Federated Architecture for Serverless Platforms Aimed at Transparent Execution in the Edge-Cloud Continuum

Vojdan Kjorveziroski[1*] and Sonja Filiposka[1]

[1*]Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University, Rudzer Boshkovikj 16, Skopje, 1000, North Macedonia.

*Corresponding author(s). E-mail(s): vojdan.kjorveziroski@finki.ukim.mk;
Contributing authors: sonja.filiposka@finki.ukim.mk;

Abstract

The stateless nature of serverless computing makes it a viable choice for establishing the long-desired edge-cloud continuum. Current efforts to provide a unified view over both the cloud and the edge are vendor-centric, with proprietary interfaces. This makes interoperability between different infrastructures difficult, while also raising questions about future-proofing. To overcome this problem, we introduce a federation layer which provides a unified view over distinct edge-cloud solutions. We summarise the current open questions and define a set of functional requirements for a unifying federation layer. We identify the main pillars required for fulfilling the requirements from a technical perspective and discuss concrete implementation approaches. Finally, we also showcase a practical verification of the architecture, leveraging a federation of geographically distributed Kubernetes clusters. The verification is done using multiple serverless runtime options, with computing environments scattered both in the cloud and in the edge, overcoming various connectivity restrictions in place.

Keywords: serverless computing, edge-cloud continuum, orchestration, webassembly, federated infrastructures, kubernetes, compute clusters, distributed systems

1 Introduction

One of the more prominent paradigms which have evolved significantly in recent years thanks to the cloud revolution is serverless computing [1]. Formally defined as a symbiosis between Function as a Service (FaaS) and Backend as a Service (BaaS) [2], it allows developers to write concise units of code, unburdening them from dealing with the underlying execution environment. Infrastructure aspects not directly related to the code logic itself such as scaling, runtime environment setup, and isolation are the responsibility of the cloud provider offering execution of the developer written functions, as a service. External dependencies including databases, message queues, and notification services, are also provided by the cloud platform and are placed under the umbrella of BaaS offerings, allowing simple integration with third party functions through the use of an application programming interface (API) [3].

Parallel to these serverless developments in the cloud, another seemingly unrelated area that has also experienced astronomical growth in recent years is the Internet of Things (IoT) [4]. Distinct classes of IoT devices require fast response times which are not achievable with traditional cloud providers whose data centers might be on different continents, thus introducing very high communication latency overhead. A natural solution to this problem is to simply move the compute infrastructure closer to the data source, to the edge of the network, thus reversing the current "ship data to code" principle into a "ship code to data" strategy [5]. While it is intuitive that this will overcome the latency problem, when implemented in practice, a number of questions arise [6, 7]. Perhaps one of the most burning ones is how to empower developers to easily employ this "ship code to data" strategy, while abstracting away details about the underlying infrastructure distributed across different locations.

One option that has the potential to allow seamless code shipping to different parts of the network is the adoption of serverless computing. Until recently, serverless computing was seen as a cloud-centric paradigm, with a focus on massively scalable workloads. However, the event-driven workflows typical for IoT devices can also benefit from serverless' advantages [8], such as:

- effortless scalability down to zero instances during idle time of the functions, preserving both power and compute resources, leading to more efficient utilisation of the limited compute capacity associated with resource constrained infrastructures;
- improved developer experience allowing easier implementation of new features through granular functions, as well as simplified maintainability of existing code;
- granular billing for only the compute resources which are actually being utilised, while scaling down idle ones.

It should be recognised though, as with any new paradigm and complex system, that simply moving serverless computing from the cloud to the edge of the network is not a solution to all open questions today. There are still workloads with significant processing requirements which require the extensive computing capacity available in the cloud. Migrating such workloads exclusively to the edge of the network would be counterproductive since any gains in terms of the reduced latency would be offset by longer execution times. Additionally, certain complex scenarios might require low latency during one phase of their execution, but require high compute performance

during another, thus blurring the lines between the seemingly discrete cloud and edge infrastructures. To overcome this problem, the concept of an edge-cloud continuum can be introduced where cloud infrastructure is interconnected with edge sites in a transparent way to the users of a particular platform. In this manner, a given workload can be dynamically scheduled on the most optimal infrastructure for its execution, taking into account performance, latency, and cost requirements. Even though a relatively new concept, there are a number of edge-cloud continuum initiatives today, some even from the major cloud providers [9–11].

The main drawback of existing continuum implementations is the fact that the majority of them are proprietary, locking users into the ecosystem of a single infrastructure provider, forcing them to use both cloud and edge offerings from a single source [12–14]. In a sense this is paradoxical, since one of the driving points of adopting edge computing, apart from latency reduction is the possibility to avoid the centralisation of the cloud. By locking users into a single edge-cloud ecosystem, this benefit is voided, a practice leading to the development of centralised (from an infrastructure provider perspective) continuum silos. This is a notable research gap which if bridged would allow the creation of heterogenous computing continuums, improving both the quality of the infrastructure itself, but also its reliability and cost effectiveness from consumers' perspective.

One solution to breaking down the continuum silos is to introduce a layered design, where a unifying layer would allow seamless inter-connectivity between different continuums, no matter where they are located or who they are managed by. Such a layer would act as a uniform interface between the users who require a performant and cost-effective infrastructure for deploying their functions on one hand, and the various edge-cloud continuums already established today on the other. In essence, this vision entails creating a federation across multiple edge-cloud continuums.

To this end, the goal of this paper is to analyse approaches in which provider agnostic edge-cloud continuum federations can be established, free from vendor lock-in. Motivated by, and leveraging, state-of-the-art research done in the last couple of years, we design a concept architecture for federated, widely distributed, multi-tenant serverless infrastructures spanning both the edge of the network and the cloud, offering easy extensibility and network slicing. We identify the main challenges hindering the implementation of such federated infrastructures, and offer ways to overcome them. As such, the main contributions of this paper are:

- Derive a set of functional requirements for a multi-tenant, federated, and highly scalable serverless infrastructures spanning both the edge of the network as well as the cloud, providing a unified view over federated edge-cloud continuums;
- Design and develop a vendor-neutral architecture conforming to the defined functional requirements, leveraging the latest findings in the related research fields;
- Identify the required set of open-source tools and technologies for implementing the proposed architecture in practice, including novel orchestration of WebAssembly serverless workloads across the continuum federations;
- Validate the proposed architecture and discussed implementation approaches, reporting on experience from applying them in a real-world context and providing the means for its replication in both existing and new environments.

The rest of this paper is organised as follows: in Section 2 we reflect on related work, discussing the state-of-the-art literature in the fields of federated infrastructures, edge-based serverless deployments, and edge-cloud continuums. We then proceed with Section 3 where we derive the functional requirements for federated serverless infrastructures, spanning both the edge and the cloud, with the goal of establishing a unifying layer over existing edge-cloud continuums. In Section 4 we discuss the foundational pillars necessary for fulfilling the previously laid out requirements, outlining the technologies for doing so. We then continue with Section 5, where we analyse an example use-case, mapping concrete tools and methods to each of the pillars, thus meeting the requirements. We report on a real-world verification of the complete architecture in Section 6. Finally, we conclude the paper with Section 7, reflecting on the discussed issues and drawing conclusions, setting the stage for future research in this area.

2 Related Work

The design of a federated serverless infrastructures that encompass both the edge and the cloud is an elaborate task, potentially spanning multiple research subareas including orchestration, edge-cloud-continuum, multi-cloud systems, and multi-access edge computing (MEC). In this section we discuss the latest advancements on these topics, evaluating the progress made in terms of establishing versatile edge-cloud continuums.

Beginning the discussion with multi-cloud infrastructures, Saxena et al. [15] provide a comprehensive overview of possible multi-cloud architectures, requirements, and challenges relevant to the implementation of such complex distributed systems. Their work identifies cost saving and elimination of vendor lock-ins as one of the motivations for adoption of multi-cloud systems in the real world. These advantages are well accepted and described by other existing works as well, and the authors of [16–18] directly focus on them. In all three works an additional, unifying, abstraction layer is introduced to the APIs of common public cloud platforms, allowing users to specify application requirements in a declarative way and then to schedule the workloads across multiple clouds, optimising for either cost or performance. Unfortunately, a common limitation in all of these works is that they only cater to public cloud infrastructures, foregoing integration with private, self-managed, cloud systems. This also has an implication on the conducting of the evaluation, as all of the tests relate to the execution of serverless functions within propriety runtime environments with no customizability options.

Continuing with the discussion on how such multi-cloud infrastructures can improve application performance, Smith et al. [19] and Vasconcelos et al. [20] independently present designs for utilising multiple cloud infrastructures in a serverless context, making execution decisions on-the-fly for each request, instead of statically during their initial scheduling of the workload. They realise this goal with the development of a proxying component capable of routing requests to multiple backends which are actually hosted on distinct cloud providers. The advantage of this approach is that each request can be routed to the most optimal instance for its execution, depending on the geographical location and current network conditions. Both works leverage the popular open-source serverless platform OpenFaaS on top of Kubernetes

for their demonstrations. No discussion on alternative runtime environments apart from containers, which are natively supported by OpenFaaS, is given. Such inclusion of additional runtime environments can help in cases with limited resources, where containers might not be the most suitable choice.

The use of the most popular container orchestrator today, Kubernetes, is a common theme across a number of papers dealing with complex distributed infrastructures. Many researchers have decided to leverage the tried and tested features of Kubernetes and adapt or in some cases extend them to completely new scenarios. Faticanti et al. [21] describe the use of federated Kubernetes clusters placed in different geographical regions, centrally managed using KubeFed [22]. KubeFed is now a deprecated project which allowed central management of multiple Kubernetes clusters, where a single central control plane interacts with the Kubernetes API of each of the distributed clusters to orchestrate container workloads. The authors introduce two different scenarios. The first one revolves around the use of a single Kubernetes cluster with a dedicated control plane and distributed worker nodes in different geographical regions. The second scenario leverages the capabilities introduced by KubeFed, federating multiple, initially independent Kubernetes clusters, into a single federation, thus providing a unified view of the available resources. In both cases the focus is on deploying microservices in traditional container-based runtime environments without elaborating on different deployment models such as serverless which might contribute to further simplification of the deployment process. Their results show that the non-federated cluster is not able to provide fault tolerance in terms of network issues or control plane faults. The authors of [23] also validate the use of KubeFed in practice, federating multiple Kubernetes clusters running in the cloud. The main area of interest here is the management and monitoring of such federated cluster environments instead of workload deployment and execution performance.

Closely related to the issue of infrastructure federation, especially in terms of interconnecting multiple complex environments such as Kubernetes, is the problem of how to initially configure all of the individual environments, before joining them together using a unified management solution. This is especially important for MEC scenarios, where the number of clusters can be astronomically large, as a result of the granular deployment of compute infrastructure collocated with 5G base stations. Ungureanu et al. [24] aim to overcome exactly this problem, by testing multiple approaches to creating new Kubernetes clusters at the edge. They evaluate the use of both the Cluster API (CAPI) and K3s as means of efficiently scaling the number of Kubernetes deployments. CAPI is a Kubernetes project which can leverage the public APIs of either public or private cloud providers to autonomously deploy the necessary infrastructure and establish a Kubernetes cluster on top of it. K3s is a lightweight Kubernetes distribution targeted at the edge, which simplifies the deployment of Kubernetes clusters, providing default configuration for many of the previously complicated aspects related to new deployments. In this example the focus is on the deployment models for Kubernetes clusters and how different strategies impact the overall resource consumption and performance of 5G cloud-native network functions. The network functions themselves are deployed as containerized micro-services. No discussion on whether the

proposed architecture can be adapted to other deployment models, and how, is provided. The use of CAPI has also been verified in a real-world scenario by [25] through the deployment of 35 clusters for 5G workloads at the edge. The authors have focused on the management aspects of the clusters dedicated to running 5G applications, but also without discussing a scenario-agnostic framework which could be repurposed in a generic way. To alleviate the persistent problem of finding suitable testing environments which are sufficiently large and reflect real-world MEC usage, often generated synthetic datasets can be used instead [26].

While most works focus primarily on the use of public cloud platforms, Risco et al. [27] present a solution capable of utilising both public and private cloud environments for serverless workloads. They discuss the use of Kubernetes clusters which are deployed on a private on-premise infrastructure for running serverless functions. An automated cloud bursting behaviour is developed which can leverage the AWS Lambda serverless offering to achieve higher levels of throughput and elasticity. Using this approach a continuum is established between the local resources and the cloud, although only a single on-premise Kubernetes cluster is mentioned, without specifying how multiple Kubernetes clusters can be joined together to augment the cloud bursting behaviour.

Table 1 provides a structured overview of the different topics tackled by the papers discussed above, classifying them in terms of the selected service providers, utilized deployment methods, number of sites used in the evaluation, the application scenario, and the runtime in which the workloads have been executed. Only papers directly related to infrastructure aspects are included, omitting any previous references to software tools or evaluation methodologies which were used as further clarification.

It is evident that existing efforts have been made to introduce multi-cloud, serverless federated infrastructures, with the primary goals of reducing costs, eliminating vendor lock-ins, or improving performance. However, to the best of our knowledge, no comprehensive work exists which covers all currently recognised open-issues when it comes to unifying different edge-cloud continuums. Contemporary work that focuses on federated infrastructures does not tackle the issue of slow start up times for serverless functions, or discuss possible options for interconnecting the federated clusters from the network perspective. While the theoretical concepts are sound, in some cases now deprecated software projects have been used. Our aim is to close this gap, and address these open issues in details, using modern software components, leveraging research results from previous works.

3 Functional Requirements

The current state of computing revolves around the idea of centralised providers which offer a variety of products that seamlessly integrate with one-another, either at the edge, in the cloud, or at both locations. While this makes it easy for customers to get started and set up their infrastructure initially, it has the potential to raise significant challenges in the future, depending on the nature of the requirements and the type of workload that is being executed. These challenges can manifest themselves in the form of: incompatibility with other services, vendor lock-in, suboptimal billing policies,

Table 1 Comparison of related papers

Paper	Inf. Type[1]	Deployment M.[2]	Sites[3]	Scenario[4]	Runtimes[5]
[16]	commercial	serverless	3 sites	sleep[a]	proprietary (Go)
[17]	commercial	serverless	3 sites	NLP[b], media, arithmetic, sleep	proprietary (Go, Python)
[18]	commercial	serverless	3 sites	web, media, ML[c]	proprietary (Python)
[19]	mixed	serverless	4 sites	web, media	proprietary & containers
[20]	private	serverless	2 sites	web	containers (Node.JS)
[21]	mixed	micro-services	2 sites	web	containers
[23]	commercial	micro-services	2 sites	N/A	containers
[24]	mixed	micro-services	2 sites	Open5GS	containers
[25]	mixed	micro-services	35 sites	5G use-cases	containers
[27]	mixed	serverless	2 sites	scientific workloads	proprietary & containers

[1]Infrastructure type. Possible options: private, public (commercial), mixed.

[2]Deployment model. Possible options: serverless functions, micro-services, monolith applications.

[3]Number of included sites in the federation, if any.

[4]Scope of the evaluation scenario.

[5]Supported runtime environments. The programming language used to develop the tests is provided in brackets, if known.

[a]Simple function that simply blocks for a predetermined amount of time

[b]Natural Language Processing

[c]Machine Learning

limited support for certain technologies, complex scalability, and even sustainability issues, should the service provider limit the use of certain regions or stop operations all together.

In this section we outline the functional requirements for a versatile, federated, edge-cloud continuum infrastructure spanning different geographical regions which are seamlessly interconnected with one another at the network level. Having in mind the complexity of the issue at hand, we focus only on the infrastructure aspects and omit discussion on the use of concrete workload scheduling algorithms. This research field is very dynamic and a variety of workload scheduling and migration algorithms already exist in this space [28, 29] targeting both the edge of the network and the cloud while at the same time supporting different runtime environments [30, 31]. The goal is to make the resulting systems even more versatile by allowing experimentation and use of any workload scheduling algorithms, either existing or new, instead of forcing the use of a particular one.

Putting in place such an infrastructure requires the development of an additional layer which would provide a unified view and management over globally distributed resources, no matter where they are located. This would allow any interested party such as: organisations, communities or even individuals to set up a highly performing infrastructure that would be able to satisfy the requirements of even the most demanding tasks. To accomplish this, serverless functions would be distributed across the edge and the cloud, depending on a smart scheduling system taking into account

the latency and computing needs of each task, and matching them to the underlying infrastructure which is currently at disposal. Taking into account the hierarchical nature of such a system, the actual functions would be transparently (from the user perspective) distributed across the federated cloud continuums offered by either private or public infrastructure providers. Utilising highly-performant and redundant inter-connectivity between the continuums participating in the federation would help to introduce high-availability through a seamless failover process, as well as easy extensibility and adoption of additional computing capacity. Fig. 1 provides an overview of the hierarchical relationships between the different involved components.

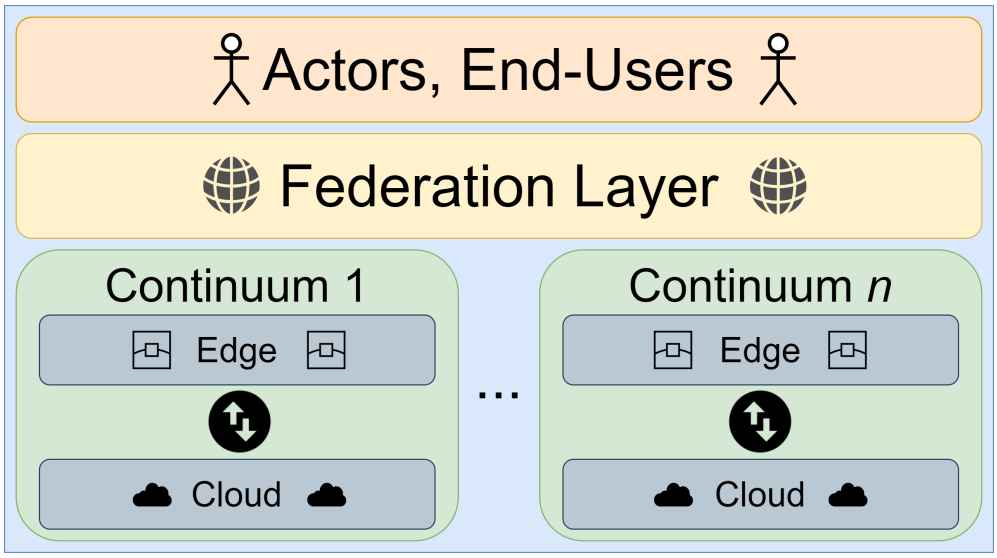

Fig. 1 Overview of the hierarchical relationships between the different components involved in the federation

The introduction of a unified federation over the existing distinctive and opinionated cloud continuums could be beneficial for any use-case where there is a large set of geographically distributed set of users, requiring both low latency and vast compute power. The serverless paradigm has a potential to deliver all of these requirements via its inherent scalability and elimination of local state, instead relying on BaaS systems for persistence. Such behaviour would allow seamless migration across the whole federation.

To this end, concrete functional requirements for a federated serverless environment that can span across the edge-cloud continuums are discussed below. These requirements have been defined based on current open questions in existing literature, practical limitations of contemporary solutions, as well as gathered lessons learnt. Although numbered, the requirements are not ordered based on any criteria. Instead, the identifiers are there to simply make it easier to reference each of the functional

requirements when discussed later in the text, and they should be deemed as having equal importance.

- **F1: Geographical Transparency** - Compute devices can be incorporated into the environment no matter their current geographical location, network, or infrastructure provider. For increased flexibility during deployment, it should also be possible to incorporate devices that reside in restrictive networks, where traffic is excessively filtered. This makes it possible to decouple the infrastructure deployment from the underlying network, allowing reuse of the existing network connectivity as is, without reconfiguration, in a fast and secure manner. Depending on the network conditions and performance requirements, multiple deployment topologies should be supported, spanning both the edge of the network and the cloud.
- **F2: Multitenancy** - The full potential of a federated serverless environment can only be realised if it is multitenant, allowing independent actors to leverage it for executing potentially untrusted functions. To this end, each actor should have access to a dedicated environment with strong isolation measures in place both from the perspective of the underlying network, as well as in terms of the runtime technologies used for executing the serverless functions.
- **F3: Elasticity** - Addition of new, but also removal of existing spare computing capacity should be a seamless task, allowing the platform to adapt to current demand. The elasticity aspects also extend to the overall resiliency when faced with unforeseen compute or network failures, ensuring the high-availability of the federation by balancing the load across the remaining infrastructures.
- **F4: Runtime Transparency** - A wide-ranging support for multiple independent serverless runtime technologies should be in place, with the possibility of extending the set of supported runtimes in the future, as new solutions are introduced. Current state-of-the-art runtimes cannot independently cover all usage scenarios, and suffer from either large cold start delays, making them unsuitable for time critical serverless workloads, or reduced execution performance. Multiple runtime environment options should be explored, while striving to provide compatibility between them as well, thus making it possible to reuse existing functions. If required, migration from one runtime environment to another should be feasible, assuming that the given function code has been appropriately refactored.
- **F5: Cost Effectiveness** - Even though a federated serverless environment does not necessarily need to be used exclusively in a commercial setting, cost effectiveness also applies to operational costs, and not only to user-facing costs. As such, the underlying runtime environments where the functions are executed should be sufficiently efficient, as to eliminate the need to resort to various ad-hoc techniques to keep the required function performance to a satisfactory level.
- **F6: Ease of Use** - Deployment of serverless functions in the federated serverless environment should be as seamless as possible from the actor perspective, hiding any complicated details about the underlying infrastructure. To accomplish this, scaling, placement, and isolation decisions should be left to the platform itself. The overall architecture of the federated system should also be transparent for the end-users who directly invoke the serverless functions as part of a service that they are using. No discernible difference compared to their experience when using traditional

infrastructures, apart from improved performance, should be present. Interacting with services hosted in the federation should not require any new additional tools or knowledge.

- **F7: Performance** - With the adoption of F4, it is expected that support for multiple runtime environments for serverless functions will be in place. However, taking into account that different runtimes have different sets of advantages and disadvantages in terms of overall cost, cold-start delay, complexity, hardware requirements, and sustained execution performance, smart defaults would need to be established. It should be possible for the correct performance profile to be applied granularly, at the function level, ensuring that the overall execution performance is kept to an acceptable level.

4 Architecture Design

The design of a federated edge-cloud continuum architecture covers multiple domains, and requires careful consideration when designing a unifying architectural solution. To better visualise the foundational pillars which would support the 7 functional requirements discussed previously, Fig. 2 depicts the various relationships between the involved components. Four distinct pillars have been identified, where Networking and Orchestration aspects extend all the way to the upper federation layer, while runtimes and clustering details are confined to each provider's implementation. The pillars themselves, as well as the way they implement the defined requirements are described in more details below.

4.1 Networking

Networking aspects are a crucial part of the overall architecture, allowing the interconnection between edge and cloud sites within a single continuum, while also providing transparent connectivity between the continuums themselves. The networking solution employed between the federation layer and the continuums will most likely be different from the one employed by a given infrastructure provider for interconnecting their edge sites to the cloud. The networking implementation spanning the federation and continuum layers should hide these peculiarities from the end-users of the federation, providing a unified, geographically transparent view of the federated infrastructure. Additionally, the solutions employed within this pillar should provide multitenancy, allowing multiple users to leverage the same federated infrastructure for different purposes. Two main aspects need to be taken into consideration when discussing multitenancy: implementing access control on the infrastructure level on one hand, and on the application level on the other.

Access control lists (ACLs) on the infrastructure level would enable granular control over which users can leverage which continuums, and regulate the communication between the continuums themselves. Furthermore, network rules would have to be applied on the application level as well, utilising the least-privilege model, restricting the communication between unrelated functions, thus lowering the overall system's exposure in the face of a compromise or malicious workloads in unison with advanced intrusion detection system (IDS) algorithms [32].

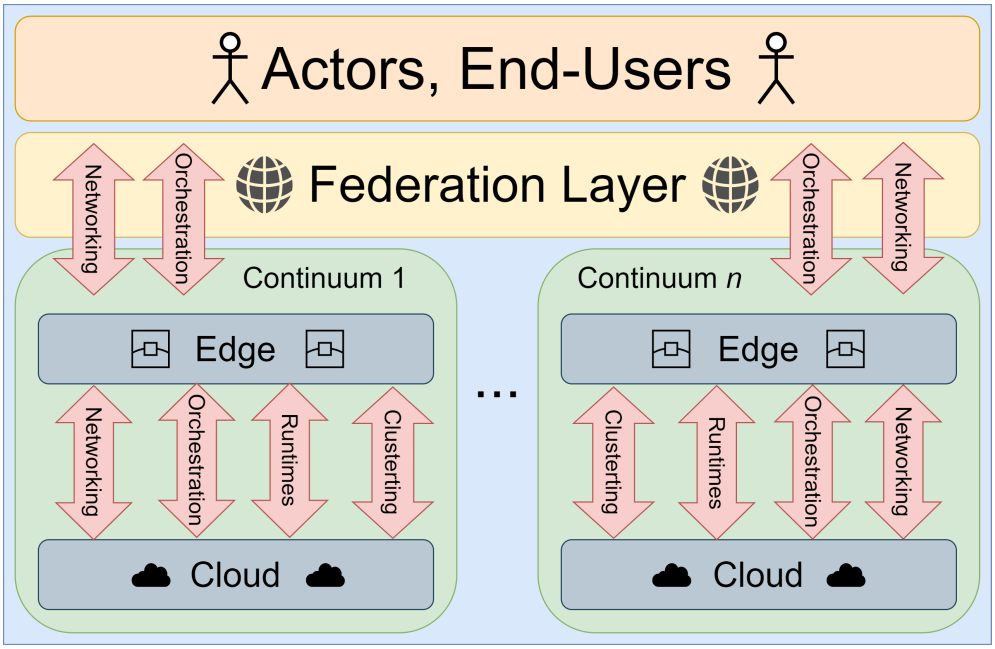

Fig. 2 Identified pillars of a unifying federated edge-cloud continuum architecture

Finally, the overall elasticity of the federation by joining additional continuums or removing existing ones is also partly dependant on the networking choices. Provisioning and deprovisioning of devices within the network should be as seamless as possible, where the overall network topology would adapt depending on the number of continuums to be interconnected. Such versatility also plays a central role in the high-availability of the overall federation, allowing it to fail-over resources from one continuum to another, should the network be robust to adjust to the new conditions.

The Networking pillar enables the implementation of the following functional requirements: F1 (Geographical Transparency), F2 (Multitenancy), F3 (Elasticity).

4.2 Orchestration

Intelligent orchestration is a critical component that plays a significant role both in the continuum layer, as well as in the federation layer. Leveraging the hierarchical nature of the proposed architecture, when a new serverless function needs to be deployed, a two-staged placement process is initiated. The advanced scheduling algorithms would first choose the most appropriate continuum that has a potential to satisfy the function's constraints in terms of latency, performance, or cost. Then, the orchestration component within the particular continuum, aware of the complete topology and available resources within the cloud and edge sites, chooses the most optimal location.

The orchestrators within the continuum should be aware of the runtime environments that are supported at each location and can also share this information summatively with the orchestration component running in the federation layer. This

would achieve both runtime transparency and make it easier to create deployments from the user perspective, where the appropriate environment would be automatically selected, based on the compatibility between the submitted function and the possible runtimes where it can be executed. The runtime transparency functional requirement also extends to the support for multiple compute architectures, allowing the serverless platforms not only to leverage traditional x86 servers, but also alternatives such as ARM or RISC-V [33, 34].

Finally, orchestration aspects are closely coupled with overall cost effectiveness, and this can be yet another factor in the final scheduling decision. As expected, not all continuums under the federation would offer the same terms, and there can be different cost implications even between different runtimes within the same continuum. Particular runtime environments can have a distinctive behaviour in terms of idle resource usage, start up time, and overall resource utilisation respectively, aspects which all need to be carefully taken into account when making the final placement decision. Efficiency of the underlying computing architecture and by extension the supported instruction sets, can also play a role.

In conclusion, the Orchestration pillar contributes to the following functional requirements: F4 (Runtime Transparency), F5 (Cost Effectiveness), F6 (Ease of Use), F7 (Performance).

4.3 Clustering

The infrastructure at either the cloud or the edge within a given continuum is inherently acting as a distributed system, comprised of multiple nodes which need to interact with one-another. In the proposed architecture, it is the responsibility of the clustering pillar to facilitate overall geographical transparency of the compute resources, ensure seamless elasticity in terms of scaling the compute clusters either up or down, all the while providing the expected level of performance.

Since the clustering pillar is confined to the continuums themselves, it is expected that their internal topologies can differ significantly between one-another. The impact of these differences might depend on the extent to which tight or wide clusters have been employed. A tight compute cluster is a cluster where all of the constituent devices are organised in a flat hierarchy, are usually nearby, and have direct local network connectivity between each other (e.g., nodes placed at a single edge site). Wide clusters on the other hand, are clusters where the participating devices are placed in different networks, and intra-cluster communication between the nodes traverses multiple intermediary networks, in some cases even over public links. While one benefit of such wide clusters is easier scaling of resources since the hardware itself is decoupled from the underlying network, allowing each to be owned by a separate entity, careful attention needs to be paid to the overall connection latency between the participating nodes. High delays might impact the integrity of the cluster in addition to reducing its performance. In the case of wide clusters, attempts should be made not only to rely on close geographical proximity, but also to reflect on peering agreements of the networks in case nodes are placed in different autonomous systems (AS).

No matter the topology of the involved clusters, it needs to provide effortless addition of new and removal of existing computing nodes, forgoing complex provisioning

processes or manual reconfiguration by an operator. The ability to rapidly extend the available computing capacity through addition of new nodes can also positively impact the performance during peaks in customer demand. Any details of such topology reorganisations, no matter whether done at the edge or in the cloud, should be confined to the given continuum, thus being transparent to the federation layer. Since it is expected that infrastructure originally part of different continuums or simply managed by different entities can join the overall federation, robust monitoring strategies need to be in place to ensure the expected performance levels of all involved components [35].

In summary, the Clustering pillar plays a role in meeting the following functional requirements: F1 (Geographical Transparency), F3 (Elasticity), F7 (Performance).

4.4 Runtimes

Adding support for multiple runtime environments at the edge and cloud sites within the continuums would ensure that the overall federation is both suitable and easy to use for a wide variety of use-cases. It is expected that different runtimes would have different performance implications, but also impact what serverless functions can be executed at which location. Certain runtimes might be optimised for security, execution speed, compatibility, or cost-effectiveness.

Security aspects during function execution, which are also dependant on the isolation features offered by the underlying runtime, play an important role in realising the multitenancy functional requirement. To this end, the Runtimes pillar enables the option of not reusing the execution environments between independent function executions, thus allowing per request isolation for selected serverless workloads which are deemed highly sensitive. In such cases, each request is handled by a separate function instance, executed in an isolated environment. This also eliminates any potential of unwanted data leaks between independent invocations of the same function, since the ephemeral execution environment is torn down and recreated for each call, thus clearing any left-over temporary files and memory content. Such multitenancy requirements, both from the network and execution perspective, go hand-in-hand with the slicing concepts which are relevant for multiple use-cases, including MEC, allowing easy partition of the serverless environments offered by the continuums, and by extension the overall federation across multiple independent actors.

When it comes to cost-effectiveness, true scale-to-zero behaviour is also essential, preventing any instances of a given function to be active when no inbound requests are received. The adoption of such strategies benefits both the users and the operators. The users are billed only for the time that their function has spent doing meaningful work, while from the operators perspective, the same hardware can accommodate more functions, leading to better utilisation and power efficiency.

Finally, it is expected that no single runtime environment can satisfy all of the aspects by itself, and at the end a careful decision based on a thorough analysis of the advantages and drawbacks of each option needs to be made when selecting the concrete option, as discussed within the orchestration subsection. Nonetheless, it is up to the continuums to support a diverse set of options, accommodating different use-cases, programming languages, and requirements.

The Runtimes pillar caters to the following functional requirements: F2 (Multitenancy), F4 (Runtime Transparency), F5 (Cost Effectiveness), F6 (Ease of Use), F7 (Performance).

5 Use-Case Example

Based on the functional requirements laid out in Section 3 and the distinctive pillars which support them, presented in Section 4, we discuss possible implementation routes below. The proposed solution is intentionally Kubernetes-centric with the aim of reusing existing work already done in terms of this ubiquitous orchestrator today, some of which is directly related to the area of serverless computing [36]. However, those Kubernetes related aspects are only relevant for platform operators, and attention was paid as to not require regular users to be aware of the underlying infrastructure or network topology when deploying functions. As partially discussed in Section 2, existing works have already verified the applicability of Kubernetes in real-world serverless scenarios [27, 37].

In the subsections that follow we will be applying the bottom-up approach, discussing concrete solutions that map to each pillar, starting from the continuum layer and then moving towards the federation layer.

5.1 Kubernetes as a Clustering Solution

We propose the use of Kubernetes to fulfil the clustering requirements within the different continuums, since the majority of private and public infrastructure providers already offer first-party support for this orchestrator. Additionally, Kubernetes is an open-source project that is widely popular and has amassed a large community of users and developers since its first stable release in 2015, ensuring its longevity and continued development.

When it comes to technical details, Kubernetes clusters by default are comprised of at least one master node representing the control plane responsible for the functioning of the cluster, and one or more worker nodes which execute the actual workloads. The worker nodes constantly interact with the control plane, reconciling the actual cluster state with the desired state as specified by the Kubernetes API. As such, robust network connectivity needs to exist between participating cluster nodes.

Kubernetes clusters can be managed using a variety of tools [38], ranging from manual ones which require the cluster operator to be aware of the whole stack, to completely automated solutions. In many cases different infrastructure providers will offer different tools to manage the clusters on their infrastructure. When it comes to an approach that is both provider and location agnostic, the recommended way for provisioning new Kubernetes clusters, as well as extending existing ones is through the use of the K3s Kubernetes distribution [39]. K3s is a lightweight distribution with significantly lower hardware requirements than a full-fledged Kubernetes deployment, without sacrificing on available features. Existing research has also shown that K3s offers very competitive performance compared to other lightweight alternatives such as MicroK8s [40], as well as traditional Kubernetes deployments.

A notable alternative to K3s which can be utilised for easy and independent Kubernetes cluster deployments is the Cluster API (CAPI) [41]. CAPI is a project which has support for automated Kubernetes deployments on multiple private and cloud providers, leveraging their APIs. Its use has already been validated by related MEC papers [24, 25]. However, CAPI is not directly targeted at edge deployments, thus making K3s the preferred choice when it comes to environments that should span both the cloud and the more resource constrained edge of the network.

The simple provisioning of both master and worker nodes using K3s guarantees robust scaling mechanisms. Worker nodes where the actual applications are being executed can be seamlessly added or removed at any point in time, without affecting the operation of the complete cluster. However, one question that arises is how to choose between the previously introduced tight and wide cluster deployments. A wide deployment of cluster nodes where the workers are placed on different infrastructure to the control plane without local area connectivity between them, should only be performed when network conditions, including the measured latency, permit for such distributed deployments. We refrain from giving exact thresholds on when it is permissible to use wide clusters since requirements can vastly differ from one case to the next. We simply point out that too high latency between the worker nodes and the control plane can make the cluster unstable, which can also have negative effect on the existing functions running on it, as well as when (re)scheduling functions. Depending on the use-case, official recommendations in terms of latency limits should be followed. As an example, when discussing MEC deployments and 5G, the IMT-2020 recommendation on latency requirements for 5G sets the maximum user experienced network latency at 4ms when talking Ultra-Reliable Low Latency Communications (URLLC) [42]. Similar limits have also been defined by working groups or industry bodies for various other use-cases.

Another thing that is worth pointing out is that Kubernetes in the past was indeed first and foremost a container orchestrator. However, as discussed previously, the vision for a federated serverless environments requires flexibility in terms of runtime environments and should not be constrained to a single implementation. Building upon recent advancements in the area of runtimes, it is now possible to leverage the underlying Containerd container runtime and integrate it with additional external runtime environments. In this way, the Containerd API acts as a unifying layer over other distinct environments such as virtual machines, micro virtual machines, or WebAssembly. From the clustering perspective, Kubernetes is only integrated with Containerd, and does not require any source code changes to support other runtime environments in addition to containers. Deploying clusters in the continuum with support for more than one runtime environment ensures that the overall performance requirements of the various workloads to be deployed will be met using the most efficient approach. We discuss possible alternative runtimes and how to integrate them in practice with Containerd in Subsection 5.2 below.

5.2 Runtime Options

Different continuums part of the federation can support different runtimes. It is the goal of the federation layer to provide a unified interface and thus hide these differences from the end users.

As discussed previously, containers are the default runtime option for Kubernetes clusters and one of the most popular options when executing serverless workloads [43, 44]. By design, containers are confined to their own process and network namespace, providing isolation with regards to the parent operating system. For real-world deployments, careful attention needs to be paid to the used container base images for building the serverless functions, as well as the privileges with which the workloads are executed within the container. Incorrectly configured permissions, or running as a root user, can jeopardise the compute infrastructure.

The main drawback of containers, and the primary incentive for researching additional runtime environments, is that they have long start up times, making them unfeasible for scenarios where per-invocation isolation is required, such that each request is handled by a separate container instance. Additionally, even if containers are reused between successive executions of the same function, the cold start problem will again be relevant when it comes to scaling the number of instances. The term cold start refers to the increased start up time experienced during a function's first execution. Sub-optimal solutions to these problems include foregoing per-invocation isolation, leaving containers running, and pre-warming additional instances, keeping them ready for a burst in incoming requests. However, such approaches negate one of the main benefits of serverless computing, namely the ability to scale idle instances to zero replicas and restart them as new requests are received.

An alternative runtime environment to containers for serverless functions which has already attracted noticeable research interest is WebAsssembly [45, 46] or WASM for short. WASM as a binary instruction format is compatible with many popular programming languages. It guarantees strict isolation through its stack based virtual machine, while also having cold start times which are in the range of milliseconds, compared to start up times of containers which amount to seconds [47]. WebAssembly has support for many popular compiled languages such as C, C++, Rust, and Go. Recently, efforts have also been made to transform WebAssembly into a viable execution environment for interpreted languages such as Python and PHP [48]. In such cases, the interpreter itself should be compiled to a WASM module, which would then execute the code written in the respective programming language.

To tackle the issue of large cold start delays, we propose support for WebAssembly serverless functions to be added to existing infrastructures through the use of the Spin runtime [49]. Spin is a WASM runtime based on Wasmtime which makes WebAssembly a viable option for development of serverless functions. WebAssembly modules utilising the Spin environment can be triggered in one of two ways, either via HTTP requests or via Redis triggers. More details about Spin's features, as well as its performance when compared to traditional serverless frameworks are available in [45] and [50].

Leveraging Containerd's extensibility, Kubernetes can run Spin serverless workloads through the use of a software shim. It's worth reiterating that such a

WebAssembly integration through a Spin shim [51], does not restrict the existing containerisation functionality of Containerd, allowing both WebAssembly modules and traditional containers to be executed on the same Kubernetes nodes, managing their deployment using existing Kubernetes constructs such as "Deployments", "Pods", or "Services". The proposed implementation, through the use of the Spin framework, has support for Rust, Go, Python, TypeScript, and C#. In the future it is expected that other languages will have first party support as well [52]. Converting existing functions written in the aforementioned languages to WebAssembly modules is also a straightforward process which in most cases requires only minimal changes to the source code. Depending on the preferred invocation method (HTTP or Redis), Spin specific programming libraries might need to be used, but these libraries follow the same API as the standard libraries offered by the language itself (e.g., an HTTP library). In such scenarios, it is enough to simply alter the import statement. When it comes to more complex serverless workloads which require, for example, the use of threads or other currently unsupported WebAssembly features [53], the use of an alternative runtime, such as containers, is recommended.

Keeping in line with the ease of use functional requirement, WebAssembly modules which are meant to be executed by Spin can be packaged as regular Open Container Initiative (OCI) images, same as traditional container images. In the case of WASM, these OCI images are constructed from a single "scratch" layer which simply contains the WASM function's binary, thus making them much smaller, leading to faster instantiation and reduced download times. As a result of this, existing tools with which developers are already familiar with can be reused both for the building of the images themselves, as well as distributing them. Standard container registries can be used for hosting WebAssembly modules packaged as OCI images.

A potential drawback to WebAssembly at the moment is the reduced sustained performance when compared to containers, and potential unfamiliarity with this new technology among developers. As such, it is recommended for serverless infrastructures to support both WebAssembly and alternatives such as containers. WebAssembly can be utilised for functions which are very frequently executed and require massive scaling, but the actual workload does not require sustained compute performance [54]. On the other hand, containers are still relevant for complex cases which do require significant compute resources. With this dual-runtime approach it is ensured that the best aspects of both worlds (containers and WebAssembly) are taken into account while also minimising their drawbacks.

Similarly, the solution can also easily be extended to support even more runtime environments, such as virtual machines, through the use of Kubevirt [55]; micro virtual machines by using the Kata runtime [56, 57]; or other WebAssembly environments in addition to Spin [58].

In conclusion, the question of runtime transparency is raised due to the fact that multiple execution environments need to be supported. Current technologies such as containers, virtual machines, or WebAssembly modules do not satisfy all requirements by themselves in terms of both execution performance and satisfactory cold start

times. Utilising one well-defined API, that of Containerd, to distribute serverless functions across different runtimes, provides the possibility to ensure both ease of use and required performance, while being cost effective at the same time.

5.3 Orchestration throughout the Federation

As set out in the proposed architecture, the orchestration pillar should cover both the continuum and the federation layer, allowing centralised management of all compute infrastructure, and scheduling of serverless functions across different locations and environments. Until recently, Kubernetes had an officially supported federation project called KubeFed. However, KubeFed has recently been deprecated in favour of more modern alternatives. One such option is Karmada [59], which can also be presented as a successor to KubeFed's legacy. Karmada enables the creation of Kubernetes cluster federations, without requiring any specific deployment strategies, or special infrastructure. The only requirement is that the Kubernetes cluster is running a certified Kubernetes distribution, implementing the base APIs (which is the case across all major infrastructure providers and the proposed independent Kubernetes distributions). A central control plane which is comprised of a regular Kubernetes cluster consisting of master and worker nodes is required for deploying the core Karmada components. This central control plane is usually placed in a strategic cloud datacenter, offering satisfactory latency to the other clusters taking part in the federation. A new cluster can be added to the infrastructure in one of two ways (from the perspective of the central control plane):

- Push mode - the central control plane connects to the Kubernetes API server of the federated cluster
- Pull mode - a dedicated agent deployed in the federated cluster periodically connects to the central control plane, reconciling states.

The recommended method by Karmada is push mode, since it does not require any additional dependencies in the federated clusters, and is also more efficient than its pull counterpart which uses periodic polling. Assuming connectivity across the various locations is already in place as discussed in Subsection 5.4, push registration of all participating clusters from the various continuums into the federation is a viable solution. Tunnelling between the Kubernetes API servers using state-of-the-art Virtual Private Network (VPN) implementations also negates the requirement for opening the Kubernetes API ports publicly, thus increasing security.

When it comes to performance aspects, the proposed architecture relies on the selection of the appropriate runtime environment to optimise the total execution time and thus the overall performance: WebAssembly for frequently executed albeit simple cases, and containers for more complex functions with larger execution times and in need of increased compute performance. However, taking into account that there might be many clusters participating in the federation, as well as that end-users are not expected to be static and can migrate from one location to another, attention should be paid to the question of scheduling as well.

The problem which needs to be solved in regards to the scheduling of the serverless functions is how to route incoming requests to the most optimal instance, which

in most (but not all) cases is assumed to be the one running on the infrastructure geographically closest to the user. Such routing to the nearest replica would ensure optimal communication latency, but does not account for overloading of functions, where the execution delay may offset any time savings incurred by the reduced latency. We will be focusing on the scheduling perspective of this problem in this subsection, while tackling the Networking aspect to the solution in the next one, Subsection 5.4. The chosen federation solution, Karmada, has support for advanced container placement, including failover across multiple clusters, as well as constraint based scheduling which can take into account current node conditions during its decision making. On the other hand, from the WebAssembly perspective, such advanced scheduling and descheduling strategies are no longer required, since function instances are executed from scratch upon every request, and no resources are wasted during idle times. This makes it feasible to register large numbers of functions across all federated clusters, and only run them should an invocation request be received. Since resources are utilised solely while the function is being executed, overloading issues can be solved with appropriate routing decisions.

An additional advantage of serverless computing that can be exploited during the request routing and can contribute to greater cost effectiveness is the fact that function executions should be stateless. For any information that needs to be persisted between invocations, BaaS solutions should be used. Such BaaS products have a well-defined API which can be consumed by any function instance, anywhere, removing the need to keep local state within the runtime environment of a given function itself. As a result, routing decisions are much more flexible, and requests can be routed to any available function replica, eliminating the need for sticky sessions, regardless of where the previous request landed. With this approach it is also possible to always execute the functions on the most cost effective infrastructure at the moment, and be able to effortlessly move from one continuum to another as terms change over time, foregoing complex state migrations.

Finally, taking into account that all underlying components, starting from Spin, to Containerd, and Kubernetes have full ARM support, the proposed orchestration approach across the whole federation also allows for mixing and matching master/-compute nodes architectures. More advanced scheduling decisions depending on the target architecture can also be realised by using existing Kubernetes functionality such as Kubernetes labels, which can be considered during Karmada's scheduling [60]. This helps the orchestration pillar meet the expected runtime and platform transparency requirements.

5.4 Unified Networking

Providing a unified view over networking resources available across the federation, as well as within the continuums revolves around two major topics: container network interface (CNI) plugins and virtual private networks (VPNs). CNI plugins focus on providing multitenancy and intra-cluster connectivity between the different nodes comprising the given Kubernetes cluster, while VPNs help to connect remote locations, thus providing geographical transparency. We discuss both in details below.

CNI plugins can be utilised to create an overlay container network which will provide network connectivity for workloads scheduled across the Kubernetes cluster [61]. Even though the C in CNI stands for container, the same overlay network can also be reused by other runtime environments, such as WebAssembly. Depending on the already available network infrastructure, such CNI plugins can work in a variety of modes, with the perhaps most popular and straight-forward one being encapsulation [62]. The reference implementation selects Calico as the Kubernetes CNI plugin due to its versatility, maturity, and implementation of a wide range of features by default, such as support for NetworkPolicy objects capable of regulating inbound and outbound network traffic on a per function level, and multiple encapsulation methods, in addition to optional BGP peering with existing routers in the infrastructure [63]. Taking into account that BGP configuration will be an unfeasible requirement for many edge deployments, Calico's VXLAN encapsulation alternative can be used instead. Full VXLAN encapsulation of all Kubernetes pod traffic also solves the problem of (justifiably) restrictive security measures commonly present in both virtualized and physical infrastructures such as MAC address spoofing, which would otherwise have prevented the regular functioning of the overlay network.

Using a different CNI plugin other than Calico within a given continuum does not pose an issue and does not hinder the overall inter-compatibility between the various clusters. At the end of the day, the CNI behavior can be controlled using Kubernetes constructs which will be the same across all infrastructures.

Moving on to the security aspects of deploying overlay networks over untrusted underlay networks, VXLAN as other similar encapsulation techniques, does not offer built-in encryption. However, this can be elegantly solved using a VPN as the underlay, over which the actual Calico overlay will be established. One potential disadvantage to this approach is the reduced network throughput as a result of the VPN overhead, as well as a more tedious scaling process, requiring reconfiguration of the underlying VPN. Recent research into modern VPN tools has shown that modern full-mesh VPN topologies utilising the Wireguard protocol offer performance comparable to the native scenario, while offering automatic reconfiguration when adding new nodes [64]. The preferred VPN solution in the reference architecture is Headscale [65], an open-source implementation of the Tailscale control-plane [66], which utilises Wireguard and a central coordination server that distributes Wireguard public keys across participating nodes, facilitating full-mesh connectivity between them. Such full-mesh connectivity, apart from overcoming potential bottle-necks commonly associated with traditional hub-and-spoke VPN topologies, also introduces increased resiliency, especially important in the discussed case of distributed deployment of cluster nodes. While other alternatives [67–69] might offer better performance in certain cases, Headscale offers a more streamlined on-boarding experience, and advanced Network Address Translation (NAT) traversal techniques, along with full relaying, if absolutely necessary. Any feature that enables addition of extra nodes placed in restrictive networks across the federation is welcome. In extreme cases, when UDP traffic is completely blocked, and no direct connection can be established with the remaining peers, Headscale can use relaying servers, tunnelling the VPN traffic over TCP on port 443 (HTTPS), thus bypassing any restrictions [70]. Since the relayed traffic is end-to-end encrypted, the

network relays cannot eavesdrop on its content. Such relaying servers can also be self-hosted and placed in geographically strategic locations, ensuring optimal latency and throughput. Granular access control lists can also be used to restrict which nodes might form a mesh, thus adding support for network slicing.

Even though not covered by the functional requirements, it is worth mentioning that versatile full-mesh VPN solutions with support for advanced ACLs and NAT traversal can also enable edge based infrastructures to be extended by volunteering users, contributing their own computing capacity by joining their devices to the federated infrastructure.

The networking pillar also supports the orchestration solutions to efficiently route the requests to the correct replica, providing overall geographic transparency and elasticity. Continuing the discussion from the previous subsection, in order to ensure routing to the most optimal replica, the reference implementation proposes two approaches from the networking perspective: Domain Name System (DNS) based load-balancing and anycast.

In the case of DNS load-balancing, CoreDNS which is a required component for Kubernetes clusters can be used for split DNS with response policy zones. This is a viable scenario for MEC deployments, since the given CoreDNS instance can be pushed as the DNS server to be used to all devices in a given geographical region by the mobile carrier [71, 72]. The CoreDNS would then resolve the invocation URLs of the invoked functions to the respective instances running on the nearest Kubernetes cluster. Should the user move from one location to another, the CoreDNS instance will also change, resulting in the routing of requests to the nearest cluster again.

An alternative approach which can be utilised is to leverage the functionality of anycast [73]. This can be performed in one of two ways either for the CoreDNS servers or for the function instances themselves. In the first case, the CoreDNS instances running in the clusters are exposed using a single IP address which is shared among them, despite the fact that they are deployed at different locations. Network routing policies would ensure that DNS requests are routed to the nearest CoreDNS instance, which would in turn resolve the function URL to the instance running on a nearby cluster. While this approach is similar to the one previously described, it eliminates the need for explicit changes of the DNS servers whenever the user moves from one location to another.

In the second case, which also revolves around the use of anycast, each function is exposed via a load-balancer IP address which is shared across all clusters. Anycast would ensure routing to the nearest load-balancer instance running within a given Kubernetes cluster which would then, in turn, route the request to the appropriate function replica. With this approach, any existing public DNS resolver can be used by the end-users invoking the functions, regardless of its location or caching policy, eliminating the need for using a region specific DNS server.

Concluding this section, Fig. 3 presents a simplified view of the overall architecture, focusing on network connectivity between the different sites as well as the mode of interaction with the central control plane, as discussed in subsection 5.3. Depending on the connectivity options and the required security level, VPN connectivity can be used for connecting nodes in a full-mesh topology within the same site (such is the

case with Cluster B in the figure). Alternatively, VPN connectivity can only be used to reach the central control plane and intra-cluster communication can be over the local area network (Cluster A). In the case of Cluster C, as it is employs a wide topology distributed among multiple sites which are geographically close by, full mesh VPN connectivity is required between the nodes.

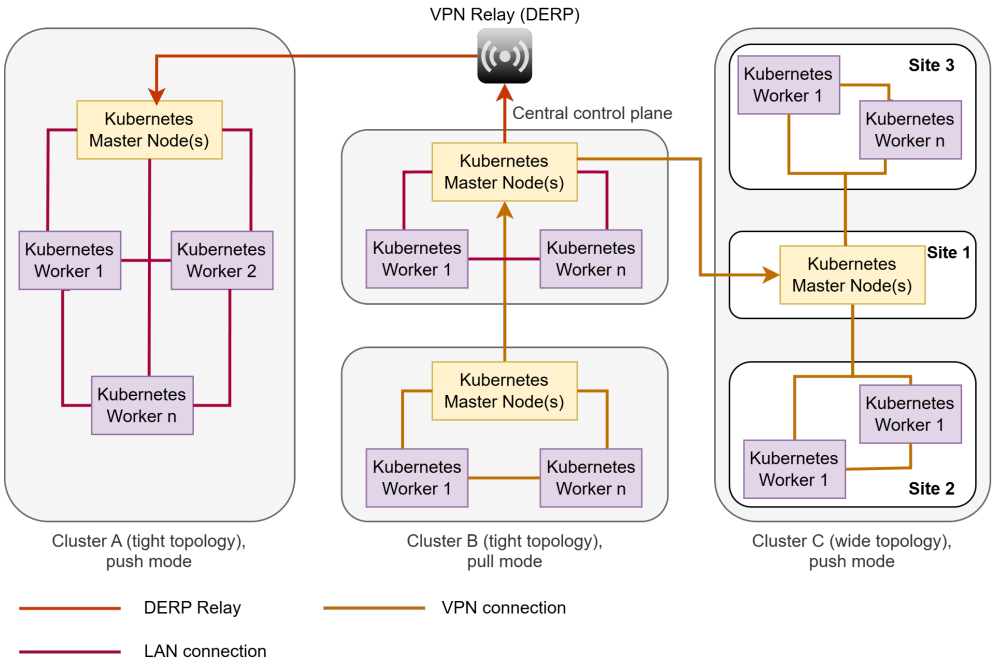

Fig. 3 Overall Architecture

6 Practical Verification

To verify that the discussed solutions mapped to the 4 main pillars satisfy the 7 functional requirements under real-world conditions, we created a federated infrastructure spanning multiple countries and continents. The deployment consisted of 5 Kubernetes clusters in total, deployed either in a tight or wide fashion, distributed across different geographical regions. Table 2 provides more details about each cluster.

Clusters 1, 2, and 3 were deployed as tight clusters, within the same autonomous system and geographical location but not necessarily in the same subnet. Clusters 4 and 5 were deployed as wide clusters on hosts placed in different autonomous systems, at various locations, but in the same geographical vicinity (at a country level). In order to simulate restricted network environments, cluster 3 was deployed behind a NAT router, testing out the NAT traversal capabilities of Headscale. Additionally, in the case of cluster 5, all UDP traffic was blocked at the firewall level, making it

Table 2 Details about the federated infrastructure

#	Location	Nodes	Type[1]	Restrictions[2]
1	N. Macedonia	3	Tight	No
2	USA	3	Tight	No
3	N. Macedonia	3	Tight	NAT
4	Germany	3	Wide	No
5	N.Macedonia	3	Wide	w/o UDP

[1]Type of Kubernetes deployment. Possible options: tight or wide.

[2]Network restrictions, if any.

impossible to establish direct full-mesh VPN connectivity to the rest of the nodes. The only viable connection option in this case was to utilise the Headscale's relaying functionality using an intermediary DERP server.

In all cases Headscale was used as the underlay network on top of which the Calico overlay was deployed, thus satisfying F1. For each cluster corresponding ACL rules were applied, forming a full-mesh with only other participating nodes in it, as well as limited connectivity to the central control plane which was hosted on cluster 1. All clusters were deployed using the K3s Kubernetes distribution, and were then federated using Karmada in the recommended "push" mode. Not all cluster nodes were added at the same time, due to the time it took to make arrangements for the resources required for the wide clusters. These circumstances opened an opportunity to test the scalability of clusters deployed with K3s, since additional nodes were added only after the cluster was established. As a result, we have validated that the requirements set by F3 have been met.

F2 and F4 were satisfied by implementing support for Spin WebAssembly modules through a Containerd shim, as already described in [50]. We have verified that Karmada is indeed capable of scheduling WebAssembly backed serverless functions, across all federated clusters, leveraging its existing scheduling methods. We focused our attention on the federation and scheduling aspects, refraining from comparing WebAssembly's performance to that of other runtimes since this has been evaluated and extensively elaborated as part of existing works [54, 74–76]. We have reused the same functions as presented in [58] and [50], which were developed either from scratch or by converting existing non-WebAssembly functions into WASM. For all functions the Spin HTTP invocation method was used, and interaction with third party BaaS offerings was made using the provided HTTP client libraries. Once compiled into WASM modules, the OCI images were built from a scratch layer using a non-modified version of Docker and uploaded to public container registries, thus validating F6.

To evaluate the last two remaining functional requirements, F5 and F7, we have deployed the same set of WASM functions across all 4 clusters (cluster 1, hosting the central control plane, was intentionally omitted, as to ensure its performance). To test the feasibility of the DNS load-balancing approach discussed in 5.4, we deployed a set of client VMs hosted in the same countries as the federated clusters. Each client VM used the in-cluster CoreDNS instance of the geographically closest cluster as a recursive DNS resolver, confirming that this strategy is indeed a viable option to route

requests to the nearest function instance. The Spin shim was executing the respective WASM module representing the given function for each new invocation, thus offering true scale-to-zero behaviour, along with per-request isolation.

All of the software versions of the various components used for the deployment of the federated infrastructure are given in Table 3.

Table 3 Details about the execution environment

Component	Version
Operating System	Ubuntu 22.04 LTS
Containerd Version	v1.5.16
Kubernetes version	v1.22.17
K3s version	v1.22.17+k3s1
Karmada version	v1.6.1
Spin version	v0.7
Tailscale client version	v1.46.1
Headscale version	v0.21.0

To quantify the real-world benefits of the proposed architecture, further analysis was performed in terms of how network latency impacts the overall execution performance of serverless functions. We used the results described in [50] as a baseline and then reevaluated them, taking network latency as an additional factor impacting the total execution time. The user-manager serverless function was chosen as a suitable representative, since it models a typical serverless workload, receiving an HTTP request and contacting an external BaaS product to persist the computed data. Fig. 4 shows the average network latencies between the three core locations evaluated during the research. In cases when multiple clusters were deployed in a single country, the average latency as measured from all locations was taken into account.

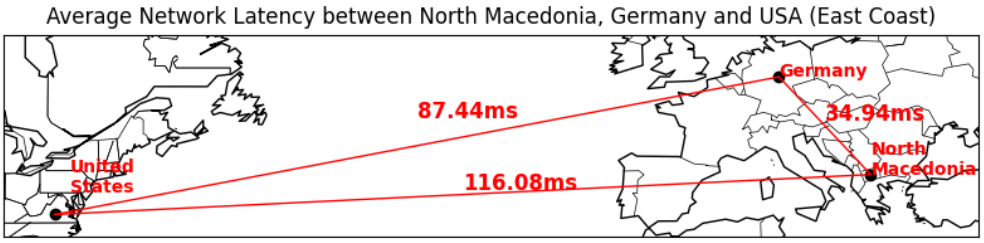

Fig. 4 Average network latency between the cluster locations

It should be noted that network latencies can vary significantly depending on the infrastructure provider, current network conditions, as well as peering agreements determining the hop count between two endpoints. Nevertheless, the goal of this analysis was to determine the extent to which serverless function placement can play a role

in the total execution time. Fig. 5 visualises how the total execution time for the user-manager function increases as it is invoked from the different geographical locations. The metrics which are presented in the figure represent:

- Network Latency - the measured network latency between the client and the infrastructure hosting the serverless function instance at the time of their communication.
- WASM Cold Start - the cold start latency for a WebAssembly based serverless function to enter a running state. Measured from the moment the HTTP request from the invoker is received and includes the time needed to select the most suitable function instance and scale it up so that it is running and ready to begin processing data.
- Processing Delay WASM - the time to process the incoming data once the WASM serverless function is ready and has received the client's request. It includes the time needed to send a response and scale down to zero.
- Container Cold Start - the cold start latency for a container based serverless function to enter a running state. Measured from the moment the HTTP request from the invoker is received and includes the time needed to select the most suitable function instance and scale it up so that it is running and ready to begin processing data.
- Processing Delay Container - the time to process the incoming data once the container based function is ready and has received the client's request. It includes the time needed to send a response and scale down to zero.

To aid the readability, the y-axis is presented in a logarithmic scale, while at the top of each bar in the figure the total execution time as measured from the moment the request was sent until a response was received is shown.

The increase in the total execution time for more than 100ms in the most extreme case clearly demonstrates the advantages of the federated architecture which would ensure that the most suitable location is chosen for each invocation in a way that is transparent to the end-user. This is even more pronounced in scenarios where the cold-start delay is not a factor due to invoking already warm instances, since the network latency can be even several orders of magnitude larger than the processing time, thus diminishing any benefits derived from runtime performance optimisation. As we focus on the federation aspects and the impact of network latency, further analysis in terms of how different runtime environments impact the execution of various functions is out of scope of this paper and is already available in [50].

7 Discussion and Conclusion

Taking into account the meteoric rise in the number of connected IoT devices available today, it is evident that neither the cloud, nor the edge can deal with all of the computing and latency requirements by themselves. A comprehensive, vendor independent, edge-cloud continuum strategy is required, spanning multiple geographical regions, and leveraging the latest state of the art technology in terms of both runtime environments and scheduling. The event-based nature of IoT workloads on one hand and serverless' statelessness on the other, make serverless functions a viable choice for implementing such federated infrastructures.

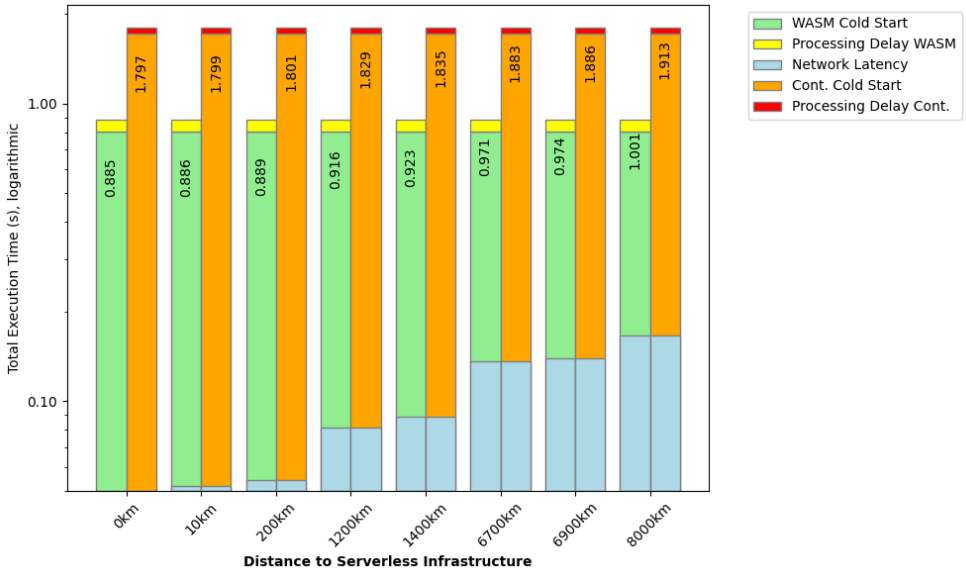

Fig. 5 Average network latency between the cluster locations

To overcome the problems currently associated with vendor-specific edge-cloud continuums, we have proposed a multi-layer approach where a new federation layer is defined with the goal of unifying different continuums offered by third-party infrastructure providers. Federating different infrastructures and providing a single interface for interacting with them would provide better user experience, reduce vendor lock-in, optimise costs, and offer better performance.

By deriving a set of 7 functional requirements, based on analysing state-of-the-art literature and past experience, we have described the necessary aspects which need to be taken into account when federating multiple edge-cloud continuums in practice. The proposed architecture meeting these requirements is based on 4 foundational pillars: Networking, Orchestration, Clustering, and Runtimes. Leveraging recent breakthroughs in related fields such as serverless runtimes, orchestration engines, and secure network connectivity options, we have described a concrete use-case implementation of a federated serverless infrastructure spanning the edge-cloud continuum, conforming to the set out requirements. The architecture has been verified in practice using a deployment of 6 federated clusters, across 3 countries and 2 continents, validating its potential in terms of closing the previously discussed open questions.

In terms of constraints of the proposed solution, we have approached the federation aspect purely from a scientific point of view, aiming to test its technical feasibility using state-of-the-art research and open-source technologies. As such, its commercial feasibility has not been take into account. In this context, additional incentive models can be developed in the future with the aim of enticing infrastructure operators or public institutions to become part of existing federations and contribute compute

resources. This can also lead to the establishment of thematic federations targeted at a specific community or workloads.

As the number of distinct sites in the federation grows, so will the associated network traffic costs. One option to overcome this challenge is to leverage the existing scheduling and workload migration algorithms, albeit in a federated context, and optimize for reduced bandwidth usage between locations where network traffic is prohibitively expensive in addition to optimizing energy usage and compute performance.

Finally, the proposed architecture rests on a a central management component – the central Kubernetes cluster which coordinates the complete federation. While this is not a problem from a technical perspective as it can be made redundant and even deployed in a wide topology as discussed previously, it can pose challenges from a management perspective if it is operated by a single entity. Robust policies would need to be in place preventing abuse or overreach.

The proposed architecture is modular and can act as a starting point for future extension for context specific scenarios. The layered design ensures that changes made to a given layer will not impact the upper layers, as long as the same interface for communication between them is maintained. This provides the opportunity to introduce more efficient scheduling algorithms in the future, new runtime environments, and new hardware for serverless functions execution, without impacting the top layer representing the user-deployed serverless functions.

Declarations

7.1 Ethics approval and consent to participate

Not applicable.

7.2 Consent for publication

Not applicable.

7.3 Availability of data and material

The accompanying material is publicly available under a permissive license on https://github.com/korvoj/serverless-federations.

7.4 Competing interests

The authors have no relevant financial or non-financial interests to disclose.

7.5 Acknowledgements

Not applicable.

References

[1] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J.E., Popa, R.A., Stoica, I., Patterson, D.A.: Cloud Programming Simplified: A Berkeley View on Serverless Computing. arXiv:1902.03383 [cs] (2019) arxiv:1902.03383 [cs]

[2] Kratzke, N.: A Brief History of Cloud Application Architectures. Applied Sciences **8**(8), 1368 (2018) https://doi.org/10.3390/app8081368

[3] Shafiei, H., Khonsari, A., Mousavi, P.: Serverless Computing: A Survey of Opportunities, Challenges and Applications (2019) https://doi.org/10.13140/RG.2.2.32882.25286

[4] Čolaković, A., Hadžialić, M.: Internet of Things (IoT): A review of enabling technologies, challenges, and open research issues. Computer Networks **144**, 17–39 (2018) https://doi.org/10.1016/j.comnet.2018.07.017

[5] Hellerstein, J.M., Faleiro, J., Gonzalez, J.E., Schleier-Smith, J., Sreekanti, V., Tumanov, A., Wu, C.: Serverless Computing: One Step Forward, Two Steps Back. arXiv:1812.03651 [cs] (2018) arXiv:1812.03651 [cs]

[6] Bittencourt, L., Immich, R., Sakellariou, R., Fonseca, N., Madeira, E., Curado, M., Villas, L., DaSilva, L., Lee, C., Rana, O.: The Internet of Things, Fog and Cloud continuum: Integration and challenges. Internet of Things **3**–**4**, 134–155 (2018) https://doi.org/10.1016/j.iot.2018.09.005

[7] Hazra, A., Rana, P., Adhikari, M., Amgoth, T.: Fog computing for next-generation Internet of Things: Fundamental, state-of-the-art and research challenges. Computer Science Review **48**, 100549 (2023) https://doi.org/10.1016/j.cosrev.2023.100549

[8] Kjorveziroski, V., Filiposka, S., Trajkovik, V.: IoT Serverless Computing at the Edge: A Systematic Mapping Review. Computers **10**(10), 130 (2021) https://doi.org/10.3390/computers10100130

[9] Intelligence at the IoT Edge — AWS IoT Greengrass — Amazon Web Services. https://aws.amazon.com/greengrass/ Accessed 2023-11-08

[10] IoT Edge — Cloud Intelligence — Microsoft Azure. https://azure.microsoft.com/en-us/products/iot-edge Accessed 2023-11-08

[11] Cloudflare Workers. https://workers.cloudflare.com/ Accessed 2023-11-08

[12] Ekwe-Ekwe, N., Amos, L.: The State of FaaS: An Analysis of Public Functions-as-a-Service Providers. In: 2024 IEEE 17th International Conference on Cloud Computing (CLOUD), pp. 430–438 (2024). https://doi.org/10.1109/CLOUD62652.2024.00055

[13] Donta, P.K., Murturi, I., Casamayor Pujol, V., Sedlak, B., Dustdar, S.: Exploring the Potential of Distributed Computing Continuum Systems. Computers **12**(10), 198 (2023) https://doi.org/10.3390/computers12100198

[14] Floerecke, S., Ertl, C., Herzfeldt, A.: Major drivers for the rising dominance of the hyperscalers in the infrastructure as a service market segment. International Journal of Cloud Computing **12**(1), 23–39 (2023) https://doi.org/10.1504/IJCC.2023.129772

[15] Saxena, D., Gupta, R., Singh, A.K.: A Survey and Comparative Study on Multi-Cloud Architectures: Emerging Issues And Challenges For Cloud Federation. arXiv (2021). https://doi.org/10.48550/arXiv.2108.12831

[16] Baarzi, A.F.: Multi-Cloud Serverless Deployment. Master Thesis. https://etda.libraries.psu.edu/catalog/18726azf82

[17] Baarzi, A.F., Kesidis, G., Joe-Wong, C., Shahrad, M.: On Merits and Viability of Multi-Cloud Serverless. In: Proceedings of the ACM Symposium on Cloud Computing. SoCC '21, pp. 600–608. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3472883.3487002

[18] Zhao, H., Benomar, Z., Pfandzelter, T., Georgantas, N.: Supporting Multi-Cloud in Serverless Computing. In: 2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC), pp. 285–290 (2022). https://doi.org/10.1109/UCC56403.2022.00051

[19] Smith, C.P., Jindal, A., Chadha, M., Gerndt, M., Benedict, S.: FaDO: FaaS Functions and Data Orchestrator for Multiple Serverless Edge-Cloud Clusters. In: 2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC), pp. 17–25 (2022). https://doi.org/10.1109/ICFEC54809.2022.00010

[20] Vasconcelos, A., Vieira, L., Batista, Í., Silva, R., Brasileiro, F.: DistributedFaaS: Execution of Containerized Serverless Applications in Multi-Cloud Infrastructures:. In: Proceedings of the 9th International Conference on Cloud Computing and Services Science, pp. 595–600. SCITEPRESS - Science and Technology Publications, Heraklion, Crete, Greece (2019). https://doi.org/10.5220/0007877005950600

[21] Faticanti, F., Santoro, D., Cretti, S., Siracusa, D.: An Application of Kubernetes Cluster Federation in Fog Computing. In: 2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), pp. 89–91 (2021). https://doi.org/10.1109/ICIN51074.2021.9385548

[22] Kubernetes Cluster Federation (Archived) (2023). https://github.com/kubernetes-retired/kubefed Accessed 2023-08-15

[23] E, B.-S.B., Devi, D.T.U.: Performance evaluation of Kubernetes cluster federation

using Kubefed. International Journal of Advance Research, Ideas and Innovations in Technology **9**(2), 217–223 (2023)

[24] Ungureanu, O.-M., Vlădeanu, C., Kooij, R.: Collaborative Cloud - Edge: A Declarative API orchestration model for the NextGen 5G Core. In: 2021 IEEE International Conference on Service-Oriented System Engineering (SOSE), pp. 124–133 (2021). https://doi.org/10.1109/SOSE52839.2021.00019

[25] Mfula, H., Ylä-Jääski, A., Nurminen, J.: Seamless kubernetes cluster management in multi-cloud and edge 5g applications. In: International Conference on High Performance Computing and Simulation (HPCS 2020) (2021). International Conference on High Performance Computing and Simulation, HPCS ; Conference date: 22-03-2021 Through 27-03-2021. http://hpcs2020.cisedu.info/

[26] Pandey, C., Tiwari, V., Rathore, R.S., Jhaveri, R.H., Roy, D.S., Selvarajan, S.: Resource-Efficient Synthetic Data Generation for Performance Evaluation in Mobile Edge Computing Over 5G Networks. IEEE Open Journal of the Communications Society **4**, 1866–1878 (2023) https://doi.org/10.1109/OJCOMS.2023.3306039

[27] Risco, S., Moltó, G., Naranjo, D.M., Blanquer, I.: Serverless Workflows for Containerised Applications in the Cloud Continuum. Journal of Grid Computing **19**(3), 30 (2021) https://doi.org/10.1007/s10723-021-09570-2

[28] Karthikeyan, R., Sundaravadivazhagan, B., Cyriac, R., Balachandran, P.K., Shitharth, S.: Preserving Resource Handiness and Exigency-Based Migration Algorithm (PRH-EM) for Energy Efficient Federated Cloud Management Systems. Mobile Information Systems **2023**(1), 7754765 (2023) https://doi.org/10.1155/2023/7754765

[29] He, T., Buyya, R.: A Taxonomy of Live Migration Management in Cloud Computing. ACM Computing Surveys **56**(3), 1–33 (2024) https://doi.org/10.1145/3615353

[30] Ahmad, I., AlFailakawi, M.G., AlMutawa, A., Alsalman, L.: Container scheduling techniques: A Survey and assessment. Journal of King Saud University - Computer and Information Sciences **34**(7), 3934–3947 (2022) https://doi.org/10.1016/j.jksuci.2021.03.002

[31] Oleghe, O.: Container Placement and Migration in Edge Computing: Concept and Scheduling Models. IEEE Access **9**, 68028–68043 (2021) https://doi.org/10.1109/ACCESS.2021.3077550

[32] Shitharth, S., Mohammed, G.B., Ramasamy, J., Srivel, R.: Intelligent Intrusion Detection Algorithm Based on Multi-Attack for Edge-Assisted Internet of Things. In: Srivastava, G., Ghosh, U., Lin, J.C.-W. (eds.) Security and Risk Analysis for Intelligent Edge Computing vol. 103, pp. 119–135. Springer, Cham (2023).

https://doi.org/10.1007/978-3-031-28150-1_6

[33] Xie, D., Hu, Y., Qin, L.: An Evaluation of Serverless Computing on X86 and ARM platforms: Performance and Design Implications. In: 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), pp. 313–321 (2021). https://doi.org/10.1109/CLOUD53861.2021.00045

[34] Lumpp, F., Barchi, F., Acquaviva, A., Bombieri, N.: On the Containerization and Orchestration of RISC-V architectures for Edge-Cloud computing. Proceedings of the 3rd Eclipse Security, AI, Architecture and Modelling Conference on Cloud to Edge Continuum, 21–28 (2023) https://doi.org/10.1145/3624486.3624490

[35] C, M., Y, M.S., S, S., Alhebaishi, N., Mosli, R.H., Alhelou, H.H.: Three-phase service level agreements and trust management model for monitoring and managing the services by trusted cloud broker. IET Communications **16**(19), 2309–2320 (2022) https://doi.org/10.1049/cmu2.12484

[36] Kjorveziroski, V., Bernard, C., Gilly, K., Filiposka, S.: Implementing Multi-Access Edge Computing with Kubernetes. In: 14th ICT Innovations Conference, pp. 137–150 (2022). https://proceedings.ictinnovations.org/2022/paper/569/implementing-multi-access-edge-computing-with-kubernetes

[37] Agarwal, S., Rodriguez, M.A., Buyya, R.: A Reinforcement Learning Approach to Reduce Serverless Function Cold Start Frequency. In: 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 797–803 (2021). https://doi.org/10.1109/CCGrid51090.2021.00097

[38] Kjorveziroski, V., Filiposka, S.: Kubernetes distributions for the edge: Serverless performance evaluation. The Journal of Supercomputing **78**(11), 13728–13755 (2022) https://doi.org/10.1007/s11227-022-04430-6

[39] K3s: Lightweight Kubernetes. https://k3s.io/ Accessed 2021-09-05

[40] MicroK8s - Zero-ops Kubernetes for Developers, Edge and IoT — MicroK8s. http://microk8s.io Accessed 2021-09-05

[41] Introduction - The Cluster API Book. https://cluster-api.sigs.k8s.io/ Accessed 2023-08-15

[42] IMT Vision – Framework and Overall Objectives of the Future Development of IMT for 2020 and Beyond. https://www.itu.int/dms_pubrec/itu-r/rec/m/R-REC-M.2083-0-201509-I!!PDF-E.pdf Accessed 2023-08-15

[43] Djemame, K., Parker, M., Datsev, D.: Open-source Serverless Architectures: An Evaluation of Apache OpenWhisk. In: 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC), pp. 329–335 (2020). https://doi.org/10.1109/UCC48980.2020.00052

[44] Li, J., Kulkarni, S.G., Ramakrishnan, K.K., Li, D.: Analyzing Open-Source Serverless Platforms: Characteristics and Performance. arXiv:2106.03601 [cs], 15–20 (2021) https://doi.org/10.18293/SEKE2021-129 arxiv:2106.03601 [cs]

[45] Kjorveziroski, V., Filiposka, S., Mishev, A.: Evaluating WebAssembly for Orchestrated Deployment of Serverless Functions. In: 2022 30th Telecommunications Forum (TELFOR), pp. 1–4 (2022). https://doi.org/10.1109/TELFOR56187.2022.9983733

[46] Kakati, S., Brorsson, M.: WebAssembly Beyond the Web: A Review for the Edge-Cloud Continuum. In: 2023 3rd International Conference on Intelligent Technologies (CONIT), pp. 1–8 (2023). https://doi.org/10.1109/CONIT59222.2023.10205816

[47] Long, J., Tai, H.-Y., Hsieh, S.-T., Yuan, M.J.: A lightweight design for serverless Function-as-a-Service. IEEE Software **38**(1), 75–80 (2021) https://doi.org/10.1109/MS.2020.3028991 arxiv:2010.07115 [cs]

[48] Spin Language Support Overview (2022). https://developer.fermyon.com Accessed 2023-08-15

[49] Introducing Spin. https://spin.fermyon.dev Accessed 2022-08-29

[50] Kjorveziroski, V., Filiposka, S.: WebAssembly Orchestration in the Context of Serverless Computing. Journal of Network and Systems Management **31**(3), 62 (2023) https://doi.org/10.1007/s10922-023-09753-0

[51] Containerd Wasm Shims. Deis Labs (2023). https://github.com/deislabs/containerd-wasm-shims Accessed 2023-08-15

[52] Building Spin Components in Other Languages (2022). https://developer.fermyon.com Accessed 2023-08-15

[53] WebAssembly Proposals. WebAssembly (2023). https://github.com/WebAssembly/proposals Accessed 2023-06-13

[54] Ménétrey, J., Pasin, M., Felber, P., Schiavoni, V.: WebAssembly as a Common Layer for the Cloud-edge Continuum. In: Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge, pp. 3–8 (2022). https://doi.org/10.1145/3526059.3533618

[55] KubevIRT - Building a Virtualization API for Kubernetes (18:28:23 +0000). https://kubevirt.io// Accessed 2022-05-20

[56] Nencioni, G., Garroppo, R.G., Olimid, R.F.: 5G Multi-access Edge Computing: Security, Dependability, and Performance. arXiv:2107.13374 [cs] (2021) arxiv:2107.13374 [cs]

[57] Kata Containers - Open Source Container Runtime Software. https://katacontainers.io/ Accessed 2023-08-15

[58] Kjorveziroski, V., Filiposka, S.: WebAssembly as an Enabler for Next Generation Serverless Computing. Journal of Grid Computing **21**(3), 34 (2023) https://doi.org/10.1007/s10723-023-09669-8

[59] Open, Multi-Cloud, Multi-Cluster Kubernetes Orchestration — Karmada. https://karmada.io/ Accessed 2023-08-15

[60] Resource Propagating — Karmada (2023). https://karmada.io/docs/userguide/scheduling/resource-propagating/ Accessed 2023-08-15

[61] Kumar, R., Trivedi, M.C.: Networking Analysis and Performance Comparison of Kubernetes CNI Plugins. In: Bhatia, S.K., Tiwari, S., Ruidan, S., Trivedi, M.C., Mishra, K.K. (eds.) Advances in Computer, Communication and Computational Sciences. Advances in Intelligent Systems and Computing, pp. 99–109. Springer, Singapore (2021). https://doi.org/10.1007/978-981-15-4409-5_9

[62] Overlay Networking — Calico Documentation. https://docs.tigera.io/calico/latest/networking/configuring/vxlan-ipip Accessed 2023-08-15

[63] Configure BGP Peering — Calico Documentation. https://docs.tigera.io/calico/latest/networking/configuring/bgp Accessed 2023-08-15

[64] Kjorveziroski, V., Canto, C.B., Gilly, K., Filiposka, S.: Full-Mesh VPN Performance Evaluation for a Secure Edge-Cloud Continuum. Submitted (2023)

[65] Juanfont/Headscale: An Open Source, Self-Hosted Implementation of the Tailscale Control Server. https://github.com/juanfont/headscale Accessed 2023-06-25

[66] Tailscale - Secure Remote Access to Shared Resources. https://tailscale.com/ Accessed 2023-06-25

[67] ZeroTier — Global Area Networking. https://www.zerotier.com/ Accessed 2023-06-25

[68] NetBird - Zero Configuration VPN for Fast-Moving Teams. https://netbird.io/ Accessed 2023-06-25

[69] Gravitl/Netmaker: Netmaker Makes Networks with WireGuard. Netmaker Automates Fast, Secure, and Distributed Virtual Networks. https://github.com/gravitl/netmaker Accessed 2023-06-25

[70] Tailscale: DERP Servers (2022). https://tailscale.com/kb/1232/derp-servers/ Accessed 2023-06-25

[71] Suzuki, M., Miyasaka, T., Purkayastha, D., Fang, Y., Huang, Q., Zhu, J.: Enhanced DNS Support towards Distributed MEC Environment. https://www.etsi.org/images/files/ETSIWhitePapers/etsi-wp39-Enhanced-DNS-Support-towards-Distributed-MEC-Environment.pdf Accessed 2023-06-13

[72] Yang, S., Xu, K., Cui, L., Ming, Z., Chen, Z., Ming, Z.: EBI-PAI: Towards An Efficient Edge-Based IoT Platform for Artificial Intelligence. IEEE Internet of Things Journal, 1–1 (2020) https://doi.org/10.1109/JIOT.2020.3019008

[73] Rizvi, A.S.M., Bertholdo, L., Ceron, J., Heidemann, J.: Anycast Agility: Network Playbooks to Fight DDoS. In: 31st USENIX Security Symposium (USENIX Security 22), pp. 4201–4218 (2022)

[74] Jangda, A., Powers, B., Berger, E., Guha, A.: Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code (2019). https://doi.org/10.5555/3358807.3358817

[75] Hockley, D., Williamson, C.: Benchmarking Runtime Scripting Performance in Wasmer. In: Companion of the 2022 ACM/SPEC International Conference on Performance Engineering. ICPE '22, pp. 97–104. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3491204.3527477

[76] Wang, W.: How Far We've Come – A Characterization Study of Standalone WebAssembly Runtimes. In: 2022 IEEE International Symposium on Workload Characterization (IISWC), pp. 228–241 (2022). https://doi.org/10.1109/IISWC55918.2022.00028