L. Acquaviva, P. Bellavista, F. Bosi, A. Corradi, L. Foschini, S. Monti, and A. Sabbioni, “NoMISHAP: A Novel Middleware Support for High Availability in Multicloud PaaS,” *IEEE Cloud Computing*, vol. 4, no. 4, 2017.

[4] D. Petcu, “Multi-Cloud: Expectations and Current Approaches,” in *Proceedings of the 2013 International Workshop on Multi-cloud Applications and Federated Clouds*, MultiCloud ’13, (New York, NY, USA), pp. 1–6, ACM, 2013.

[5] L. U. Khan, I. Yaqoob, N. H. Tran, S. M. Kazmi, T. N. Dang, and C. S. Hong, “Edge-Computing-Enabled Smart Cities: A Comprehensive Survey,” *IEEE Internet of Things Journal*, vol. 7, pp. 10200–10232, 10 2020.

[6] Y. Lin and H. Shen, “CloudFog: Leveraging Fog to Extend Cloud Gaming for Thin-Client MMOG with High Quality of Service,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, pp. 431–445, 2 2017.

[7] Z. Chen, W. Hu, J. Wang, S. Zhao, B. Amos, G. Wu, K. Ha, K. Elgazzar, P. Pillai, R. Klatzky, D. Siewiorek, and M. Satyanarayanan, “An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance,” *2017 2nd ACM/IEEE Symposium on Edge Computing, SEC 2017*, 10 2017.

[8] N. N. Dao, Y. Lee, S. Cho, E. Kim, K. S. Chung, and C. Keum, “Multi-tier multi-access edge computing: The role for the fourth industrial revolution,” *International Conference on Information and Communication Technology Convergence: ICT Convergence Technologies Leading the Fourth Industrial Revolution, ICTC 2017*, vol. 2017-December, pp. 1280–1282, 12 2017.

[9] M. Iorga, L. Feldman, R. Barton, M. J. Martin, N. Goren, and C. Mahmoudi, "Fog Computing Conceptual Model," *NIST Special Publication*, vol. 500-325, 2018.

[10] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, pp. 637–646, 10 2016.

[11] "General Data Protection Regulation (GDPR) – Official Legal Text."

[12] I. Sittón-Candanedo, R. S. Alonso, J. M. Corchado, S. Rodríguez-González, and R. Casado-Vara, "A review of edge computing reference architectures and a new global edge proposal," *Future Generation Computer Systems*, vol. 99, pp. 278–294, 10 2019.

[13] J. Wang, J. Pan, and F. Esposito, "Elastic Urban Video Surveillance System Using Edge Computing," *Proceedings of the Workshop on Smart Internet of Things - SmartIoT '17*, vol. 6, 2017.

[14] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A survey on the edge computing for the internet of things," *IEEE Access*, vol. 6, pp. 6900–6919, 2018.

[15] K. Zhang, S. Leng, Y. He, S. Maharjan, and Y. Zhang, "Mobile Edge Computing and Networking for Green and Low-Latency Internet of Things," *IEEE Communications Magazine*, vol. 56, pp. 39–45, 5 2018.

[16] F. J. Ferrández-Pastor, H. Mora, A. Jimeno-Morenilla, and B. Volckaert, "Deployment of IoT Edge and Fog Computing Technologies to Develop Smart Building Services," *Sustainability 2018, Vol. 10, Page 3832*, vol. 10, p. 3832, 10 2018.

[17] O. O. Ajibola, T. E. El-Gorashi, and J. M. Elmirghani, "Disaggregation for improved efficiency in fog computing era," *International Conference on Transparent Optical Networks*, vol. 2019-July, 7 2019.

[18] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, vol. 62, pp. 44–54, 12 2019.

[19] "Watchdog - OpenFaaS."

[20] "Apache OpenWhisk," 7 2020.

[21] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The serverless trilemma: Function composition for serverless computing," *Onward! 2017 - Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, co-located with SPLASH 2017*, pp. 89–103, 10 2017.

[22] S. Risco, G. Moltó, D. M. Naranjo, and I. Blanquer, "Serverless Workflows for Containerised Applications in the Cloud Continuum," *Journal of Grid Computing*, vol. 19, pp. 1–18, 9 2021.

[23] K. R. Sheshadri and J. Lakshmi, "QoS aware FaaS for Heterogeneous Edge-Cloud continuum," *IEEE International Conference on Cloud Computing, CLOUD*, vol. 2022-July, pp. 70–80, 2022.

[24] M. Cimbaljević, U. Stankov, and V. Pavluković, "Current Issues in Tourism Going beyond the traditional destination competitiveness-reflections on a smart destination in the current research," 2018.

[25] Eurostat, "Tourism statistics," 2018.

[26] "EUROPEAN CAPITAL OF SMART TOURISM."

[27] G. Ulrike, M. Sigala, X. Zheng, and K. Chulmo, "Smart tourism: foundations and developments," *Electronic Markets*, vol. 25, pp. 179–188, 9 2015.

[28] I. Wen, "Factors Affecting the Online Travel Buying Decision: A Review," *International Journal of Contemporary Hospitality Management*, vol. 21, pp. 752–765, 5 2009.

[29] B. Radojević, L. Lazić, and M. Cimbaljević, "Rescaling smart destinations: The growing importance of smart geospatial services during and after COVID-19 pandemic," *Geographica Pannonica*, vol. 24, no. 3, pp. 221–228, 2020.

[30] G. Cardone, L. Foschini, P. Bellavista, A. Corradi, C. Borcea, M. Talasila, and R. Curtmola, "Fostering participaction in smart cities: A geo-social crowdsensing platform," *IEEE Communications Magazine*, vol. 51, no. 6, pp. 112–119, 2013.

[31] G. Cardone, A. Cirri, A. Corradi, L. Foschini, R. Ianniello, and R. Montanari, "Crowdsensing in Urban areas for city-scale mass gathering management: Geofencing and activity recognition," *IEEE Sensors Journal*, vol. 14, pp. 4185–4195, 12 2014.

[32] S. Deterding, D. Dixon, R. Khaled, and L. Nacke, "From game design elements to gamefulness: Defining "gamification"," *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments, MindTrek 2011*, pp. 9–15, 2011.

[33] F. Xu, J. Weber, and D. Buhalis, "Gamification in Tourism," *Information and Communication Technologies in Tourism 2014*, pp. 525–537, 2013.

[34] B. Devkota, H. Miyazaki, A. Witayangkurn, and S. M. Kim, "Using Volunteered Geographic Information and Nighttime Light Remote Sensing Data to Identify Tourism Areas of Interest," *Sustainability 2019, Vol. 11, Page 4718*, vol. 11, p. 4718, 8 2019.

[35] J. Y. Lee and M. H. Tsou, "Mapping Spatiotemporal Tourist Behaviors and Hotspots Through Location-Based Photo-Sharing Service (Flickr) Data," *Lecture Notes in Geoinformation and Cartography*, vol. 0, no. 208669, pp. 315–334, 2018.

[36] H. Q. Vu, G. Li, R. Law, and B. H. Ye, "Exploring the travel behaviors of inbound tourists to Hong Kong using geotagged photos," *Tourism Management*, vol. 46, pp. 222–232, 2 2015.

[37] T. N. Maeda, M. Yoshida, F. Toriumi, and H. Ohashi, "Extraction of Tourist Destinations and Comparative Analysis of Preferences Between Foreign Tourists and Domestic Tourists on the Basis of Geotagged Social Media Data," *ISPRS International Journal of Geo-Information 2018, Vol. 7, Page 99*, vol. 7, p. 99, 3 2018.

[38] C. Prandi, P. Salomoni, and S. Mirri, "mPASS: Integrating people sensing and crowdsourcing to map urban accessibility," in *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*, pp. 591–595, 2014.

[39] "BOLOGNA – Prime esperienze di mappatura dell'accessibilità cittadina. Dopo Firenze kimappers in Emilia-Romagna."

[40] N. Chung, H. Lee, J. Ham, and C. Koo, “Smart Tourism Cities’ Competitiveness Index: A Conceptual Model,” *Information and Communication Technologies in Tourism 2021*, pp. 433–438, 2021.

[41] A. Varfolomeyev, D. Korzun, A. Ivanovs, H. Soms, and O. Petrina, “Smart Space based Recommendation Service for Historical Tourism,” *Procedia Computer Science*, vol. 77, pp. 85 – 91, 2015.

[42] A. Sabbioni, T. Villano, and A. Corradi, “An Architecture for Service Integration to Fully Support Novel Personalized Smart Tourism Offerings,” *Sensors 2022, Vol. 22, Page 1619*, vol. 22, p. 1619, 2 2022.

[43] “Programma Affiliati Booking.com per hotel - Guadagna dal tuo sito.”

[44] “Hibernate. Everything data..”

[45] “EclipseLink.”

[46] “SQLAlchemy - The Database Toolkit for Python.”

[47] A. Trivedi, L. Wang, H. Bal, and A. Iosup, “Sharing and Caring of Data at the Edge,” in *Proc. of USENIX HotEdge Workshop*, USENIX Association, June 2020.

[48] V. Sreekanti *et al.*, “Cloudburst: Stateful Functions-as-a-Service,” *Proc. VLDB Endow.*, vol. 13, p. 2438–2452, jul 2020.

[49] C. Wu, J. M. Faleiro, Y. Lin, and J. M. Hellerstein, “Anna: A KVS for Any Scale,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, pp. 344–358, 2 2021.

[50] “Durable Functions Overview - Azure — Microsoft Docs.”

[51] “Understanding database options for your serverless web applications — AWS Compute Blog.”

[52] T. Zhang, D. Xie, F. Li, and R. Stutsman, “Narrowing the Gap Between Serverless and its State with Storage Functions,” *Proceedings of the ACM Symposium on Cloud Computing*, p. 12, 2019.

[53] A. Mahgoub, S. Chaterji, S. Bagchi, K. Shankar, S. Mitra, and A. Klimovic, “SONIC: Application-aware data passing for chained serverless applications,” in *2021 USENIX Annual Technical Conference*, 2021.

[54] A. Sabbioni, A. Bujari, S. Romeo, L. Foschini, and A. Corradi, "An Architectural Approach for Heterogeneous Data Access in Serverless Platforms," *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, pp. 129–134, 12 2022.

[55] P. Samarati and S. C. de Vimercati, "Access control: Policies, models, and mechanisms," in *International School on Foundations of Security Analysis and Design*, pp. 137–196, Springer, 2000.

[56] "AWS - Use API Gateway Lambda authorizers."

[57] K. Alpernas et al., "Secure Serverless Computing Using Dynamic Information Flow Control," *Proc. ACM Program. Lang.*, vol. 2, oct 2018.

[58] P. Datta et al., "Valve: Securing Function Workflows on Serverless Computing Platforms," in *Proceedings of The Web Conference 2020*, WWW '20, (New York, NY, USA), p. 939–950, Association for Computing Machinery, 2020.

[59] A. Sankaran, P. Datta, and A. Bates, "Workflow Integration Alleviates Identity and Access Management in Serverless Computing," in *Annual Computer Security Applications Conference*, ACSAC '20, (New York, NY, USA), p. 496–509, Association for Computing Machinery, 2020.

[60] A. Sabbioni, C. Mazzocca, A. Bujari, R. Montanari, and A. Corradi, "A Decentralized Architecture for Dynamic and Federated Access Control Facilitating Smart Tourism Services," *ACM International Conference Proceeding Series*, pp. 403–408, 9 2022.

[61] E. Brewer, "A certain freedom," pp. 335–335, 2010.

[62] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang, "A Survey on Mobile Edge Networks: Convergence of Computing, Caching and Communications," *IEEE Access*, vol. 5, pp. 6757–6779, 2017.

[63] M. Agiwal, A. Roy, and N. Saxena, "Next generation 5G wireless networks: A comprehensive survey," *IEEE Communications Surveys and Tutorials*, vol. 18, pp. 1617–1655, 7 2016.

[64] D. Buhalis and R. Leung, "Smart hospitality—Interconnectivity and interoperability towards an ecosystem," *International Journal of Hospitality Management*, vol. 71, pp. 41–50, 4 2018.

[65] "Indoor 5G Scenario Orientated White Paper - Huawei 2019 - GSA."

[66] C. Avasalcai, B. Zarrin, and S. Dustdar, "EdgeFlow -Developing and Deploying Latency-Sensitive IoT Edge applications," *IEEE Internet of Things Journal*, 2021.

[67] J. Pan and J. McElhannon, "Future Edge Cloud and Edge Computing for Internet of Things Applications," *IEEE Internet of Things Journal*, vol. 5, pp. 439–449, 2 2018.

[68] Y. C. Hu, M. Patel, D. Sabella, and V. Young, "ETSI White Paper #11 Mobile Edge Computing - a key technology towards 5G," tech. rep., 2015.

[69] G. Castellano, F. Esposito, and F. Risso, "A Distributed Orchestration Algorithm for Edge Computing Resources with Guarantees," *Proceedings - IEEE INFOCOM*, vol. 2019-April, pp. 2548–2556, 4 2019.

[70] R. E. Schantz, J. P. Loyall, C. Rodrigues, D. C. Schmidt, Y. Krishnamurthy, and I. Pyarali, "Flexible and Adaptive QoS Control for Distributed Real-Time and Embedded Middleware," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2672, pp. 374–393, 2003.

[71] M. Sojka, P. Píša, D. Faggioli, T. Cucinotta, F. Checconi, Z. Hanzálek, and G. Lipari, "Modular software architecture for flexible reservation mechanisms on heterogeneous resources," *Journal of Systems Architecture*, vol. 57, pp. 366–382, 4 2011.

[72] L. Baresi, D. Filgueira Mendonça, and M. Garriga, "Empowering Low-Latency Applications Through a Serverless Edge Computing Architecture," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10465 LNCS, pp. 196–210, 2017.

[73] S. Shillaker and P. Pietzuch, *Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing*. {USENIX} Association, 2020.

[74] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, H. Chen, and C.-g. Qin, "Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting,"

[75] A. Garbugli, A. Sabbioni, A. Corradi, and P. Bellavista, "TEMPOS: QoS Management Middleware for Edge Cloud Computing FaaS in the Internet of Things," *IEEE Access*, vol. 10, pp. 49114–49127, 2022.

[76] P. Patros *et al.*, "Toward Sustainable Serverless Computing," *IEEE Internet Computing*, vol. 25, no. 6, pp. 42–50, 2021.

[77] S. S. Shinde, D. Marabissi, and D. Tarchi, "A Network Operator-biased approach for Multi-service Network Function Placement in a 5G Network Slicing Architecture," *Computer Networks*, vol. 201, p. 108598, 2021.

[78] F. A. Salaht, F. Desprez, and A. Lebre, "An Overview of Service Placement Problem in Fog and Edge Computing," *ACM Comput. Surv.*, vol. 53, jun 2020.

[79] Microsoft, "Azure Functions Serverless Compute."

[80] Amazon, "AWS Step Functions."

[81] I.E. Akkus *et al.*, "SAND: Towards high-performance serverless computing," in *Proc. of USENIX Annual Technical Conference (USENIX ATC)*, (Boston, MA), pp. 923–935, July 2018.

[82] K. Inc., "Krustlet."

[83] A. Sabbioni, L. Rosa, A. Bujari, L. Foschini, and A. Corradi, "DIFFUSE: A DIstributed and decentralized platForm enabling Function composition in Serverless Environments," *Computer Networks*, vol. 210, p. 108993, 6 2022.

[84] "MessagePack: It's like JSON. but fast and small.."

[85] "Apache Kafka."

[86] "Welcome - NATS Docs."

[87] A. Bujari, C. Bergamini, A. Corradi, L. Foschini, C. E. Palazzi, and A. Sabbioni, "A Geo-distributed Architectural Approach Favouring Smart Tourism Development in the 5G Era," in *ACM International Conference Proceeding Series*, 2020.

[88] H. B. Hassan, S. A. Barakat, and Q. I. Sarhan, "Survey on serverless computing," *Journal of Cloud Computing*, vol. 10, 12 2021.

[89] M. Kerrisk, *The Linux programming interface: a Linux and UNIX system programming handbook.* No Starch Press, 2010.

[90] C. Jung, D. K. Woo, K. Kim, and S. S. Lim, "Performance characterization of prelinking and preloadingfor embedded systems," *EMSOFT'07: Proceedings of the Seventh ACM and IEEE International Conference on Embedded Software*, pp. 213–220, 2007.

[91] "wasmer_engine_dylib - rust."

[92] "wasmtime/cranelift at main.bytecodealliance/wasmtime."

[93] A. Baumann, J. Appavoo, O. Krieger, and T. Roscoe, "A fork() in the road," *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019*, pp. 14–22, 2019.

[94] J. C. Hamerski, A. R. Domingues, F. G. Moraes, and A. Amory, "Evaluating Serialization for a Publish-Subscribe Based Middleware for MPSoCs," in *Proc. of IEEE ICECS*, pp. 773–776, 2018.

[95] C. Scordino and G. Lipari, "Linux and Real-Time: Current Approaches and Future Opportunities," *Anipla 2006*, 2006.

[96] W. W. Ho and R. A. Olsson, "An approach to genuine dynamic linking," *Software: Practice and Experience*, vol. 21, no. 4, pp. 375–390, 1991.

[97] L. Lo Bello and W. Steiner, "A Perspective on IEEE Time-Sensitive Networking for Industrial Communication and Automation Systems," *Proceedings of the IEEE*, vol. 107, no. 6, pp. 1094–1120, 2019.

[98] A. Nasrallah, A. S. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, and H. ElBakoury, *Ultra-low latency (ULL) networks: The IEEE TSN and IETF DetNet standards and related 5G Ull research*, vol. 21. 2019.

[99] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, p. 299–319, Dec. 1990.

[100] S. Jha *et al.*, "Derecho: Fast state machine replication for cloud services," *ACM Trans. Comput. Syst.*, vol. 36, Apr. 2019.

[101] "An introduction to redis streams."

[102] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond softnet.," 2001.

[103] "networking:napi [Wiki]."

[104] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," 2 2019.

[105] C. Mazzocca, A. Sabbioni, R. Montanari, and M. Colajanni, "Evaluating Tangle Distributed Ledger for Access Control Policy Distribution in Multi-region Cloud Environments," *Communications in Computer and Information Science*, vol. 1621 CCIS, pp. 296–306, 2022.

[106] A. Garbugli, A. Bujari, and P. Bellavista, "End-to-end QoS Management in Self-Configuring TSN Networks," *IEEE International Workshop on Factory Communication Systems - Proceedings, WFCS*, vol. 2021-June, pp. 131–134, 2021.

List of Figures

1 Cloud deployment models, distinguished by infrastructure ownership and permission to access resources 10
2 Cloud Continuum integrating computational resources available from multiple Cloud, Edges, and On-premises. 19
3 Levels managed by Provider or by Customers in the different Cloud Computing Service Models 20
4 Cloud Computing models of service with FaaS as an intermediate offer between PaaS and SaaS. 25
5 Zoom-in showing in details differences among PaaS and FaaS models in terms of duties managed by provider or customer. . . . 27
6 High-level FaaS architecture components. 28
7 High-level decomposition of a FaaS controller into its constituting component addressing different orchestration and management duties affecting infrastructural and application layers. 29
8 FaaS architecture introducing a MOM as a medium for event delivery to Function Executor nodes. 31
9 High-level FaaS Invoker architecture approaches: (a) per cluster, (b) per node, and (c) per function, respectively. 33
10 Function composition approaches. (a) Reflective invocation: a third entity coordinates the function invocation and forwarding of the result to the successive function in the chain. (b) Continuous Passing at the Business Layer: the business code directly invokes the next function in the chain through the associated trigger (c) Continuous Passing at the Infrastructural layer: the invoker is tasked to forward the output of the business logic to the next function in the chain. 35
11 High-level vision of the APERTO5.0 architecture in terms of the layers connecting Providers to Customers. 47
12 APERTO5.0 layers more detailed view. All components are put together for a more comprehensive effort of integration and synergy coordination. 51
13 Serverless platform as enabling technology exploited in APERTO5.0 plane to speed up service and data integration 55
14 APERTO FaaS architecture partitioned in functional Layers. . . . 58

15 Different deployment options of Trigger: A) locally to source, B) in the middle between multiple sources and the MOM C) locally to the MOM as a bridge to other event systems 60
16 High-level APERTO FaaS architecture highlighting decentralization of controller process and removal from component interactions chain in workflow executions. 65
17 High-level architecture of persistence layer integration in APERTO FaaS showing the SPS operating as an abstraction and optimization layer for data operations requested in functions 70
18 Our decentralized architecture for access control verification integrated into APERTO FaaS platform environment. 75
19 High-level vision of TEMPOS integration in APERTO FaaS architecture showing QoS differentiation based on TEMPOS channel composition 83
20 DIFFUSE relies on the peculiarities of multiple MOM solutions to provide enhanced function-to-function communication. 91
21 DIFFUSE relies on a MoM-based approach for function-to-function communication; invokers directly trigger execution of the next function by publishing function output on the corresponding topic. 93
22 High-level architecture of SPS showing the interaction sequence from the function activation to the query of data storage and ending with function termination and return of result 105
23 TEMPOS spanning different logical levels and orchestrating physical resources and component configuration in order to achieve QoS differentiation. 110
24 Mean execution times for the different invocation methods gathered in a run of 5 min. Each run repeated on nodes A, B, and C. 121
25 Average end-to-end response latency, function execution time, and database operation latency of the various configurations. The system is subjected to a synthetic, constant load of 20 requests/second, assessing the performance of an (a) read operation and a (b) create operation (log scale). 125
26 Log scale representation of the end-to-end response latency of the different solutions assessed when subjecting the system to an increasing rate of (a) read operation and a (b) create operation requests starting from 1 and reaching 16400. 126

27 End-to-end latency of access verification at the (i) trigger, (ii) invoker, and (iii) function level when the requests are granted (a) and denied (b). 128
28 Comparison of end-to-end latency performance between centralized and decentralized access verification when scaling the number of executor nodes from 1 to 4. 130
29 Comparison of end-to-end latency performance on the *log scale* between centralized and decentralized access control verification of increasingly complex policies. 131
30 First testbed section showing performance of Delivery slice. Average end-to-end latency of best effort and Strict effort traffic when executed in separated environments. 136
31 First testbed section showing performance of Delivery slice. Average end-to-end latency of best effort and Strict effort traffic with 1 and 3 concurrent best effort producers and 1 strict effort. 138
32 Testbed results of concurrent invocation of functions configured with different QoS using the *Dynamic Loaded Functions* invocation model 139
33 Testbed results of concurrent invocation of functions configured with different QoS using *Function Spawn* invocation model. . . . 140
34 Testbed results of concurrent invocation of functions configured with different QoS, using *WASM Function* invocation model. . . . 141
35 End-to-end test performance of the TEMPOS platform operated in two different scenarios: isolated execution of workflows with different QoS (1 BQ and 1 SQ), and concurrent execution of them. Comparison of end-to-end latency averages for BQ and SQ traffic executed both separately and simultaneously with an increasing number of messages/seconds (from 10 to 1000). 142
36 End-to-end test performance of the TEMPOS platform operated in two different scenarios: isolated execution of workflows with different QoS (1 BQ and 1 SQ), and concurrent execution of them. Zoom-in on end-to-end latency results showing single contributions of TEMPOS components (execution time) to the overall response times 143
37 End-to-end latency at a steady regime. 147
38 End-to-end latency and throughput with a varying rate of ingress traffic.. 149

39 End-to-end latency and throughput with a varying rate of ingress traffic.. 150
40 Response time CDF MoM comparison for a composition of length 3. 152
41 Response time CDF Kafka performance in isolation for a composition of length 3 153
42 Load balancing behavior of the different MoMs 154

List of Tables

1 MoM technologies with respective Delivery semantic, Delivery Order, and Load Balancing capabilities. 116
2 Specifications of the nodes used for the processing performance evaluation testbed. 120
3 Number of invocations executed by the different invocation methods during the processing test (5 min. run). 122
4 Specifications of the nodes used for the evaluation testbed. 132