The continuum of computing: a pervasive service-oriented architecture

G. J. Hu[a], T. Vardanega[a]

[a]*Department of Mathematics, University of Padova, Italy*

Abstract

Abstract

Keywords: Continuum of Computing, Edge Computing, Cloud Computing, Webassembly

Email addresses: jiayi.ghu@gmail.com (G. J. Hu), tullio.vardanega@unipd.it (T. Vardanega)

1. Introduction

The Internet has evolved enormously since its inception. From just a simple communication layer for information sharing between researchers, it has grown into an ubiquitous platform for every user and any use. A flurry of organic changes to its infrastructure and interfaces has driven this vast transformation.

In a little over a decade of existence over the Internet, the Cloud [1] has earned users gigantic benefits by rendering virtually unlimited quantities of computing resources available in an affordable, fit-for-purpose, and rapidly scalable manner. The massive success of the Cloud, however, has also highlighted important deficiencies in the nature of its architecture: the huge energy footprint of its (very large) data centers; the intrinsic vulnerability to single-point failure of its centralized model; the inevitable latency incurred by storing and processing in the Cloud all the data ingested at the Edge; the threats to data security and privacy caused by the transfer of sensitive data.

Vision. The dramatic improvement to mobile connectivity, for ubiquity, reliability, and affordability, has allowed anyone to access the Internet from anywhere and at anytime. The massive boost and evolution of mobile computing, which has led to the emergence of richer client-side web apps, is expected to grow further with the uptake of 5G connectivity. Commercial forecasts predict that by the end of 2026, over 3.5 billion people, 45/% per cent of the world population, will have a 5G coverage subscription [2], with everyday objects connected to the Internet and to each other. The consequent impetuous growth of the Internet of Things, with the number of connected devices predicted to grow exponentially in the coming years [3], makes Cloud [1] centralization increasingly less practical. A more decentralised solution is required, instead, – the Continuum – where data processing may take place where it is deemed most convenient under any of the criteria enumerated above.

In this arrangement, a multitude of heterogeneous computing nodes, ranging from consumer devices like mobile phones and wearables to industrial sensors and actuators [4], positioned at the outer edge of the Internet network, allow the traditional Internet and the Internet of Things to integrate into a seamless Continuum, where a multitude of as-a-service applications may be developed, deployed, and employed regardless of location [5].

The Cloud can benefit from forming the Continuum together with the Fog and Edge, by allowing access to the physical world to occur in a more distributed and dynamic manner, and by favouring the creation of numerous novel latency-free, private and secure, energy-savvy services. Admittedly, however, scaling Fog and Edge platform infrastructures in the manner of the Cloud is made complex by the heterogeneity of the nodes, the fragmentation of the resources, the looming instability of wireless network, and the difficulty to predict capacity needs in advance [6].

This vision of seamless integration extends the view put forward by [7], which regards the Cloud and the IoT as distinct spaces, with the latter sending data and offloading computation to the former but not vice versa. The Continuum

concept goes beyond merely connecting network nodes to allow computation to happen at predetermined locations in the computing space.

Similarly to [8], [9], and [5], the Continuum of Computing as we understand it in this paper aggregates distributed services and deploys them from across the Edge, close to data sources, through to the center of the Cloud, along the path that best serves the user need for location, response latency, and resources. Depending on the use case and service level requirements, user applications may require processing and storage at the Edge, in the Cloud, or somewhere in between.

Numerous use cases of such vision can be associated to a variety of application areas, from managing extreme events (e.g. environmental monitoring [10]) to optimising everyday processes (e.g. manufacturing [4]) and improving life quality (e.g. healthcare [11] and smart cities [12]). Figure 1 attempts to capture said vision pictorially. Enacting this vision requires that sufficient com-

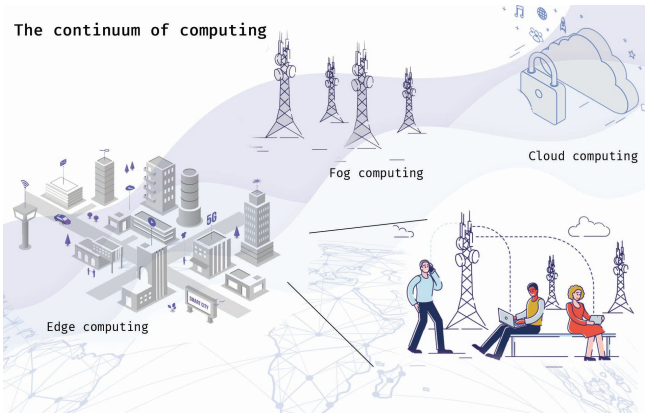

Figure 1: Pictorial view of the Continuum of Computing.

pute capabilities are deployed at the edge of the network, where physical reality begins and connected edge devices operate as the bridge between the physical and the digital worlds.

The foundation of the Continuum is made up of pervasive service platforms located anywhere the user is and a multitude of services, with different granularity, available over the Internet and composed opportunistically according to user needs. The Continuum must facilitate the elastic provisioning of the end-to-end service delivery infrastructures virtualised to allow scaling as a function of user demand and service requirements.

With [8], [9], and [5], we envision the Continuum to behave as a fluid, which continuously adapts its shapes to fit the environment where computation may freely flow *beyond* merely connecting network nodes to allow computation to happen at predetermined locations in the computing space.

Contribution. No matter how attractive this vision may be, however, very little efforts have been carried out to date, to the best of the authors' knowledge, to

explore the development of suitable enabling platforms. The work discussed in this paper aims to fill this gap.

Our research has been driven by a parallel effort in picking and evaluating the state of the technologies to assess the distance between our vision and the current state of the industry. We developed a Proof of Concept (POC) implementation of the Continuum infrastructure, combining mature Cloud tools like the Kubernetes [13] orchestrator with innovative open-source solutions, in order to address many of the challenges in integrating the Cloud into a Continuum with the Edge. We describe such challenges thoroughly in Section 2.

Following, we propose in Section 3 a reference high-level architecture for the Continuum and proceed to implement a POC of the foundation of the architecture. Notably, we the explored discoverability of edge nodes in a Kubernetes cluster, interoperability of ubiquitous services on the web and deeply experimented with the nascent sandboxing standard WebAssembly [14] to bring container-like virtualization and portability on both relatevily powerful machines and constrained devices. In Section 5 we present the comprehensive technology selection behind these features. For each of them, we describe its role with regards to the goals of the continuum and expose its the merits and the shortcomings.

In our findings, we noticed that many of today's tools tend to fit well in principle in the Continuum vision, a sign of convergence in the trends of many areas of software development like Cloud, Edge and Web. There is still however a substantial gap in terms of viability. On one side, existing production-grade tools like Kubernetes are still figuring out how to align their interface to modern use cases like Edge computing. On the other side, nascent solutions like WebAssembly are still in an early stage of thinking and are not able to provide the maturity requested by the industry to bring them to production use cases. Good illustrations of such point are the notably worser performance of interpreted WebAssembly in constrained devices and the lack of a standard networking WebAssembly interface, hindering the practicality of any high-level service other than pure computation machine learning. These and other conclusions are outlined in Section 6.

2. The challenges ahead

We acknowledge that several challenges lay ahead in the realization of the Continuum as we have described it. Besides featuring extreme heterogeneity, in fact, current Edge technology most notably lacks support for service orientation, interoperability, orchestration, reliability, efficiency, availability, and security. While not exhaustive, we deem this list of required features a decisive enabling factor. To understand these needs and relate the experimental findings discussed in Section 5 to them, we now briefly discuss each such requirement in isolation.

Context-sensitive service orientation. We deem service orientation the most appropriate style to organise and utilise distributed capabilities that may lie under the control of different ownership domains.

A service-oriented model is centered around a service provider that publishes its service interface (i.e., how users may access the corresponding functionalities) via a service registry where consumers may locate it and use it to bind to the desired service provider [15].

The prime virtue of such a model is the loose coupling it earns for services, which are solely responsible for the logic and information that they encapsulate, agnostic of the composition in which they can be aggregated by higher-level providers, and placed behind well-defined interfaces and service contracts with corresponding constraints and policies. This design is in stark contrast with the dominant practice of present day, where a multitude of ad-hoc programs are developed that are confined to single places of the network, and permanently fix the behavior of the associated devices [5].

However, major limitations have to be overcome before services can be operated seamlessly and maintained agilely.

First and foremost, the lack of vendor-neutral, trustworthy and widely accepted service intermediaries, to enable efficient retrieval of services that meet given user needs and warrant agreed levels of quality. Unfortunately, to date, interoperability is not dear to the main actors in the field [16].

Secondly, the lack of inter-operable support for composing higher-order services from lower-level ones. Individual providers adopt their own conventions for interfaces and communication protocols: for example, Google Cloud Platform services heavily use Protocol Buffers [17], a Google technology for serialising structured data, in their service APIs. A plausible implementation of the Continuum should map high-level descriptions (e.g., flexible key-value stores) to vendor-specific implementations.

Moreover, whereas services on the Internet of today are mute and unresponsive, future services should be communicative and reactive to their respective environments [15]. The current service interfaces in fact are ostensibly designed with human interaction in mind, thus being scarcely suited for machine-to-machine (M2M) discovery and interaction.

In our vision of the Continuum, binding a consumer to a particular service interface should entail minimal direct interaction with the provider's infrastructure: the provider should have complete control of the service, relieving the customer from any associated cost of ownership.

Finally, services fit for the Continuum, hence deployable at the Edge, are sensitive to the context of the environment in which they operate. The context awareness we envision is necessary to implement local control loops and trigger specific actions on local events (e.g. sensor readings in our later case study).

Orchestration. The transition to the Continuum will require coordinating and scheduling the operation of myriads of distributed service components. The complexity of that endeavor makes orchestration essential, over and above the rating it enjoys from DevOps adopters [6]. Granted, orchestrating in the Continuum is especially challenging owing to the scale, heterogeneity and diversity of resource types, and the uncertainties of the underlying environments for resource capacity (e.g. bandwidth and memory), network failures, user access

pattern (e.g., for quantity and location), and service life cycle. Extreme heterogeneity also hinders devising sound pricing models that reflect account locations, resource types, transport volumes, and service latency.

Orchestrating services in the Continuum is a remarkable challenge, which encompasses technologies from a variety of fields, including wireless cellular networks, distributed systems, virtualization, platform management, and also requires mobility handover and service migration at local and global scales.

Virtualisation. The rapid pace of innovation in data centers and application platforms has transformed the way organizations build, deploy, and manage services. Container-based virtualization, owing to its natural versatility and light unitary weight, has become the dominant solution for all seekers of elastic scalability. Thousands of containers can be stored on a physical cloud host in contrast with just very few traditional heavy-weight Virtual Machines. A natural near-future direction is a Edge-friendly containerization that allows users to deploy services and applications on heterogeneous Edge nodes with minimal effort. Several works (e.g. [18] and [19]) argue the feasibility of container virtualisation applied to cheap low-powered devices, such as the Raspberry Pi [20].

Thanks to the underlying Docker image technology [21], containers provide resource isolation, self-contained packaging, anywhere-deployment, and ease of orchestration, very fitting features for the Continuum. Several Cloud providers use this technology for their Platform-as-a-Service and Function-as-a-Service solutions. Modern serverless [22] platforms (e.g., Google Cloud Functions, Azure Functions, AWS Lambda) isolate functional units in ephemeral, stateless containers.

However, we reckon that the current state of containerization technology still comes at a too great expense in terms of memory overhead and system requirements. A typical state-of-the-art Edge runtime for containers requires at least half Gigabyte of memory even for staying idle, as shown in our evaluation in Table 5.3. Besides, containers incurs latency between hundreds of milliseconds and seconds [23], wholly unaffordable for latency-sensitive services that operate at the Edge. To achieve better efficiency, some platforms cache and reuse containers across multiple function calls within given time windows, typically 5 minutes. In the Edge, however, long-lived and over-provisioned containers can quickly exhaust local resource capacity, and become impractical for serving multiple IoT devices. Supporting a large number of serverless functions while warranting low response time, within tens of milliseconds [24], thus is one of the main performance challenges for resource-constrained Edge nodes.

In the way of hard security, containers also offer weak isolation. To achieve stronger guarantees, they are often run in per-tenant VMs, too heavy for Edge or Fog nodes like the Raspberry Pi. A lightweight yet robust isolation solution thus is another hot research question in the quest for the Continuum.

Dynamic configuration. Edge and IoT nodes must be capable of prompt reaction to context changes in the environment where they operate. Such reactions are critical to applications like video analysis [25] that are natural candidates

for deployment at the Edge. The risk scenario to be avoided is that IoT devices continue to operate needlessly or erroneously because their controllers running on nearby Edge nodes are late in making opportune adjustments.

Enabling dynamic configuration on constrained devices would allow swift adaptation to environmental events in accord with application requirements. This goal can be achieved by running an application-specific computation on the node itself, earning a considerable improvement in task accuracy (owing to physical vicinity), network bandwidth, and response time.

Opening Edge devices to arbitrary code execution, though, exposes the system to malicious acts, with compromising breaches that can exploit the slightest code weakness. Current software isolation stacks like containers can hardly be used in trustworthy embedded systems as the latter typically lack storage capacity or OS components.

A further challenge of dynamic configuration is to strike an acceptable compromise between warranting isolated execution and containing the corresponding loss of efficiency and increase in energy consumption. A common memory-safe execution technique is to adopt interpreted languages that provide type and memory safety. For instance, the authors of [10] have ported interpreters of high-level languages (Lua and Python) to C to support dynamic reconfiguration of the internal logic in telemetry sensors.

Interoperability. Many technologies are available for connecting and integrating all kinds of "things" into the Continuum. ZigBee, IPv6 over Low-Power Wireless Area Networks (6LoWPAN), MQTT, and CoAP [26]. are popular in the wireless sensor networking area, while OPC [27] has a good take-up in factory automation. The fact is, though, that such technologies are too numerous and varied for any single standard to be able to accommodate all of them.

For this reason, building the Edge infrastructure of the Continuum requires coping with extreme heterogeneity, which standards will hardly be able to tame. Best is to separate functionality from implementation, thus seeking interoperability in lieu of standardization. Service-oriented architectures are ideal in this regard as they encapsulate functionality in services that can expose a common interface, abstracting away inner idiosyncrasies.

Having an infrastructure that allows connecting and integrating a diverse set of technologies is not just a "necessary evil" but rather a strength that earns two key benefits. Firstly, it allows applying different solutions to different applications, in a best-fit logic. Secondly, an infrastructure where diverse technologies can easily be integrated into will be more future-resistant. This is particularly important for the Edge and IoT, which will certainly see new developments for technologies and protocols. An infrastructure built with technology diversity in mind will allow interoperability with existing and already deployed devices and networks.

Portability and Programmability. In Cloud-native models of computing, users of containerization are free to elect the programming language of choice, with the sole concern to ensure that the corresponding executable image, which embeds

all the necessary package libraries and configurations, can be deployed on the target platform. Such images can be constructed from minimal file system layers, sharing read-only parts of them (e.g. base OS) with other containers, thus shedding considerable footprint.

Conversely, in the Continuum, the compute nodes are vastly diverse for CPU (e.g., x86_64, ARM32, ARM64, and RISC-V) and runtime, which makes it much harder for programmers to make application development and deployment decently portable.

Docker images attempt to overcome this challenge by defining multiple variants (usually referred to as tags) of the same image, to target multiple architectures, for processor or OS. This nice feature, however, does not alleviate the pain of configuring and building each application image for each target platform. Moreover, the lack of general-purpose OSs embeddable on Edge devices or their limitation in resource capacity prevent from using conventional containers on them, further impairing portability across the Continuum.

Portability also relates to programmability, in that the choice of programming language may favor or hinder portability.

The Serverless paradigm fits this bill well on two grounds [28]. First, the serverless programming model makes developing, deploying, and managing applications dramatically less burdensome than in conventional styles. Second, individual functions may flexibly and equally run on the Edge or the Cloud alike, thus earning much in the way of portability.

However, while well suited for event-driven and request-reply applications, the serverless computing model falls short for long-running services that must feature high availability and low latency, as needed for Edge-based user interaction or industrial control loops.

Mobility. In the Continuum, services should be relocated following the user's movements, and so should the corresponding data and state. It therefore follows that all synchronization, data & state migration, handshakes and collaboration needs have to be addressed across multiple layers of the compute and communication infrastructure. When mobility is involved, provisioning data and services ought to be highly reactive and highly reliable, which is a challenge. For example, present-day vehicular networking and communication reportedly appear to be intermittent or unreliable [29].

Security and Privacy. The seamless integration of Edge, Fog, and Cloud computing is bound to raise unforeseen security issues. Previously unexplored scenarios, such as the interplay of heterogeneous Edge nodes and the migration of services across global and local scales, create the potential for original channels of malicious behaviour [30].

In point of principle, end-user data originated at the Edge should be stored locally, with the user able to control whether and which service providers be allowed to access them. As long as Edge nodes expose vulnerabilities (e.g. tampering, spoofing, falsifying), however, storing and processing privacy-sensitive

data there should be regarded as far more hazardous than retreating them at the center of the Cloud.

3. System design

Highly distributed networks are the most effective architecture for the Continuum, particularly as services become more complex and more bandwidth-hungry. Although often perceived as a single entity, the Internet is actually composed of a variety of different networks. The net result of such articulation is that content generated at the Edge may have to traverse multiple networks, crossing peering points before reaching its destination data center, at the center of the Cloud.

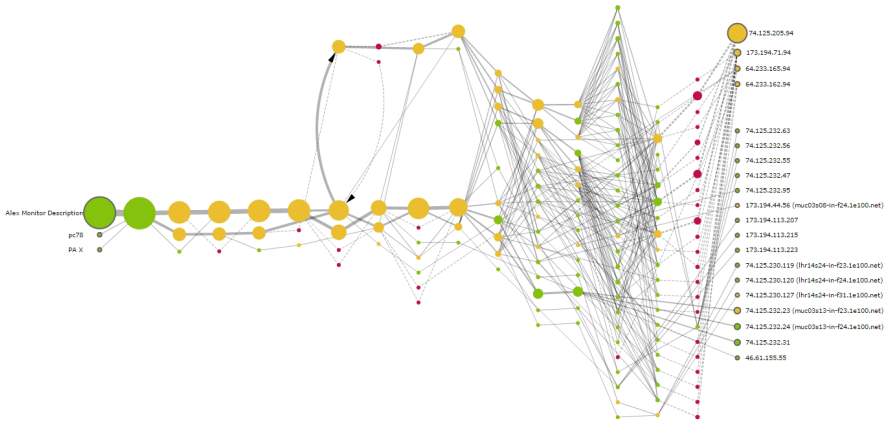

Figure 2: Traceroute virtualisation of an IP packet reaching google.com. The left green nodes are the source nodes, while packets travel across to the extreme right to servers located in data centers. Source: [31].

The peering points, depicted as nodes in Figure 2, where networks exchange traffic, are the common bottleneck of the Internet. Capacity at these points typically lags behind the reliability demand mainly due to the economic structure of the Internet [6]. The economic incentive ramps at the first (cloud data centres) and the last mile (IoT) of the path, with very little interest to invest in peering points, which become the cause for packet loss and increased latency.

For the Continuum, the throughput of the entire communication path, from IoT devices to data centres back to end users, is a paramount concern. Such realization suggests preferring processing at the Edge than causing network pressure. Offloading some compute tasks from IoT sensor or actuator nodes to the Edge is likely to be more energy efficient. This strategy may not always be as convenient, though. Long computations, as in big-data analysis for instance, in fact, are best offloaded to a more capable but distant node, possibly up to

the center of the Cloud. Response time is the sum of two components: the compute latency and the transmission latency. High compute latency can outweigh transmission efficiency. Hence, Edge computing has the responsibility to determine the preferable trade-off between the two, leveraging resources across the whole Continuum to achieve the best optimisation on a case-by-case basis.

Determining the best location for the computation to happen dynamically, requires seamless movement of data and computation. Such Continuum is not one form of computing (e.g. edge computing) supplanting another (e.g. cloud computing), but an evolution that harnesses the entire computing space as a whole.

3.1. Cluster federation

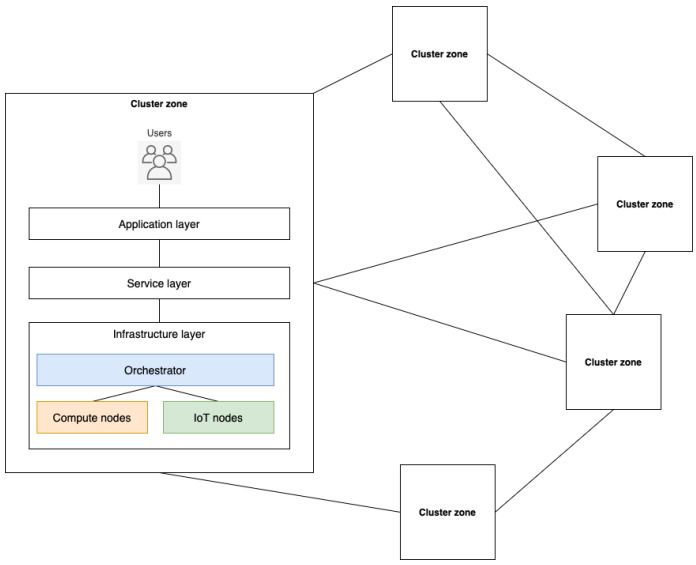

Figure 3: A high-level view of a federated set of cluster nodes.

To accomplish such dynamism of computation, we present in Figure 3 the basic building block of the system, the *cluster node*, which is amenable to a flexible, agile, and geographically bound aggregate called *cluster zone*. Each such zone federates the resources collectively available within its nodes, and orchestrates their deployment.

The federation is achieved via a dedicated *infrastructure layer*, which discovers and aggregates services, data and compute resources transparently across cluster nodes in a manner that meets end-to-end QoS requirements.

The system as we envisage it, dynamically instantiates and schedules services along the path from source to destination, based on application-specific requirements and constraints. In the event that a single cluster zone lacks hardware, software or data resources to meet the user needs, it would propagate

the corresponding requests outside of its federation to cluster zones within an acceptable geographical distance that have the required capabilities.

Collaboration among cluster zones is essential to support user mobility across neighboring regions. In the Continuum, services should follow the user movements without significant outage or perturbation.

User applications running on a single cluster node access resources thanks to the intermediation of the *service layer*. Applications intending to run on a cluster zone specify their service requirements and constraints, namely the type of resource (e.g., expected performance, pricing), without detailed knowledge of the underlying infrastructure. The *orchestrator* receives the requirements from the *service layer* intermediary, and provisions resources and services as required, assigning them to *compute nodes* in the target cluster zone. While geographically distant, such nodes form an interconnected cluster that logically aggregates the available resources.

Services capture common dependencies like a database and persistent storage for data sources, along with pertinent constraints on them, such as latency limits and subscription plans.

At this early stage and due to the broad scope of the topic, we leave the federation architecture as an open research question for the future of the Continuum. We focus our attention towards the infrastructure architecture, on which the federation layer is based upon.

3.2. Infrastructure architecture

The infrastructure layer is composed of a set of service providers that offer data and computational resources. The data can be generated by streaming IoT devices (e.g. cameras, smartwatches, and smart infrastructure). The computational resources can be heterogeneous and distributed through the infrastructure, from the Cloud to the Edge.

A reference architecture of the infrastructure is shown in Figure 4.

3.2.1. Orchestrator control plane

The orchestrator control plane is the core of the orchestration system. It has a resource monitor module responsible for keeping track of real-time resource consumption metrics for each node in the compute cluster. The scheduler usually accesses this information to make better optimisation decisions. The scheduler is responsible for determining whether there are enough resources and services available in the Continuum to execute the submitted application. If resources are insufficient, applications can be rejected or put on wait until the resources are freed. Another possible solution is be to increase the number of cluster nodes to place the incoming application. Such nodes can be provisioned from local machines or anywhere in the network, typically close to the cluster. After determining if requirements can be satisfied, the scheduler maps application components onto the cluster resources. This deployment is done by considering several factors, e.g. the availabilities, the utilization of the nodes, priorities, or constraints.

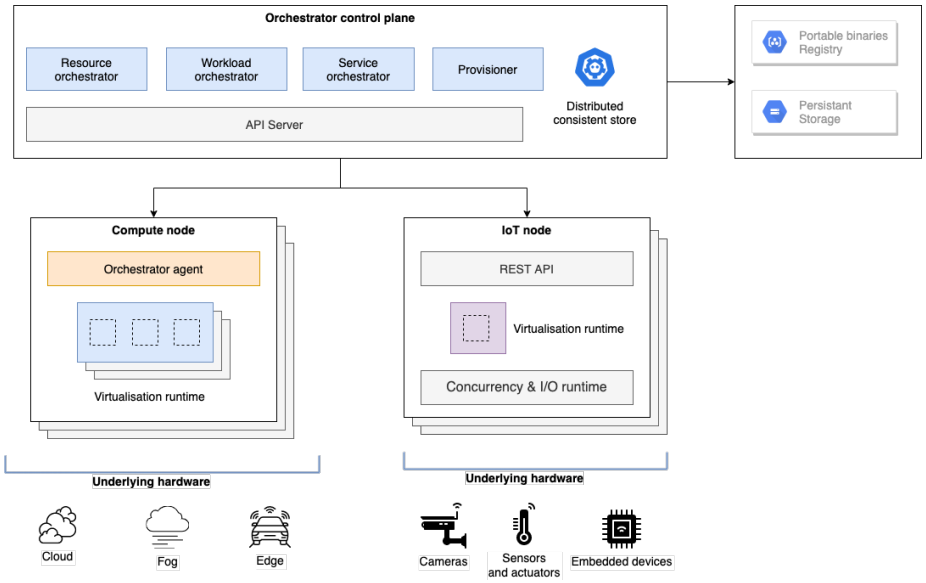

Figure 4: Reference architecture for the infrastructure.

3.2.2. Compute nodes

Each machine in the cluster that is available for services and applications is a compute node. Each of these nodes has an implementation of the orchestrator agent runtime with various responsibilities. First, it collects local information such as resource consumption metrics that can be periodically reported to the control plane. Second, it starts and stops service instances and manages local resources via a virtualisation runtime. Finally, it monitors the instances deployed on the node, sending periodic status to the control plane.

3.2.3. IoT nodes

IoT nodes are embedded devices that act as sensors and actuators, provided as services to the cluster. The IoT nodes are heterogeneous in runtime implementation and communication protocols. Therefore applications in the cluster interface with them via brokers provisioned by the cluster, as explained in Section §**??**. Besides, the embedded devices support dynamic configuration by running arbitrary virtualisation modules in a lightweight runtime, assuming the module size and the hardware requirements can be satisfied by the limited device.

3.2.4. Underlying infrastructure

One of the main requirements of the infrastructure architecture is the flexibility in being deployed on a multitude of platforms. Because of these, the cluster machines can be either VMs on public or private cloud infrastructures, physical machines on a cluster, or even mobile or edge devices, among others.

4. Weather-based services

As a practical example to guide the architecture's implementation, we apply the Continuum system design to a weather-based services. The emerging of efficient sensing methods and IoT technologies are giving the opportunity to record and analyze possible influences of weather factors in many areas like flood warning [10], electrical load forecasting [32] and precision agriculture [33].

For instance, for electrical load forecasting, weather relevant attributes are of great significance and include values like the temperature, air pressure, vapor pressure, precipitation, evaporation, wind speed, and sunshine duration. An interesting addition is the fact that detailed weather condition data can sometimes only be captured by household sensors, such as the indoor temperature, sunshine duration, and indoor air quality, which differentiate in every house but have a strong effect on the energy consumption. These data are also typically preprocessed to give out the maximum, minimum, and average values, and then normalized to generate the final inputs. This peculiar trait is a good showcase of when safe and performant arbitrary code execution is fundamental for many services in the Continuum.

Likewise, the general parameters considered in precision agriculture are soil moisture content, soil temperature, temperature of the surrounding, humidity level, CO2 level of air, and sunlight intensity level. The sharing of weather parameters between electrical load forecasting and precision agriculture is, in turn, an additional point in favour of sharing the data and computation of smart sensors on the Edge.

Lastly, in a flood warning system, the existence of local sensor-actuator networks can be leveraged to support timely disaster analysis. The telemetry stations acquire data (e.g. air humidity, soil moisture) from wireless sensors networks, process the data in a distributed manner, and locally determine potential levee breaking. On the other hand, the geographical distance between the networks, the volume of data, and the relatively low interest (when no significant event is happening) make a centralised vertical solution undesired. Thus, the distributed architecture of the Continuum is proposed as a viable architecture for advanced telemetry services with distributed intelligence.

To meet the requirements of the three types of services, we devise a system based on the architecture proposed in Figure 5:

- Sensor nodes: they are composed of sensor devices that collect data, preprocess it and transmit it to the edge cluster for further processing. One challenging task of this layer is implementing the dynamic configuration of the internal logic, as preprocessing is a necessary step presented in the case of electrical load forecasting;

- Broker nodes: they expose the sensor nodes behind a common interface. The broker subscribes to the IoT data and the device periodically sends updates, which are forwarded to the cluster. The broker ensures that both parts, IoT nodes and services nodes, are independent as far as they agree to communicate following the same API interface;

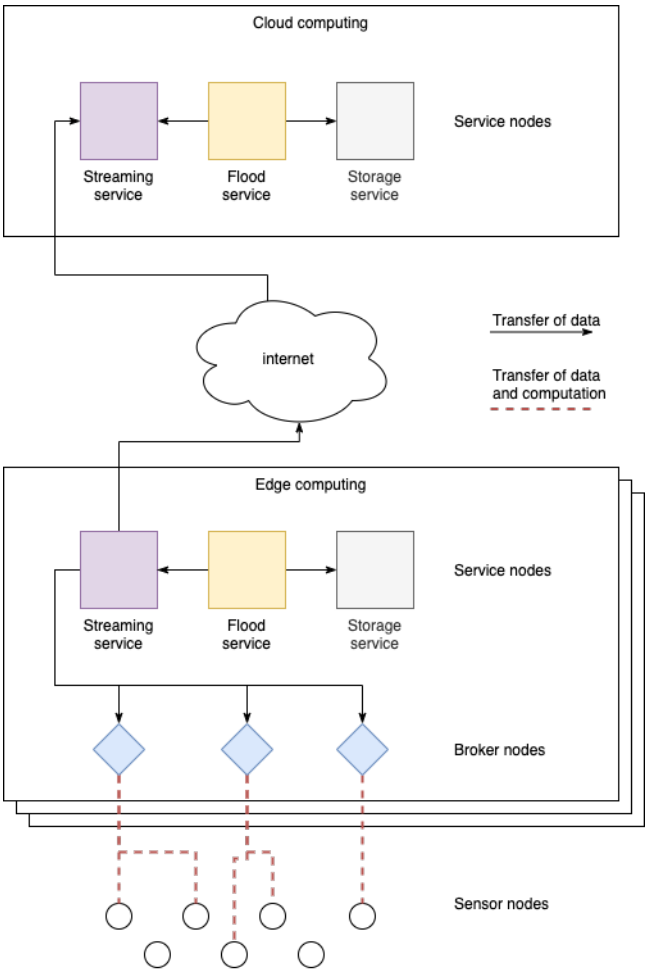

Figure 5: Architecture for the flood warning system.

- Service nodes: they implement the needed services and allow them to be reused across different clusters. Internally each service can be composed of stand-alone services. We show the example of levee monitoring, which needs a streaming service to aggregate the data from the brokers nodes, a local database to store the information for the analysis, and a flood prediction service to analyse the information and provide insight.

The service nodes are deployed at multiple edge clusters, corresponding to different stations, and at a cloud cluster. The rationale for expanding the services to the cloud is two-fold.

First, the edge clusters are heterogeneous in computing capacity, and some zones may have not enough computing power to handle streams. Leveraging

the cloud can help increase the workload at the cost of more bandwidth usage and latency. Such compromise might be acceptable, especially in the case of intensive data analysis on the sensor data.

Second, the prediction model could benefit from more knowledge derived from multiple streams geographically distributed. Likewise, a flood risk assessment model running in the cloud could achieve a globally optimal solution, whereas edge services can output only locally optimal results. On the other hand, during flood threat scenarios, the communication channels may become unavailable, so the system must perform a localised assessment. Unfortunately, the loss of communication is unpredictable, but the system must quickly adapt to the scenario. Such computing dynamism is a perfect scenario where the Continuum shines compared to relatively static cloud-only, edge-only, or pre-defined cloud+edge architectures.

5. Technology selection and evaluation

We proceed to provide the technology baseline to address a picked subset of the challenges presented in Section §2. For each of them, we propose a candidate technology and assess its maturity under two points: fitness with regards to the goals of the continuum and appropriate measurings related to the technology.

For the evaluations, we have used the following hardware and software:

- Edge cluster nodes: 4 Raspberry Pi 4 Model 3B+ with Quad-core Cortex-A53 (ARMv8) 64-bit SoC at 1.4GHz and 1 GB physical memory. The Raspberry 3B+ model has been chosen to showcase the feasibility of the presented technologies on limited low-powered machines, relatively cheap and with only 1GB of memory;
- Sensor nodes: a STM32F407 microcontroller with ARM Cortex-M4 core, 512KiB flash storage, and 128KiB of memory. The device is also capable of many 32-bit floating-point operations.

Raspberry Pi and STM32F407 microcontrollers are designed for moderately high computational performance, low unit cost, and power efficiency in edge computing environments. The author believes that this evaluation's results should generalize to other ARM machines and microcontrollers in the Cortex-M family.

5.1. Service orientation

The web has become the world's most successful vendor-independent application platform and the most dominant architectural style on it is Representational State Transfer (REST) [34] that makes information available as resources identified by URIs. The web is a loosely coupled architecture and applications communicate by exchanging representations of these resources using the HTTP protocol. HTTP is the most popular application protocol on the Internet and the pillar of web architecture. However, new communication protocols (e.g.

CoAP in Section **??**) are emerging to extend the web to the Internet of Things and HTTP itself is undergoing revisions (e.g. HTTP/3 or QUIC [35]).

Our rationale for picking REST is three-fold. First, REST resources are an information abstraction that allows servers to make any information or service available, identified via Uniform Resource Identifiers (URIs). As exampke, this allows sensor nodes to act as a server and to own the original state of a resource. The client negotiates and accesses a representation of it. Such representation negotiation is suitable for interoperability, caching, proxying, and redirecting requests and responses. These features enable seamless inter-operation and better availability of any kind of service in the Continuum, especially IoT-involved services. Besides, under the REST architecture, edge nodes can often advertise web links to other resources creating a distributed discoverable IoT web and resulting in an even more scalable and flexible architecture.

Second, REST allows the different parties to use a uniform interface: clients access the server-controlled resources in a request-response fashion using a small set of methods with different semantics (GET, PUT, POST, and DELETE). The requests are directed to resources using a generic interface with standard semantics that intermediaries can interpret. The result is an application that allows for layers of transformation and indirection independent of the information origin. We used these features again for bringing IoT nodes into the Continuum and to allow the coexistence of multiple equivalent services, offered however by different Cloud providers.

Lastly, REST enables high-level interoperability between RESTful protocols through proxies or, more generally, intermediaries that behave like a server to a client and play a client with respect to another server. REST intermediaries play well with the assumption that not every device must offer RESTful interfaces directly. This works well with the diversity of communication protocols on the Edge.

Open Service Broker. With such RESTful architecture in mind, we realized a web-based service platform by adopting the standard Open Service Broker (OSB) API [36]. Components that implement the OSB REST endpoints are referred to as service brokers and can be hosted anywhere the application platform can reach them. Service brokers offer a catalogue of services, payment plans and user-facing metadata. The main components of the OSB architecture are presented in Figure 6.

In the Continuum's service platform, providers control the access to the services and payment plans but allow developers to bring their own services to the catalogue. Over the years, we envision that a rich ecosystem of services will be developed and accessible via simple well-documented RESTful interfaces.

Contrary to this vision, however, cloud standards have failed to gain traction. Therefore, to bridge the heterogeneity gap between platforms we used brokers to orchestrate resources at different levels within a provider. As the number of cloud vendors is limited, building brokering layers that align access to different clouds was possible. The service broker translates RESTful requests from the platform to service-specific operations such as creating, updating, deleting, and

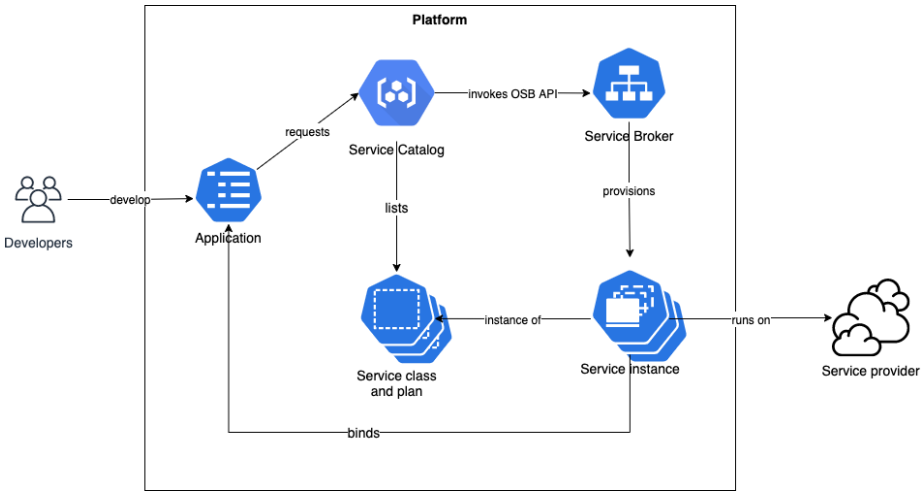

Figure 6: The Open Service Broker architecture.

generating credentials to access the provisioned services from applications. Service brokers can offer as many services and plans as desired. Multiple service brokers can be registered with the service platform so that the final catalogue of services is the aggregate of all services. The platform is thus able to provide a rich catalogue and a consistent developer experience for application developers consuming these services.

CoAP. To include the IoT nodes in the REST architecture, we adopted CoAP [37], a web communication protocol for use with constrained nodes and constrained (e.g. low-power, lossy) networks. A central element of CoAP's reduced complexity compared to HTTP is that it uses the UDP transport protocol instead of TCP and defines a very simple message layer for retransmitting lost packets.

The protocol is designed for M2M applications and provides a RESTful architecture between IoT nodes with support for built-in discovery of resources. As a result, CoAP easily interfaces with HTTP for integration with web services while meeting specialized IoT requirements such as multicast support, very low overhead and simplicity for constrained environments.

Another advantage of CoAP is that human interactions follow a familiar and intuitive pattern already used by many developers by using standard web technologies. Thus, the learning curve is smoother. This feature cannot be underestimated as allowing developers to use a familiar and seamless programming experience is essential to achieve Continuum's success.

We made CoAP nodes interoperabile with the rest of the Continuum by following the REST architecture's proxy pattern, as depicted in Figure 7. We built intermediaries, later presented in §5.2, that speak CoAP on one side and HTTP on the other without encoding specific application knowledge. Because

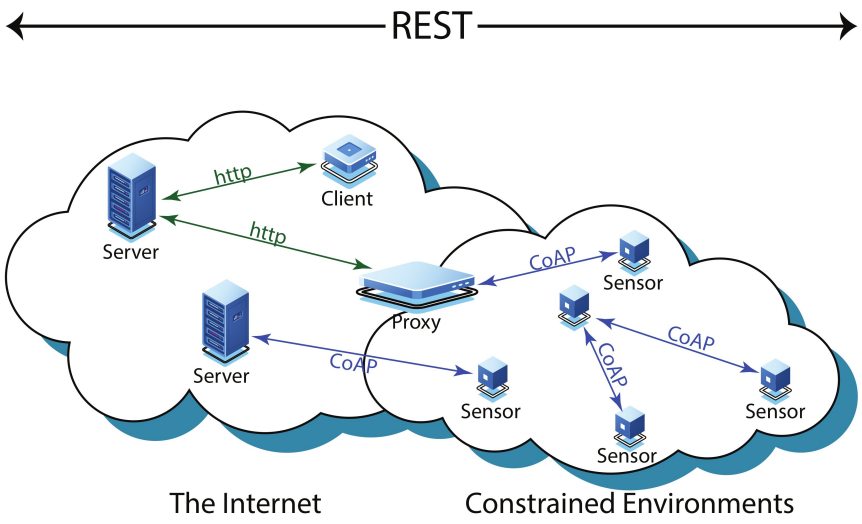

Figure 7: The REST architecture enhanced with CoAP. Source [37].

equivalent methods, response codes, and options are present in HTTP and CoAP protocols, the mapping between the two is straightforward. As a result, the intermediary can discover CoAP resources and make them available at regular HTTP URIs, enabling web services in the aforementioned service platform to access CoAP servers transparently.

5.2. Orchestration

Kubernetes [13] is an open-source orchestration framework designed to manage containerized workloads on clusters and originated from Google's experience with Cloud services. Two notable features makes Kubernetes appealing for our POC. First, it allows for various container runtimes from a technical perspective, with Docker natively supported by the platform. Thanks to the Container Runtime Interface (CRI) API standardisation, Kubernetes supports other container technologies such as containerd [38]. This extensibility allowed us to leverage a uniform virtualization platform between Cloud and Edge nodes, as later presented in 5.3.

Second, Kubernetes provides users with a wide range of options for managing their Pods and how they are scheduled, even allowing for pluggable customised schedulers to be easily integrated into the system. Notably, it also supports label-based constraints for the Pods' deployment. Developers can define their labels to specify identifying attributes of objects that are meaningful and relevant to them but that do not reflect the characteristics or semantics of the system directly. An example would allow specifying that an IoT device component must be reachable from the host. Nevertheless, more importantly, labels can be used to force the scheduler to colocate services that communicate a lot

into the same availability zone, improving the latency drastically and paving the way for context-aware services.

It is worth mentioning that we picked Kubernetes over Docker Swarm [39] because of the above points and due to the lack of multitenancy in the latter. Docker Swarm is a popular open-source orchestrator often cited for edge orchestration (e.g. [19], and [40]) due to its simplicity, but support for multitenancy is a must for our service platform.

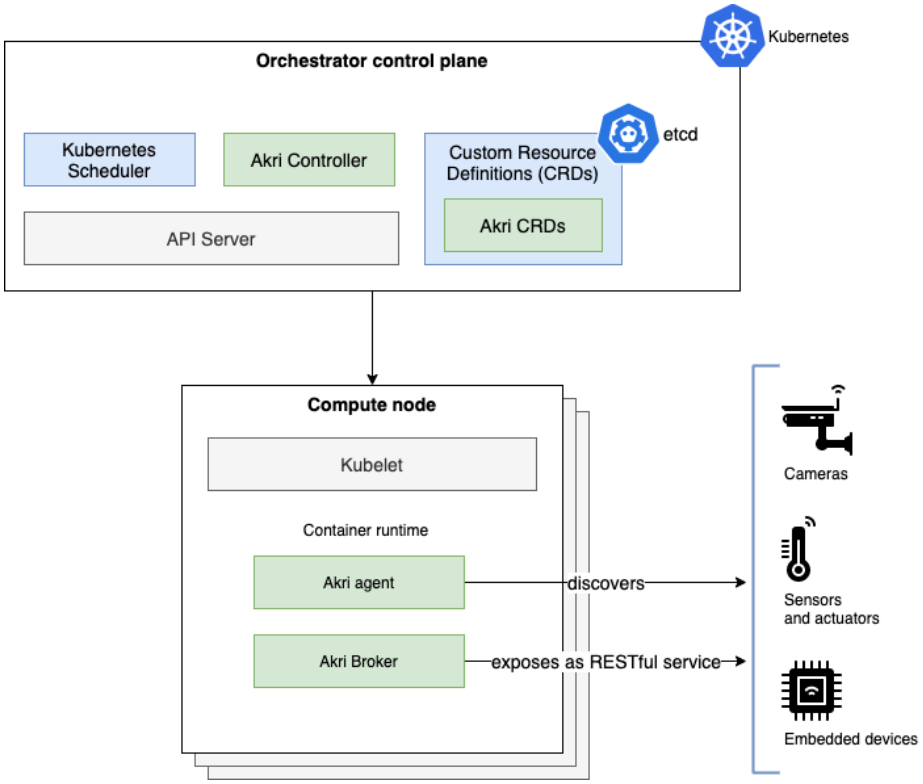

Figure 8: The Akri architecture.

Akri. To support IoT devices on the Kubernetes cluster, we adopted Akri [41], an open-source project which allows visibility to IoT devices from applications running within the Kubernetes cluster. Akri stretches Kubernetes' already experimental APIs to implement the discovery of IoT devices, with support for the diversity of communication protocols and ephemeral availability.

Akri's architecture, presented in Figure 8, can be divided into four main components: the agents, the controller, the brokers and the configuration. A configuration is a Kubernetes Custom Resource Definition (CRD) to extend the Kubernetes API with new types of resources. Specifically, a configuration

defines a communication protocol and the related metadata, such as the protocol discovery parameters or the Docker image to use as agent.

The Akri agent is a Pod responsible for discovering devices according to a communication protocol. It can be easily developed and deployed to the cluster to support new protocols in the system. The agent will track the state of the device and keep the Akri controller updated with the status. At the time of writing, the project has built-in support for ONVIF [42], udev [43] and OCP UA [27] discovery handlers, with an incoming proposal for CoAP [37] by us.

Using Akri, the Kubernetes cluster can carry out dynamic discovery to use new resources as they become available and move away from decommissioned/failed resources. Discovering IoT devices is usually accomplished by scanning all connected communication interfaces and enlisting all locally available resources.

Akri is also responsible for enabling applications to communicate with the device and deploying a broker Pod as intermediary. The broker is any application instructed to communicate with the device. We devised the broker as a web server that abstracts the actual communication between devices and applications behind a RESTful API. Akri should automatically find all the devices in the environment and make them available as web resources. As an instance, the agent discovers the devices regularly by scanning for CoAP resources.

Besides, the broker can offer local aggregates of device-level services, such as the combined temperature measurements of all the Things connected to it for later consumption in our flood prediction, electrical load forecasting services or precision agriculture services.

Our RESTful broker also helps to scale the number of concurrent HTTP requests by implementing highly performant cache mechanisms. The IoT resource periodically sends its sensor readings to the broker, where the values are cached locally. Each application request is then served directly from this cache without accessing the actual device, improving the average roundtrip time.

As many distributed monitoring applications are usually read-only during their operation (e.g. sensors collecting data in our case), this architecture exhibits a great scalability level. A potential goal is to enable new types of services where physical sensors can be shared with thousands of users with little impact on latency and data staleness.

5.3. Virtualisation, Interoperability and Portability

WebAssembly (Wasm) [14], first announced in 2015 and released as a Minimum Viable Product in 2017, is a nascent technology that provides strong memory isolation (through sandboxing) at near-native performance with a much smaller memory footprint. WebAssembly is a language designed to address safe, fast, portable low-level code on the web. Developers who wish to leverage WebAssembly may write their code in a higher-level (compared to bytecode) language such as C++ or Rust and compile it into a portable binary that runs on a stack-based virtual machine.

A WebAssembly trait critical for the Continuum is that programs cannot corrupt their execution environment, jump to arbitrary locations, or perform

other undefined behaviour, especially when the original language is memory-safe (e.g. Rust §**??**). At worst, a WebAssembly is subject to exploits regarding data in its memory. The memory and state encapsulation are applied at the module level rather than at the application level, meaning that a module's memory and functions cannot leak information unless explicitly exported/returned. This granularity in sandboxing is extremely important as security incidents have increasingly exploited vulnerabilities in the dependency chain. Reuse of third-party software is pervasive in modern languages like JavaScript, Rust or Go. On the other hand, the granular memory encapsulation means that even untrusted modules can be safely executed in the same address space as other code, a critical point for dynamic configuration in constrained devices and for multitenancy in the compute nodes of our architecture.

We picked WebAssembly as the technology enabling virtualisation, interoperability and portability in the Continuum for three reasons mainly. First, WebAssembly is advertised as safe *and* fast to execute. Benchmarks of Wasm runtimes on modern browsers have shown a slowdown of approximately 10% compared to native execution and almost always within 2x [14]. Second, WebAssembly provides language, hardware, and platform independency by offering a *consistent* execution platform independent of any underlying infrastructure to allow applications to run across all software and hardware types with the same behaviour. The importance of such feature for the Continuum cannot be emphasized enough. Third, the Wasm binary code is designed to be compact streamable and parallelisable. Code transmitted over the network has to be as compact as possible to reduce load times in compute nodes, save potentially expensive bandwidth snf reduce memory usage on constrained network-attached devices. As an example, a Wasm runtime can minimise latency by starting and parallelising streaming compilation as soon as function bodies arrive over the network, differently from container images. Minimising the latency is essential for increased mobility, quick release of resources, and support for low-latency use cases.

Because of this features, WebAssembly is currently under experimentation as a new method for running portable applications without containers. Ideally, WebAssembly is capable of providing significantly more lightweight isolation compared to VMs and containers for multi-tenant service execution. This idea is still in its infancy, but there has been some interest in recent years, as shown by the works done in [44], [45] and [46]. All these works focus primarily on serverless computing via WebAssembly.

We tested the feasibility of such idea by measuring several metrics: how long it takes to create and boot a Wasm-based Kubernetes Pod, how memory scales as the number of running Pods increases, and how both metrics compare to containers. The benchmark is a simple DateTime application that logs the current system time upon creation and goes into sleep. By going to sleep, the Pod does not complete, and the resources remain allocated. The log is later retrieved using the Kubernetes API to calculate the boot time.

Wasm Pods run on Krustlet [47], whereas container Pods are scheduled on the K3s [48] Kubelet. Krustlet (a Kubernetes-Rust-Kubelet) is an experimental

implementation of the Kubernetes compute node (Kubelet) API that supports Wasm as virtualisation technology. Therefore, it listens to the Kubernetes API event stream for new units of execution (Pods) and runs them under a WebAssembly System Interface (WASI) runtime (notably, Wasmtime by Mozilla [49]).

On the other hand, K3s is a fully certified Kubernetes distribution geared towards Edge environments backed by a commercial company. K3s is implemented in Go and packaged as a single binary of about 50MB in binary size. It bundles everything needed to run Kubernetes, notably the container runtime containerd [38].

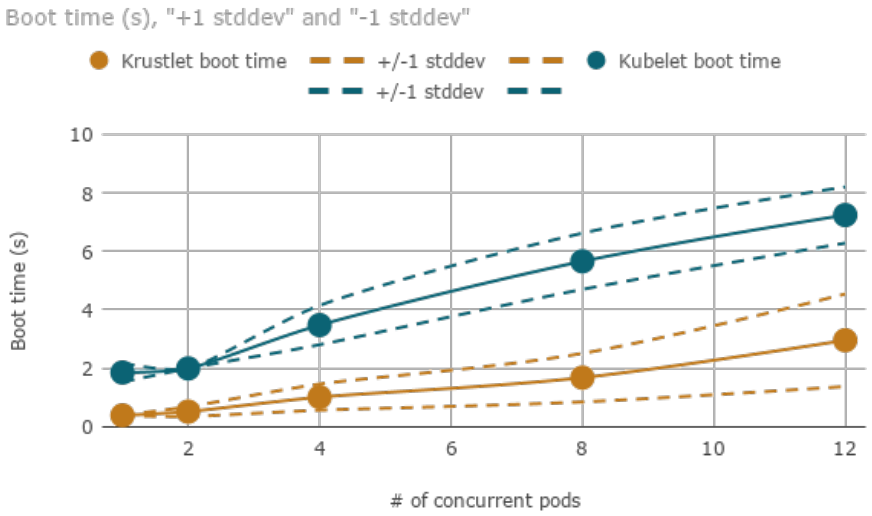

Figure 9: Average boot time for concurrent Wasm Pods.

Figure 9 shows the average boot time, along with the standard deviation, of both the Kubernetes Pods containing Wasm binaries and the conventional Pods containing containers. The benchmark concurrently deploys the Pods and repeats the process 15 times. Pods are not deleted between iterations, so that increasing memory utilisation is also collected.

The experimental results show that a Wasm-based virtualisation strategy incurs less boot time. However, the more mature container Kubelet presents a more linear curve and minor standard deviation as the number of concurrent deployments increases. Because efficient concurrency is essential as much as a fast boot time, there is no clear winner. Nevertheless, such preliminary results encourage the idea of adopting Wasm as an alternative to container technology since efficiency was not a primary design goal in the early implementations of Krustlet and Wasmtime. We expect future versions will provide even more competitive results.

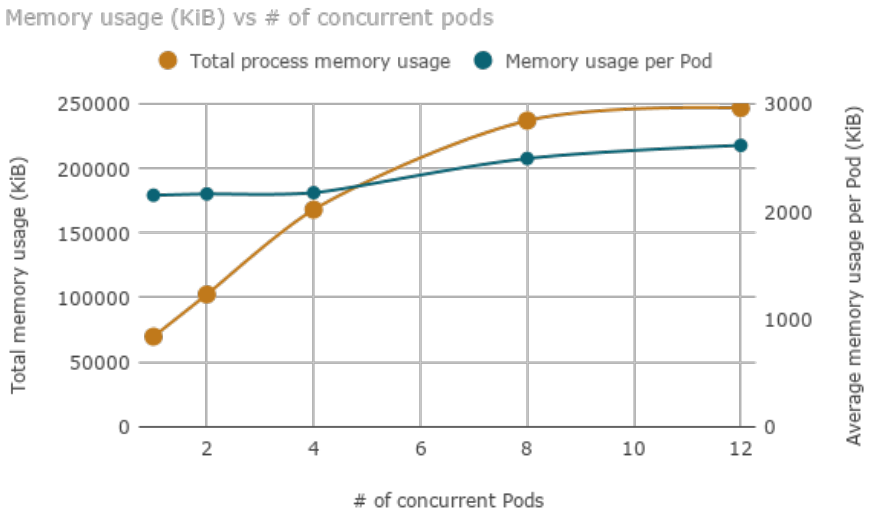

Figure 10: Average memory usage of concurrent Wasm. Pods

Figures 10 and 11 offer an overview of the memory overhead of the two different virtualisation solutions. In such a comparison, Wasm Pods are the winners in memory usage per Pod unit, but the K3s Kubelet can achieve higher total memory utilisation.Notably, the K3s Kubelet can completely use the available memory until the machine cannot even function properly. On the other hand, the Krustlet node fails to allocate new Pods even when there is sufficient memory space. The allocation results in an Out Of Memory error, and the node is still completely functional. The cause is the Rust vector allocation strategy, which doubles the capacity when its current limit is reached. As a result, when Krustlet's heap reaches about 256MB, the process demands the kernel an additional 512MB of space. Such demand cannot be met, and the Rust runtime returns an Out Of Memory error.

On a different note, as Go is a garbage-collected language, the heap utilisation is highly unpredictable. Besides, as the GC kicks in only when the heap size doubles, the hardware memory is underutilised. The freeable memory should be used for running additional pods, achieving better Pod packing. Such efficiency is crucial for edge nodes that have already limited hardware capabilities but must support multiple workloads. This consideration is another point favouring the adoption of the Rust language. Table 1 presents the system memory utilisation during an idle state.

Finally, in both technologies, the memory overhead per Pod is relatively constant. However, the Wasm Pod incurs approximately 2x-3x less overhead, allowing more efficient packing of applications on the same machine. An impor-

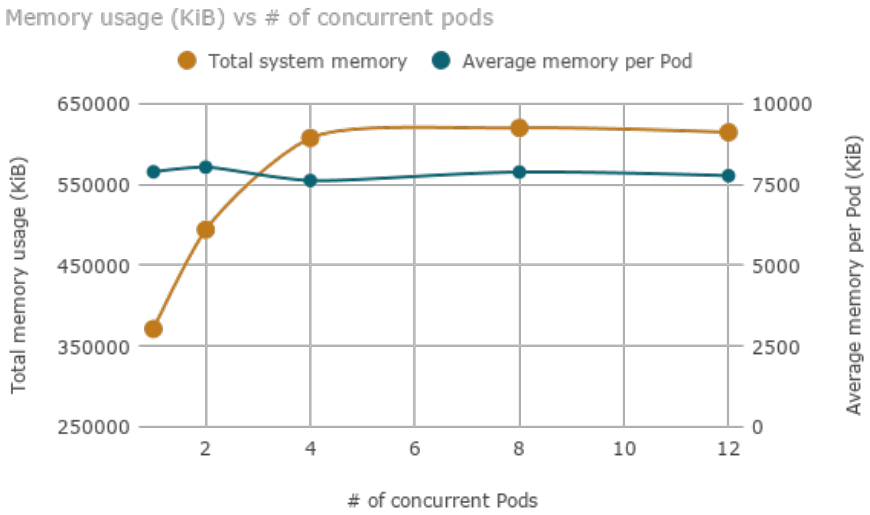

Figure 11: Average memory usage of concurrent K3s Kubelet Pods.

Table 1: Memory usage of Kubernetes on edge Raspberry Pi.

Software stack	Idle memory usage
Alpine Linux 3.12.1	50MB
Alpine Linux + K3s master	304MB
Alpine Linux + K3s agent	110MB
Alpine Linux + Krustlet	120MB

tant difference is also the fact that do not allocate more Wasm Pods when the limit is reached, but existing Pods are still completely functional. In contrast, the K3s Kubelet makes the entire node becomes completely unresponsive when maximum memory utilisation is reached. The author believes that such results pledge to favour Krustlet, as the premature Out Of Memory error can be fixed on later Wasmtime versions. In contrast, the container overhead is already the state of the art of a decade of research in container technologies.

5.3.1. Dynamic configuration

As we already mentioned, WebAssembly is critical to enable arbitrary code execution on extremely constrained devices on the Continuum,. The authors of eWASM [50] have also explored various WebAssembly-based mechanisms for memory bounds checking and have evaluated the trade-offs between efficient Wasm processing and memory consumption. Generally speaking, Just-In-Time compilers for WebAssembly exist (e.g. Wasmtime [49]) and receive more attention from the community, but their size and complexity make them unsuitable for microcontrollers.

Although WebAssembly interpreters can often be approximately 11x slower than native C [51], they help dynamically update system code and debugging but may not be applicable for code on devices susceptible to performance and energy efficiency.

Interpreting WebAssembly on microcontrollers offers a persuasive alternative to other language runtimes, e.g. Lua, which are commonly used on embedded devices to support dynamic configuration [10]. The WebAssembly standard has many features that make it appealing for embedded devices [50]. First, WebAssembly is a platform-independent Intermediate Representation that can be generated from different source languages and can run on many CPU architectures. Solving how to run WebAssembly on microcontrollers effectively allows to include the embedded world to the Continuum as an additional place of intelligent computing, rather than only as a mere data collector and dummy actuator. Forthermore, many broadly used language runtimes such as JavaScript, Lua, or Python cannot provide predictable execution and may require excessive memory for a microcontroller whereas Wasm requires no mandatory garbage collection and only a small number of runtime features around maintaining memory sandboxing. These light requirements help in an embedded adaptation.

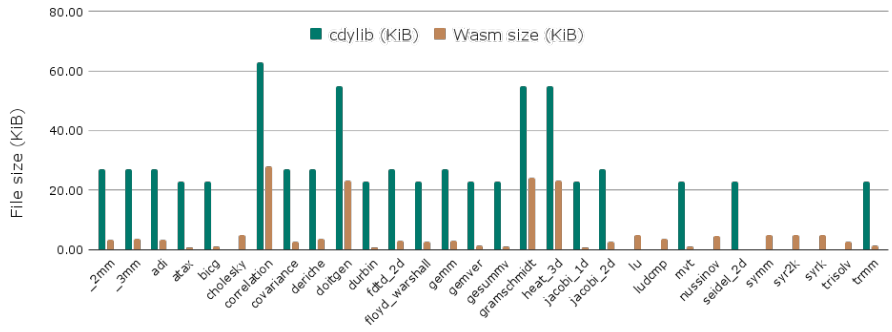

Figure 12: Comparison of Wasm size (KiB) and C dynamic library size (KiB).

Figure 12 presents a comparison of the sizes of different Wasm binaries compiled from the Polybench [52] modules. The Polybench benchmark suite offers relevant functions to embedded systems as it includes common matrix and statistical operations. We have chosen the C dynamic library size as a meaningful comparison since it is a close alternative to Wasm binary files. Both outputs have been compiled using the same LLVM toolchain and optimisation flags.

The results undeniably favour the Wasm binary format as the C dynamic lib is often many times larger. Comparing Wasm files to containers would even less relevant and greatly favour the former, as containers package a whole operative system filesystem. Even the smallest image base (Alpine Linux Mini Root Filesystem) has an additional size of about 5.5MB uncompressed.

Figure 13 offers an overview of the Wasm interpreter's execution performance on the STM32F407 microcontroller. Each Polybench benchmark has been run

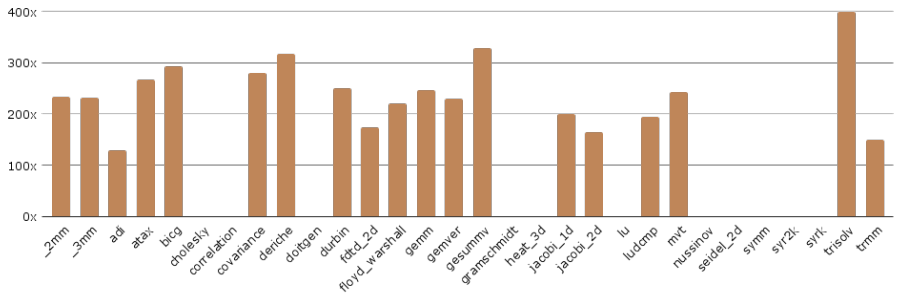

Figure 13: Comparison of Wasm interpreter performance and Rust native performance.

15 times, following the same methodology described in [51]. The results show a dramatic slowdown compared to the Rust native alternative, with a factor of 100-400 times. Such results are discouraging for the idea of using Wasm interpreters on microcontrollers to support dynamic reconfiguration.

However, it is fair to note that the Wasm interpreter we used, wasmi [53], was adapted to work on embedded devices and was not designed for highly constrained devices. wasmi is developed for blockchain execution in mind and is thus used to offer a deterministic sandboxed execution context running on cloud servers. As a result, execution performance is not paramount, unfortunately. Alternative interpreters, implemented in the C language, shows a much minor execution penalty, in the order of 30-60x slower than native [50]. Thus, it is reasonable to believe that future efforts may allow a comparable result for Rust-built Wasm interpreters. Nevertheless, 30x times execution over time can still arguably deter the usage of interpreters in microcontrollers. Future work should also provide a benchmark with respect to other popular interpreted languages commonly used in the embedded industry, e.g. Lua.

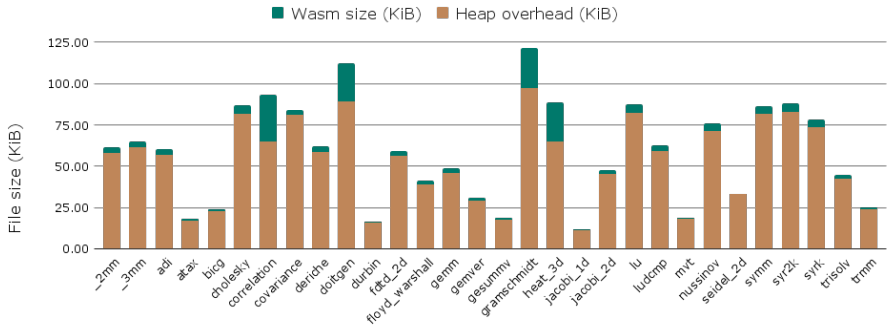

Figure 14: Comparison of Wasm size (KiB) and heap overhead (KiB).

As an additional benchmark, we have evaluated the heap overhead of interpreting Wasm on microcontrollers. Figure 14 presents a significant increase in

the heap usage with respect to the Wasm size. However, the most crucial concern is that such a heap increase is not predictable. Such unpredictability does not come again in favour of the usage of WebAssembly on microcontrollers, as embedded devices have extremely limited resources and must have predictable behaviours to ensure proper real-time execution. Such deficiency is an intrinsic issue with interpreters, as the code instructions and execution data structures must be stored in heap memory. This behaviour contrasts with the binary executables that can save and access instructions or read-only data on the more capable flash storage. Writable data is saved in the stack instead, and it can be estimated with accuracy in many production-grade toolchains like C and Ada.

Generally speaking, running Wasm on resource-constrained microcontrollers also presents a memory-design issue. Wasm's pages are 64KiB by standard, but they are too large for microcontrollers that often have between 16-256 KiB SRAM. Dynamic allocation is a common requirement even for embedded systems. However, Wasm specifies that the sandbox should expand the memory by 64KiB chunks, which is not granular enough for constrained embedded systems. As a consequence, we had to adapt the interpreter to allocate non-standard pages of 16KiB. Otherwise, it would have been impossible to execute any benchmark on the STM32F407 microcontroller, as additional heap space is required for the interpreter's internal structures and the Wasm instructions themselves.

5.4. Programmability

As the Continuum integrates compute support across the network, the attack surface greatly increases consequently. Notably, the leading cause of security vulnerabilities are memory safety bugs like data races and buffer overflows. Security has to be a top priority during software development, yet most of the infrastructure is programmed using the C, C++ programming languages. These two programming languages are chosen due to the low overhead on memory and the high processing performance. Embedded systems have favoured the two languages because of the need for low-level control over the hardware. However, C and C++ are not particularly renowned for producing secure software, as evidenced by the many vulnerabilities reported against the software written in them.

On the other hand, Rust is a strongly-typed, compiled language that uses a lightweight runtime similar to C. Unlike many other modern languages, Rust is an attractive choice for predictable performance because it does not use a garbage collector. It provides strong memory safety guarantees by focusing on "zero-cost abstractions", meaning that safety checks are done at compile-time and runtime checks (e.g. out-of-bounds access) have the minimum overhead and come with a predictable cost.

Safe Rust code is guaranteed to be free of null or dangling pointer dereferences, invalid variable values (e.g. casts are checked), reads from uninitialized memory, unsafe mutations of shared data, and data races, among other misbehaviours. The borrow checker, the most innovative feature of the language compiler, runs as part of the compilation process and catches bugs like just mentioned misbehaviours.

Lastly, thanks to the integration of LLVM, the Rust compiler can transform the Intermediate Representation (IR) to generate WebAssembly binary code. The union of Rust and WebAssembly constitutes a powerful combination. Developers can write source code in Rust to achieve high productivity and efficient memory-safe applications. On the other hand, WebAssembly can contribute with a hardened execution environment and universally portable binaries. Developers do not need to compile or distribute multiple versions (e.g. Docker image versions) of the same software.

Besides being a system language, we found Rust a sensible choice to build *any* reliable and performant software. We used it successfully throughout our work in the Continuum. We were able to run a WebAssembly interpreter and CoAP server on top of a minimal embedded runtime for extremely constrained environments like microcontrollers. Later we exploited the same language and developer experience while working on systems-level programming in Krustlet and cluster-level communication with Akri in Kubernetes. Lastly, we implemented the protocol-bridging brokers and high-level web services in Rust as well. Despite the diversity of programming levels, the fast pacing Rust community is extremely rich of libraries for each use case. On the other hand, because the relatively early age of Rust, a good level of craftsmanship is required as most of the open-source libraries published by the community are not production battle-tested and present rough edges, like unimplemented critical features, which has to develop on its own. Embedded programming typically require using unstable language features as well, which hinders the adoption of Rust for critical long-lived embedded applications or even industry-wide adoption in production.

Figure 15 summarises the key technologies mentioned above within the reference architecture of the infrastructure.

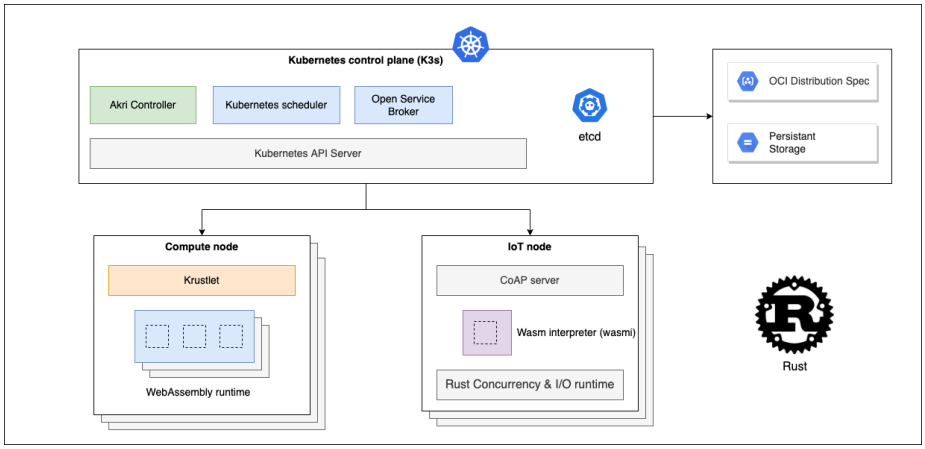

Figure 15: Technology baseline for the reference infrastructure architecture.

We briefly also revisit our previous system for weather-based service as representative of how the above technologies can be used in synergy:

- Sensor nodes: the arbitrary code execution is safely enabled by running a Rust-based WebAssembly interpreter on the Wasm binary file. The portable low-overhead Wasm format unlocks transfer of computation to dynamically instruct the sensor nodes about the preprocessing logic on a case-by-case basis;
- Broker nodes: brokers subscribe to the sensor nodes which expose their data as REST resources via CoAP messages, and the devices periodically send CoAP updates. In turn, the brokers forward them to the cluster as WebSocket packets. The broker ensures that both parts, IoT nodes and services nodes, are independent as far as they agree to communicate following the REST architecture. Service nodes typically use REST over HTTP, while sensor nodes prefer CoAP;
- Service nodes: they request the weather information as services to the service platform implementing the Open Service Broker API. The latter also exposes conventional services like storage, offering both locally provided solutions and cloud-based alternatives on Google Cloud under the same RESTful interface.

At the time of writing, it has not been possible to implement a web server (e.g. to act as electrical load forecasting service) and compile the application to Wasm. We successfully compiled an neural network inference model to WebAssembly, but there is an underlying issue with implementing network servers as there is neither sufficient network API nor multi-threading support in the standard yet.

First, the current WebAssembly System Interface (WASI) standard only contains a few methods for working with sockets that are not enough for complete networking support. Adding support for connecting to sockets is fundamental to allow Wasm modules to connect to web servers, databases, or any service.
Second, the lack of concurrency primitives means that a server running in WebAssembly is single-threaded, or its implementation has to be significantly more complex (e.g. Node.js's event loop [54]). This limitation severely narrows the workload capabilities of the server. At the time of writing, the WASM spec has outlined a thread and atomics proposal intending to speed up multi-threaded applications. The proposal is still in the early stage, and it is implemented only in web browsers, behind an experimental flag.

6. Conclusion

This paper has presented a computing continuum vision and the challenges that the industrial and academic community will face implementing such a paradigm. We have also presented a candidate architecture for the infrastructure and implemented a Proof of Concept based on the technologies available

at the time of writing. Such POC has been applied to real-world use cases such as electrical load forecasting, precise agriculture and flood prediction. The net result is a working prototype based on the presented architectural and software concepts. The implementation is publicly available on GitHub [55].

The WebAssembly standard is a potential enabling technology, but it is still in its infancy. As shown by the experimental results, WebAssembly is mainly suited for pure computational functions. The lack of multi-threading and a mature network interface severely limit the space of real-world applications. Most of the production use cases of WebAssembly involve pure computation scenarios, typically machine learning and relatively simple serverless functions. Some projects try to overcome the API limitations by providing custom system capabilities, to the detriment of the platform-independency feature of WebAssembly, unfortunately.

AppendixA. My Appendix

Appendix sections are coded under `\appendix`.

[1] P. Mell, T. Grance, et al., The nist definition of cloud computing, 2011.

[2] Ericsson, Ericsson mobility report, https://www.ericsson.com/en/mobility-report, 2020. Accessed: 2022-03-05.

[3] Gartner, Leading the iot, https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf, 2017. Accessed: 2022-03-05.

[4] B. Chen, J. Wan, A. Celesti, D. Li, H. Abbas, Q. Zhang, Edge computing in iot-based manufacturing, IEEE Communications Magazine 56 (2018) 103–109.

[5] P. Beckman, J. Dongarra, N. Ferrier, G. Fox, T. Moore, D. Reed, M. Beck, Harnessing the computing continuum for programming our world, Fog Computing: Theory and Practice (2020) 215–230.

[6] E. Nygren, R. K. Sitaraman, J. Sun, The akamai network: a platform for high-performance internet applications, ACM SIGOPS Operating Systems Review 44 (2010) 2–19.

[7] A. Botta, W. De Donato, V. Persico, A. Pescapé, Integration of cloud computing and internet of things: a survey, Future generation computer systems 56 (2016) 684–700.

[8] S. Latre, J. Famaey, F. De Turck, P. Demeester, The fluid internet: service-centric management of a virtualized future internet, IEEE Communications Magazine 52 (2014) 140–148.

[9] M. AbdelBaky, M. Zou, A. R. Zamani, E. Renart, J. Diaz-Montes, M. Parashar, Computing in the continuum: Combining pervasive devices and services to support data-driven applications, in: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), IEEE, pp. 1815–1824.

[10] R. Brzoza-Woch, M. Konieczny, P. Nawrocki, T. Szydlo, K. Zielinski, Embedded systems in the application of fog computing—levee monitoring use case, in: 2016 11th IEEE Symposium on Industrial Embedded Systems (SIES), IEEE, pp. 1–6.

[11] P. Pace, G. Aloi, R. Gravina, G. Caliciuri, G. Fortino, A. Liotta, An edge-based architecture to support efficient applications for healthcare industry 4.0, IEEE Transactions on Industrial Informatics 15 (2018) 481–489.

[12] J. He, J. Wei, K. Chen, Z. Tang, Y. Zhou, Y. Zhang, Multitier fog computing with large-scale iot data analytics for smart cities, IEEE Internet of Things Journal 5 (2017) 677–686.

[13] T. L. Foundation, Kubernetes, https://kubernetes.io/, 2021. Accessed: 2022-03-05.

[14] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, J. Bastien, Bringing the web up to speed with webassembly, in: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 185–200.

[15] S. Haller, S. Karnouskos, C. Schroth, The internet of things in an enterprise context, in: Future Internet Symposium, Springer, pp. 14–28.

[16] N. Grozev, R. Buyya, Inter-cloud architectures and application brokering: taxonomy and survey, Software: Practice and Experience 44 (2014) 369–390.

[17] Google, Protocol buffers, https://developers.google.com/protocol-buffers, 2021. Accessed: 2022-03-05.

[18] C. Pahl, S. Helmer, L. Miori, J. Sanin, B. Lee, A container-based edge cloud paas architecture based on raspberry pi clusters, in: 2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW), IEEE, pp. 117–124.

[19] P. Bellavista, A. Zanni, Feasibility of fog computing deployment based on docker containerization over raspberrypi, in: Proceedings of the 18th international conference on distributed computing and networking, pp. 1–10.

[20] R. Pi, Products, https://www.raspberrypi.org/products/, 2021. Accessed: 2022-03-05.

[21] Docker, Docker image specification 1.0.0, https://github.com/moby/moby/blob/master/image/spec/v1.md, 2021. Accessed: 2022-03-05.

[22] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al., Cloud programming simplified: A berkeley view on serverless computing, arXiv preprint arXiv:1902.03383 (2019).

[23] S. K. Mohanty, G. Premsankar, M. Di Francesco, et al., An evaluation of open source serverless computing frameworks., in: CloudCom, pp. 115–120.

[24] M. S. Elbamby, C. Perfecto, C.-F. Liu, J. Park, S. Samarakoon, X. Chen, M. Bennis, Wireless edge computing with latency and reliability guarantees, Proceedings of the IEEE 107 (2019) 1717–1737.

[25] S. Y. Jang, Y. Lee, B. Shin, D. Lee, Application-aware iot camera virtualization for video analytics edge computing, in: 2018 IEEE/ACM Symposium on Edge Computing (SEC), IEEE, pp. 132–144.

[26] N. Naik, Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http, in: 2017 IEEE international systems engineering symposium (ISSE), IEEE, pp. 1–7.

[27] S. Grüner, J. Pfrommer, F. Palm, Restful industrial communication with opc ua, IEEE Transactions on Industrial Informatics 12 (2016) 1832–1841.

[28] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, Q. Li, Lavea: Latency-aware video analytics on edge computing platform, in: Proceedings of the Second ACM/IEEE Symposium on Edge Computing, pp. 1–13.

[29] W. He, G. Yan, L. Da Xu, Developing vehicular data cloud services in the iot environment, IEEE transactions on industrial informatics 10 (2014) 1587–1595.

[30] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, X. Yang, A survey on the edge computing for the internet of things, IEEE access 6 (2017) 6900–6919.

[31] dotcom monitor, Visual traceroute, https://www.dotcom-monitor.com/wiki/knowledge-base/visual-traceroute-graphical-tool/, 2021. Accessed: 2022-03-05.

[32] L. Li, K. Ota, M. Dong, When weather matters: Iot-based electrical load forecasting for smart grid, IEEE Communications Magazine 55 (2017) 46–51.

[33] B. Keswani, A. G. Mohapatra, A. Mohanty, A. Khanna, J. J. Rodrigues, D. Gupta, V. H. C. De Albuquerque, Adapting weather conditions based iot enabled smart irrigation technique in precision agriculture mechanisms, Neural Computing and Applications 31 (2019) 277–292.

[34] R. Fielding, Representational state transfer (rest), https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm, 2000. Accessed: 2022-03-05.

[35] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, et al., The quic transport protocol: Design and internet-scale deployment, in: Proceedings of the conference of the ACM special interest group on data communication, pp. 183–196.

[36] C. Foundry, Open service broker, https://www.openservicebrokerapi.org/, 2016. Accessed: 2022-03-05.

[37] C. Bormann, A. P. Castellani, Z. Shelby, Coap: An application protocol for billions of tiny internet nodes, IEEE Internet Computing 16 (2012) 62–67.

[38] T. L. Foundation, containerd, https://containerd.io/, 2021. Accessed: 2022-03-05.

[39] Docker, Swarm mode overview, https://docs.docker.com/engine/swarm/, 2021. Accessed: 2022-03-05.

[40] B. I. Ismail, E. M. Goortani, M. B. Ab Karim, W. M. Tat, S. Setapa, J. Y. Luke, O. H. Hoe, Evaluation of docker as edge computing platform, in: 2015 IEEE Conference on Open Systems (ICOS), IEEE, pp. 130–135.

[41] D. Labs, Akri, https://github.com/deislabs/akri, 2021. Accessed: 2022-03-05.

[42] ONVIF, Onvif, https://www.onvif.org/, 2021. Accessed: 2022-03-05.

[43] archlinux, udev, https://wiki.archlinux.org/index.php/udev, 2021. Accessed: 2022-03-05.

[44] A. Hall, U. Ramachandran, An execution model for serverless functions at the edge, in: Proceedings of the International Conference on Internet of Things Design and Implementation, pp. 225–236.

[45] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, G. Parmer, Sledge: a serverless-first, light-weight wasm runtime for the edge, in: Proceedings of the 21st International Middleware Conference, pp. 265–279.

[46] S. Shillaker, P. Pietzuch, Faasm: lightweight isolation for efficient stateful serverless computing, in: 2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20), pp. 419–433.

[47] D. Labs, Krustlet, https://github.com/deislabs/krustlet, 2021. Accessed: 2022-03-05.

[48] Rancher, k3s, https://k3s.io/, 2021. Accessed: 2022-03-05.

[49] B. Alliance, wasmtime, https://github.com/bytecodealliance/wasmtime, 2021. Accessed: 2022-03-05.

[50] G. Peach, R. Pan, Z. Wu, G. Parmer, C. Haster, L. Cherkasova, ewasm: Practical software fault isolation for reliable embedded devices, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39 (2020) 3492–3505.

[51] wasm3, wasm3 performance, https://github.com/wasm3/wasm3/blob/main/docs/Performance.md, 2021. Accessed: 2022-03-05.

[52] T. Yuki, Understanding polybench/c 3.2 kernels, in: International workshop on Polyhedral Compilation Techniques (IMPACT), pp. 1–5.

[53] Parity, wasmi, https://github.com/paritytech/wasmi, 2021. Accessed: 2022-03-05.

[54] Node.js, The node.js event loop, https://nodejs.dev/learn/the-nodejs-event-loop, 2021. Accessed: 2022-03-05.

[55] J. Hu, fedra-thesis, https://github.com/jiayihu/fedra-thesis, 2021. Accessed: 2022-03-05.