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 a ubiquitous platform for every user and any use. A flurry of organic changes to its infrastructure and interfaces has accompanied this vast transformation.

In a little over a decade of existence over the Internet, the Cloud [1] has earned users tremendous 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 (immense) data centres; the intrinsic vulnerability to single-point failure of its centralised model; the latency caused by storing and processing in the Cloud all the data ingested at the periphery of it (allusively called the Edge); the threats to data security and privacy incurred by exposing sensitive data to Edge-Cloud transfers.

Vision. Dramatic improvements in mobile connectivity occurred in the last two decades, for ubiquity, reliability, and affordability, have allowed anyone to access the Internet from anywhere and at any time. 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 centralisation increasingly less practical [1]. 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 of interest (latency, privacy, energy, etc.).

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 Edge, allowing access to the physical world to occur in a more distributed and dynamic manner, and favouring the creation of numerous novel latency-free, private and secure, energy-savvy services.

This vision of seamless integration extends the view put forward by [6], 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 [7],

[8], 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 traced to a variety of application areas, from managing extreme events (e.g. environmental monitoring [9]) to optimising everyday processes (e.g. manufacturing [4]) and improving life quality (e.g. healthcare [10] and smart cities [11]). Figure 1 attempts to capture said vision pictorially. Enacting this vision requires that sufficient compute capa-

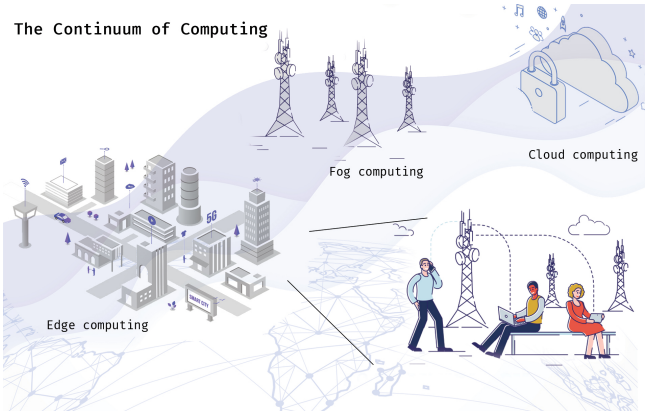

Figure 1: Pictorial view of the Continuum of Computing comprising the Edge, Fog and Cloud computing. Fog computing denotes computational resources situated at a few, usually one, hop away from end users.

bilities 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 afford elastic provisioning of the end-to-end service delivery infrastructures virtualised to allow scaling as a function of user demand and service requirements.

Contribution. No matter how attractive this vision may be, however, very few efforts have been carried out to date, to the best of our knowledge, to explore the development of suitable enabling platforms. The work discussed in this paper aims to fill this gap. Our research has leveraged a parallel effort in assessing various state-of-the-art technologies to gauge the distance between them (and their integration) and our vision. We developed a Proof of Concept (PoC) implementation of the Continuum infrastructure, combining mature Cloud tools

like the Kubernetes [12] orchestrator with innovative open-source solutions, in order to address many of the challenges that arise when attempting to integrate the Cloud into a seamless Continuum with the Edge. We enumerate such challenges in Section 2.

Subsequently, in Section 3, we present a high-level reference architecture for the Continuum and proceed to implement a PoC of such architecture. Notably, we explored the discoverability of Edge nodes in a Kubernetes cluster, the ubiquitous interoperability of web services, and thoroughly experimented with the nascent sandboxing standard WebAssembly [13] to bring container-like virtualisation and portability on comparatively powerful machines and constrained devices. In Section 5 we present the comprehensive technology selection behind these features. For each surveyed technology, we describe its role with regard to the Continuum and expose its consequent merits and shortcomings.

In our findings, we noticed that many of today's tools and technologies fit rather well, in principle, in our vision of the Continuum, which we take as a sign of convergence in the organic trends of evolution in many areas of software development like Cloud, Edge and Web. There is still, however, a substantial gap in terms of viability. On the one hand, 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 hand, nascent solutions like WebAssembly are still in a state of infancy and cannot provide the maturity requested by the industry to elevate them to production use. Good illustrations of such points are the pooor performance of interpreted WebAssembly in constrained devices and the lack of a standard networking WebAssembly interface, which hinder 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

Several challenges lay ahead in the realisation of the Continuum as we envision it. Besides featuring extreme heterogeneity, current Edge technology most notably lacks support for service orientation, interoperability, orchestration, reliability, efficiency, availability, and security. 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 argue that service orientation is 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 centred 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 [14].

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 the present day, where a multitude of ad-hoc programs are developed that are confined to single places of the network and permanently cement the behaviour of the associated devices [5].

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

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 [15].

A second critical limitation is 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 [16], 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 [14]. 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 PoC).

Orchestration. The transition to the Continuum will require coordinating and scheduling the operation of multiple distributed service components. The complexity of that endeavour makes orchestration essential, over and above the rating it enjoys from DevOps adopters [17]. 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 various fields, including wireless cellular networks, distributed systems, virtualisation, platform management, and requires mobility handover and service migration at local and global scales.

Virtualisation. The rapid pace of innovation in data centres and application platforms has transformed how organisations build, deploy, and manage services. Container-based virtualisation, 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 an Edge-friendly containerisation 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. Nonetheless, we reckon that the current state of containerisation technology still comes at 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 a Gigabyte of memory even when idle, as shown in our evaluation in Table 5.3. Besides, containers incur 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 enable 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 sys-

tem 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 the necessary storage capacity or Operating System components.

A further challenge of dynamic configuration is striking an acceptable compromise between warranting isolated execution and containing the corresponding loss of efficiency and increased energy consumption. A common memory-safe execution technique is to adopt interpreted languages that provide type and memory safety. For instance, the authors of [9] 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, seeking interoperability in lieu of standardisation. Service-oriented architectures are ideal in this regard as they encapsulate functionality in services that can expose a common interface, abstracting away inner idiosyncrasies.

An infrastructure that allows connecting and integrating diverse 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. Such flexibility is crucial for the Edge and IoT, which will undoubtedly 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 containerisation are free to select 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 (e.g. base OS) with other containers, thus shedding a 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, making it much harder for programmers to make native 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 archi-

tectures, for processor or OS. However, this nice feature 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 impedes using conventional containers, further impairing portability across the Continuum.

Portability also relates to programmability, in that the choice of programming language may favour 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 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 synchronisation, 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 should be highly reactive and 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 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 centre of the Cloud.

3. A system-level view of the Continuum

3.1. Preamble

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 centre, at the centre of the Cloud.

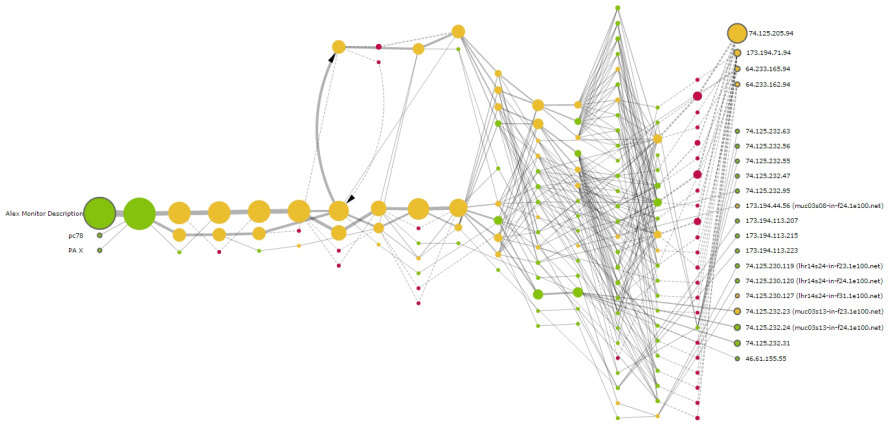

Figure 2: Traceroute virtualisation of an IP packet reaching google.com. The left green nodes are the source nodes, while packets travel across to the extreme right to servers located in data centers. Source: [31].

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 [17]. The economic incentive ramps at the first and the last mile of the source-to-destination path (Cloud data centres and IoT nodes, respectively), with very little interest attached to investing in peering points, which consequently 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 realisation suggests preferring processing at the Edge than causing network pressure. Offloading some compute tasks from IoT sensors or actuator nodes to the Edge is likely to be more energy-efficient. This strategy may not always be as convenient, though. 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 data and computation movement.

3.2. Cluster federation

Figure 3 depicts the basic building blocks of the system, as we envision it to attain the sought dynamism of computation. *Cluster nodes* allow forming

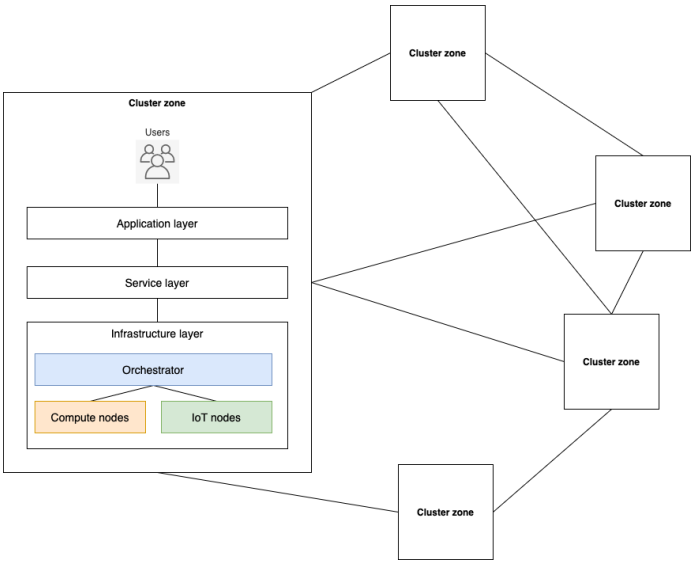

Figure 3: A high-level view of a federated set of cluster nodes.

flexible, agile, and geographically bound aggregates, called *cluster zones*. 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.

As we envision it, the system dynamically instantiates and schedules services along the path from source to destination, based on application-specific requirements and constraints. If a single cluster zone lacks hardware, software or data resources to meet the user needs, it will 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 neighbouring regions. In the Continuum, services should follow the user movements without significant outage or perturbation.

User applications running on a single cluster node are given access to requested 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 needing 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.

We leave the federation architecture as an open research question for the future of the Continuum, owing to the comparatively early stage of maturation of our concept, and the broad and challenging scope of the topic. In the following, we limit ourselves to studying the infrastructure architecture, which is a fundamental enabler to the federation layer.

3.3. Infrastructure architecture

The infrastructure layer comprises 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.

Figure 4 portrays the reference architecture of the Continuum infrastructure as we envision it.

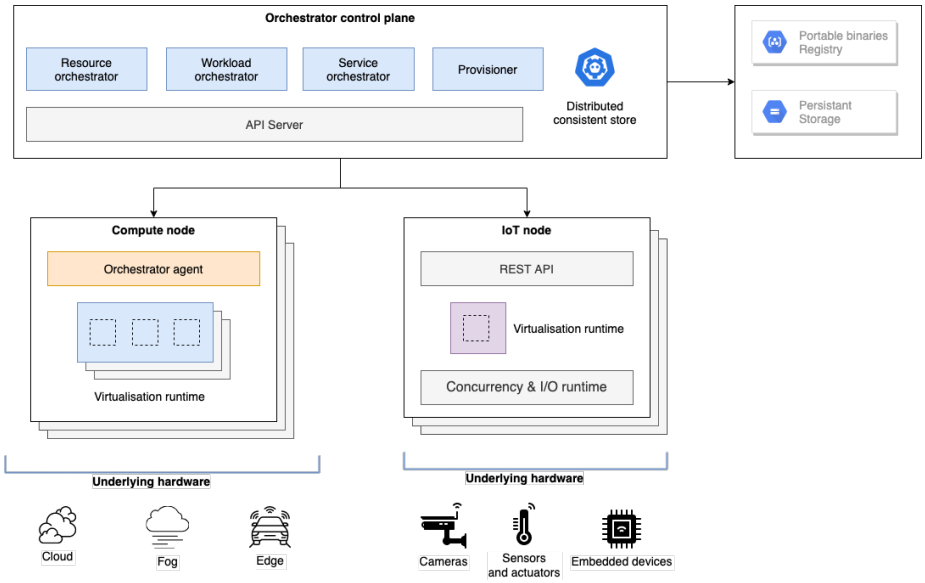

Figure 4: Reference architecture for the infrastructure.

3.3.1. 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 to increase the number of cluster nodes to host the incoming application. Such nodes can be provisioned from local machines or anywhere in the network, preferably 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. availability, utilisation, priorities, constraints.

3.3.2. Compute nodes

Each machine in the cluster that is available for services and applications is a compute node. Each of these nodes implements the orchestrator agent runtime with various responsibilities. First, it collects local information such as resource consumption metrics 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 reports to the control plane.

3.3.3. IoT nodes

IoT nodes are embedded devices that act as sensors or actuators, provided as services to the cluster. The IoT nodes are heterogeneous in runtime implementation and communication protocols. Applications in the cluster interface with them via brokers provisioned by the cluster, as we discuss in Section §5.2. 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.

3.3.4. Underlying infrastructure

One of the main requirements of the infrastructure architecture is the flexibility in being deployed on a multitude of platforms. Accordingly, 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. Use case: Weather-based services

As a practical example to guide the architecture's implementation, we apply the Continuum system design to weather-based services. The emergence of efficient sensing methods and IoT technologies are giving the opportunity to record and analyse possible influences of weather factors in many areas like flood warning [9], electrical load forecasting [32] and precision agriculture [33].

For instance, weather relevant attributes are of great significance for electrical load forecasting and include values like temperature, air pressure, vapour pressure, precipitation, evaporation, wind speed, and sunshine duration. An interesting addition is that detailed weather condition data sometimes may be captured solely 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 energy consumption. These data are also typically preprocessed to return the maximum, minimum, and average values and then normalised to generate final inputs. This peculiar trait showcases how essential it is, for many services in the Continuum, that arbitrary code may execute safely and swiftly.

Typical parameters for precision agriculture are soil moisture content, soil temperature, surrounding temperature, 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.

In a flood warning system, 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. 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. In this case, the distributed architecture of the Continuum is a viable architecture for advanced telemetry services with distributed intelligence.

To meet the requirements of the cited 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;

- 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 broker 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

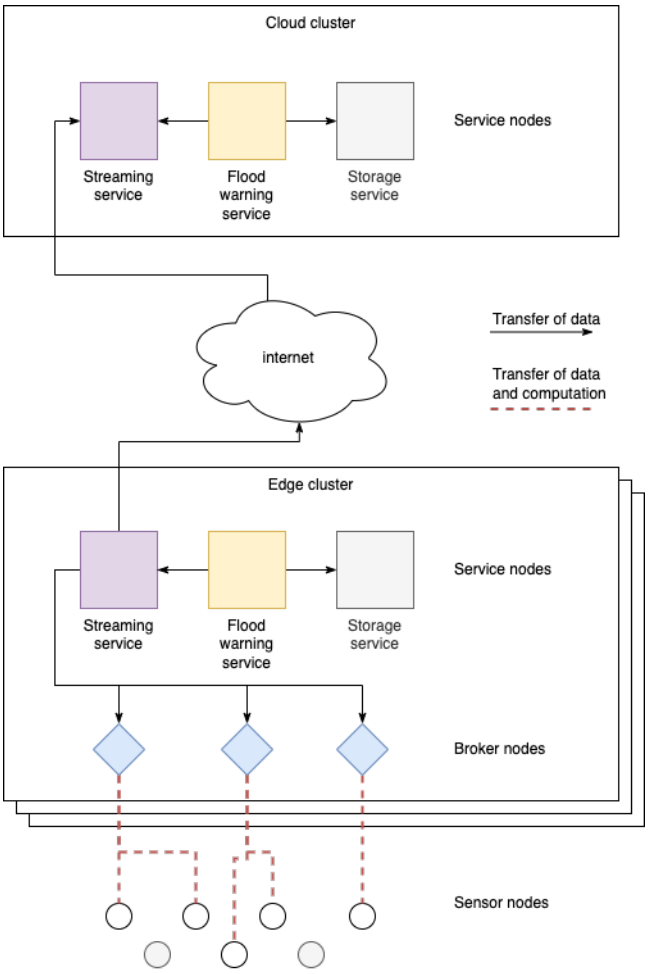

Figure 5: Architecture for the flood warning system.

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, the communication channels may become unavailable during flood threat scenarios, so the system must perform a localised assessment. Unfortunately, the loss of communication is unpredictable, but the system must quickly adapt to that eventuality. Such computing dynamism is a perfect scenario where the Continuum shines compared to relatively static Cloud-only, Edge-only, or predefined Cloud+Edge architectures.

5. Technology selection and evaluation

We now proceed to illustrating the technology baseline we adopted 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 measurements related to the technology.

For the evaluations, we have used the following devices:

- 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. We trust these empirical results generalise 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 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 the Web. However, new communication protocols (e.g. CoAP, which we discuss in Section 5.1) 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 threefold.

First, REST resources are an information abstraction that allows servers to make any information or service available, identified via Uniform Resource Identifiers (URIs). For example, this allows sensor nodes to act as a server and own the resource's original state. 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 architectural paradigm, Edge nodes can 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 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, 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 to bring IoT nodes into the Continuum and to enable multiple equivalent services offered by alternative Cloud providers to coexist on it.

Third and last, REST enables high-level interoperability between RESTful protocols through proxies or, more generally, intermediaries that behave as server to a client and play as client with respect to another server. REST intermediaries fit well with the assumption that not every device must offer RESTful interfaces directly. Such flexibility suitably accommodates the diversity of communication protocols on the Edge.

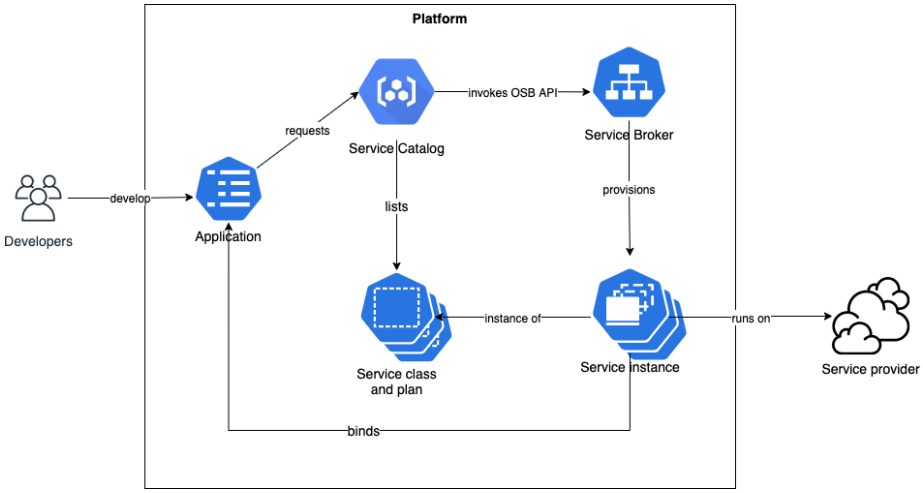

Figure 6: The Open Service Broker architecture.

Open Service Broker. With RESTfulness in mind, we realised 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 depicted in Figure 6.

In the Continuum platform, providers control access to services and payment plans but permit developers to add their own services to the catalogue. In this manner, we expect that over time a rich ecosystem of services may be developed and tapped from simple well-documented RESTful interfaces.

As Cloud standards still struggle to gain traction, however, we need to bridge the heterogeneity gap between platforms. To this end, 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 is an affordable endeavor. The service broker translates RESTful requests from the platform to service-specific operations such as creating, updating, deleting, and 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 experience for application developers who consume these services.

Over the years, the API interface of the OSB has matured considerably, learning from the experience of a wide range of marketplace services and Cloud vendors, such as Microsoft Azure and Huawei Cloud. The current standard version 2.13 is entirely designed around asynchronously provisioned services and provides valuable guidance for challenging situations such as service failures. The OSB guidance ensures consistent semantics and interoperability across various service behaviours. Sadly though, service dependency remains a pain point that needs to be coped with, as for example in the use case we have presented in Section 4. Currently, the OSB standard does not support a parent-child relationship model between services, whose handling is left inconveniently to the discretion of the broker author. The problems that arise from service dependency include whether to publish multiple services as standalone packages and how to share credentials between services, provision and remove them in the proper order, and solve all these issues uniformly across all platforms.

CoAP. To include IoT nodes in our 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, supporting built-in discovery of resources. As a result, CoAP easily interfaces with HTTP for integration with web services while meeting specialised IoT requirements such as multicast support, very low overhead and simplicity for constrained environments.

Another advantage of CoAP is that it supports a familiar and intuitive pattern already in use for development with standard web technologies, which affords a smoother learning curve.

We made CoAP nodes interoperable with the rest of the Continuum by following the REST architecture's proxy pattern, as depicted in Figure 7. We built intermediaries (discussed in §5.2) that speak CoAP on one side and HTTP on the other without encoding specific application knowledge. Because equivalent methods, response codes, and options are present in HTTP and CoAP protocols, the mapping between them is straightforward. Consequently, the in-

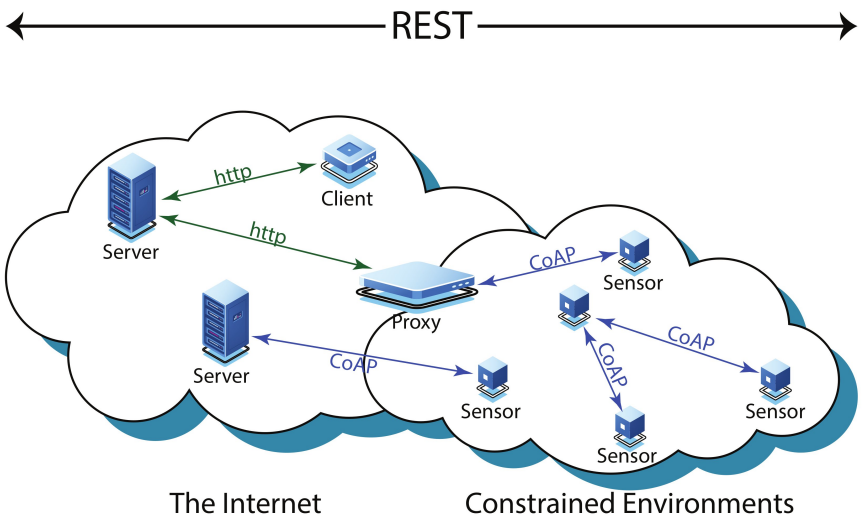

Figure 7: The REST architecture enhanced with CoAP. Source [37].

termediary can discover CoAP resources and make them available at regular HTTP URIs, enabling web services to access CoAP servers transparently in the service platform, as mentioned earlier.

5.2. Orchestration

Kubernetes [12] is an open-source orchestration framework designed to manage containerised workloads on clusters, originated from Google's experience with Cloud services. Two notable features make Kubernetes especially attractive 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 virtualisation platform between Cloud and Edge nodes (cf. Section 2).

Second, Kubernetes provides users with a wide range of options for managing their Pods (the most basic unit of deployment in Kubernetes) 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. More importantly, labels can be used also to force the scheduler to colocate services that communicate predominantly within the same availability zone, which improves latency very much and paves the way for context-aware services.

One more reason for our picking Kubernetes over Docker Swarm [39] was 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. However, support for multitenancy is a must for our service platform.

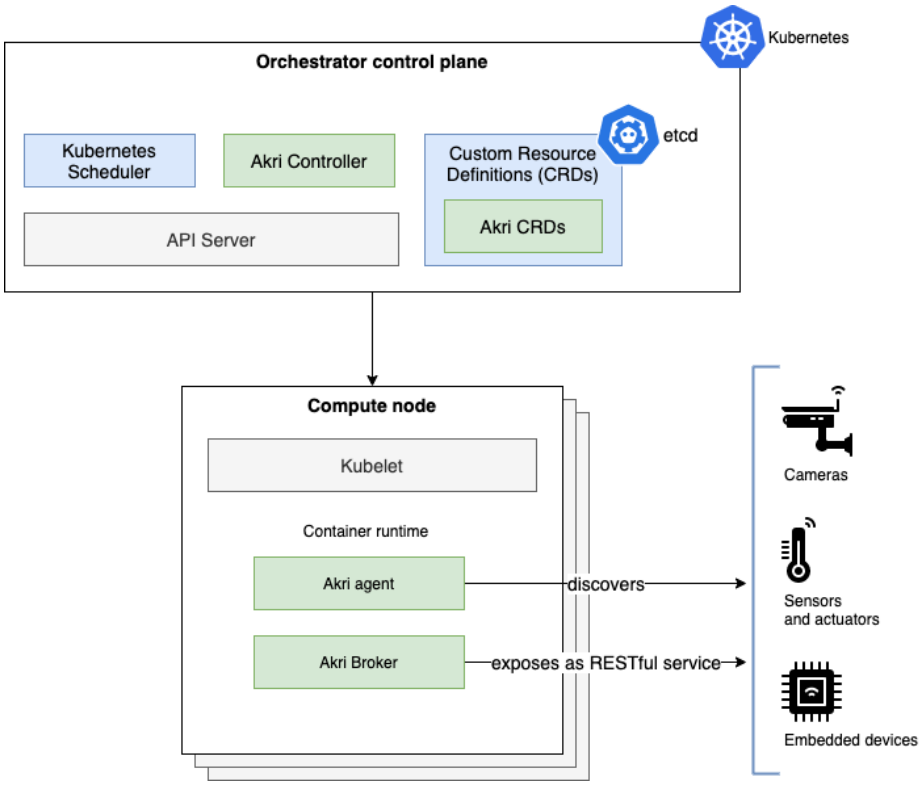

Figure 8: The Akri architecture.

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, depicted 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) that extends 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 for the agent container.

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 device's state 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. For instance, the agent regularly discovers the devices by scanning for CoAP resources.

The broker may also 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, with benefits on 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 great scalability. 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. However, at the time of writing, this is still a very distant achievement.

The Kubernetes Device Plugin API heavily influences the current Akri architecture. Such interface, already considered experimental by the Kubernetes community, was designed for hardware attached to compute nodes, e.g. GPUs. However, IoT devices can live independently from the nodes, and most of them do. Akri expects a 1:1 relationship between compute node and device, whereas most IoT devices do not have any kind of relationship to any node per se. This mismatch has several undesired consequences, including, principally, scalability and resiliency.

Another pain point in Akri's current state is that the project is still concentrating its efforts on allowing users to expose IoT nodes as services inside the Kubernetes cluster and on supporting a wide variety of IoT protocols. Regrettably, however, the project lacks more advanced yet very needed features for implementing software caching or assuring high availability or autoscaling in IoT scenarios. Such features are admittedly harder to provide but highly needed to bring the Cloud to the Edge and vice versa, an essential preliminary

step to the Continuum.

5.3. Virtualisation, Interoperability and Portability

WebAssembly (Wasm) [13], 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 that is especially critical for the Continuum is that program code cannot corrupt their execution environment, jump to arbitrary locations, or perform other undefined behaviour (which memory-safe languages such as Rust, cf. §5.4, contribute to preventing). Thanks to that execution guarantee, a WebAssembly may suffer only data exploits, which are mitigated by applying memory and state encapsulation at the module level rather than the application level. In that manner, 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, 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 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, almost always within 2x [13]. 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 import of such a feature for the Continuum cannot be emphasised 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 and reduce memory usage on constrained network-attached devices. For 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. Reducing latency is essential for increased mobility, quick release of resources, and support for low-latency use cases.

WebAssembly is currently looked at as a candidate method for running portable applications without containers. Ideally, WebAssembly can provide significantly more lightweight isolation than VMs and containers for multi-tenant

service execution. This idea is still in its infancy, but there has been some interest in recent years [44], [45] and [46], especially for serverless computing.

We tested the feasibility of employing WebAssembly in our PoC 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. The Pod does not complete by going to sleep, 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], while container Pods are scheduled on K3s [48] Kubelet. Krustlet (a Kubernetes-Rust-Kubelet stack) 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, Mozilla's Wasmtime [49]).

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 size. It bundles everything needed to run Kubernetes, notably the container runtime containerd [38].

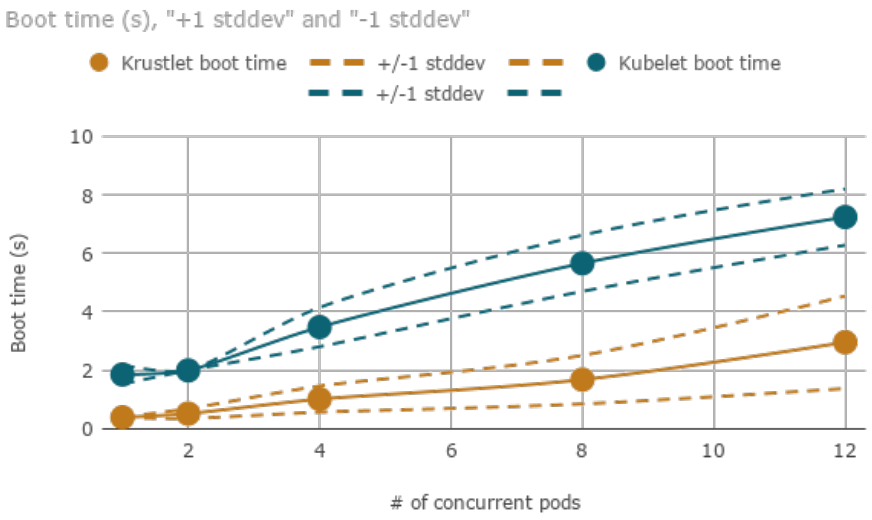

Figure 9: Average boot time for concurrent Wasm Pods.

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, there is no clear winner because efficient concurrency is essential as much as fast boot time. 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.

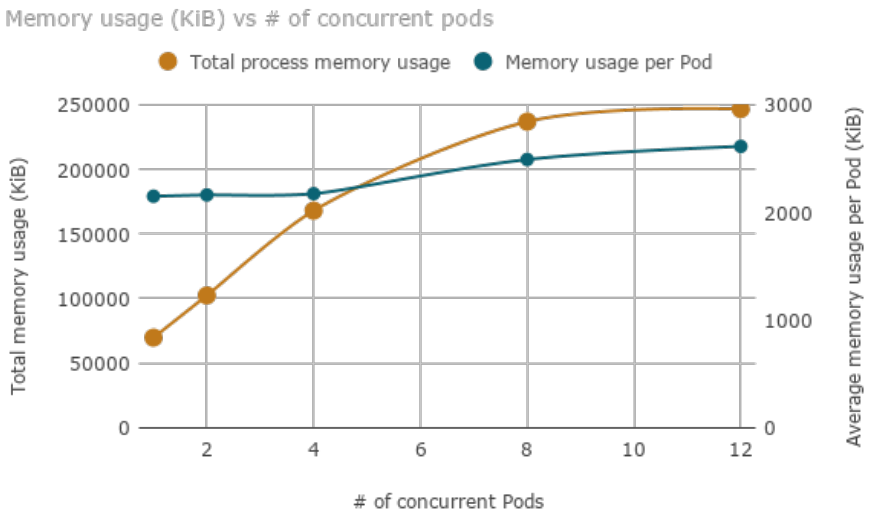

Figure 10: Average memory usage of concurrent Wasm Pods.

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 unit, but K3s Kubelet can achieve higher total memory utilisation. Notably, K3s Kubelet may use up all available memory until the machine becomes unable to function properly. Conversely, the Krustlet node fails to allocate new Pods even with sufficient memory space. The allocation results in an Out Of Memory error because of Rust's default allocation strategy, but the node is still completely functional.

On a different note, as Go is a garbage-collected language, heap utilisation is known to be highly unpredictable. Besides, as the GC kicks in only when the heap size doubles, 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 in favor of adopting Rust. Table 1 reports the system memory utilised by an idling node.

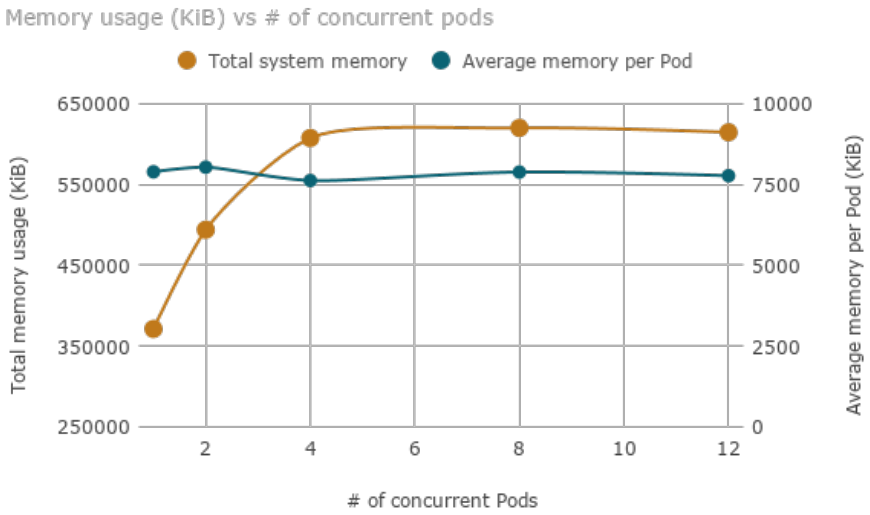

Figure 11: Average memory usage of concurrent K3s Kubelet Pods.

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

Finally, the memory overhead per Pod is relatively constant in both technologies. However, the Wasm Pod incurs approximately 2x-3x less overhead, which allows more efficient packing of applications on the same machine. Another significant difference is that Krustlet does not allocate more Wasm Pods when the limit is reached, but existing Pods are still completely functional. Conversely, the K3s Kubelet makes the entire node completely unresponsive when maximum memory utilisation is reached. Arguably, such results warrant favouring Krustlet, as the premature Out Of Memory error is likely to be fixed by future Wasmtime versions. This room for further improvement contrasts with the container overhead, which is at the state of the art of a decade of research in container technologies.

5.3.1. Dynamic configuration

WebAssembly is critically useful to enable arbitrary code execution on highly 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 mem-

ory 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 as yet 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 otherwise suited for code on devices susceptible to performance and energy efficiency.

Interpreting WebAssembly on microcontrollers offers an appealing alternative to other language runtimes, e.g. Lua, which are commonly used on embedded devices to support dynamic configuration [9]. The WebAssembly standard has many features that make it attractive for embedded devices [50]. First, WebAssembly is a platform-independent Intermediate Representation that can be generated from different source languages and run on many CPU architectures. Solving how to run WebAssembly on microcontrollers effectively allows opening the embedded world to the Continuum as an additional place of intelligent computing, rather than only as a mere data collector and dummy actuator. Furthermore, many broadly used language runtimes such as JavaScript, Lua, or Python cannot provide predictable execution. They may require excessive memory for a microcontroller, whereas Wasm requires no mandatory garbage collection and only a few runtime features around maintaining memory sandboxing. This lightweight-ness is a most valuable asset in an embedded adaptation.

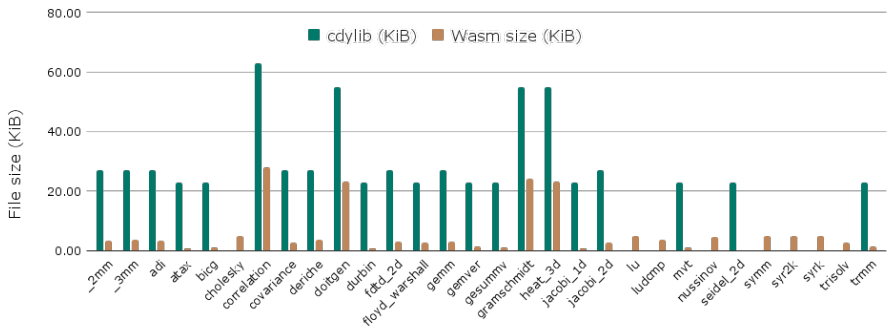

Figure 12: Comparison of Wasm size (KiB) and C dynamic library size (KiB).

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 be even less relevant and greatly favour the former, as containers package a whole operative system filesystem. Even the tiniest image base (Alpine Linux Mini

Root Filesystem) has an additional size of about 5.5MB uncompressed.

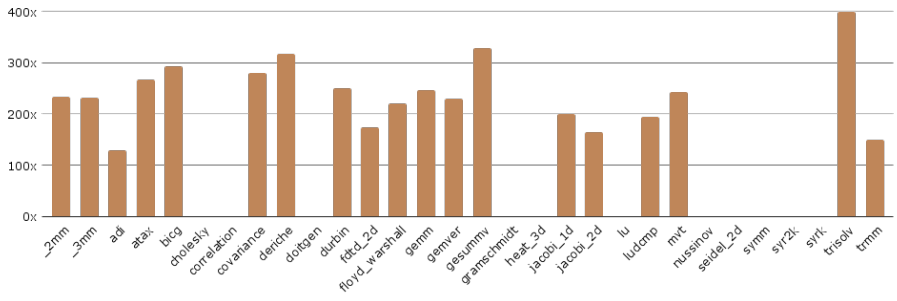

Figure 13: Comparison of Wasm interpreter performance and Rust native performance.

Figure 13 plots the slowdown of the Wasm interpreter executing Polybench benchmarks on the STM32F407 microcontroller against native Rust. Each Polybench benchmark has been run 15 times, following the methodology described in [51]. The results show a dramatic slowdown, with a factor of 100-400X. Such results dispel the notion of using Wasm interpreters on microcontrollers to support dynamic reconfiguration. However, it is fair to say 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; as such it is used to offer a deterministic sandboxed execution context running on Cloud servers. Accordingly, embedded execution performance is not paramount to it, unfortunately. Alternative interpreters, implemented in the C language, shows a much inferior execution penalty, in the order of 30-60x slower than native [50]. Thus, it is reasonable to believe that future efforts may allow reaching decent performance for Rust-built Wasm interpreters. Arguably, however, a 30x execution penalty can still seriously 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.

We have evaluated the heap overhead of interpreting Wasm on microcontrollers as an additional benchmark. 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

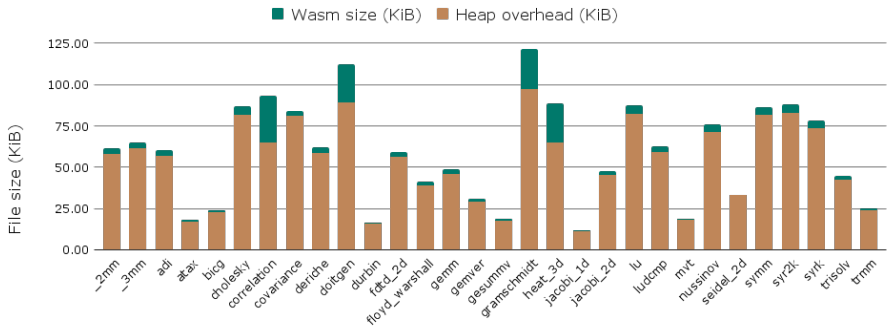

Figure 14: Comparison of Wasm size (KiB) and heap overhead (KiB).

also presents a memory-design issue. Wasm's pages are 64KiB by standard, 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 memory by 64KiB chunks, insufficiently granular for constrained embedded systems. Consequently, 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 dramatically increases with it. 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 infrastructure is programmed with C and C++. These two programming languages are chosen due to the low memory footprint and the high processing performance endowed by their very nimble runtime. Embedded systems have favoured the two languages because of the need for low-level control over the hardware. However, C and C++ do not excel at producing secure software, as evidenced by the many vulnerabilities reported against the software written in them [54].

Conversely, 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 uninitialised

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 integration with LLVM, the Rust compiler can transform the Intermediate Representation (IR) of the program 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. 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 highly constrained environments like microcontrollers. Later we leveraged the same language and developer experience while working on systems-level programming in Krustlet and cluster-level communication with Akri in Kubernetes. Lastly, we also implemented the protocol-bridging brokers and high-level web services in Rust. Despite the diversity of programming levels, the fast-paced Rust community is extremely rich in libraries for a variety of use cases. Admittedly, though, a good level of craftsmanship is still required as most open-source libraries are not battle-tested and present rough edges, like unimplemented yet critical features, which one has to develop independently. Additionally, embedded or low-level system programming may require using unstable language features, which hinders the adoption of Rust for critical, long-lived applications or even industry-wide adoption in production.

Figure 15 summarises the key technologies we employed in the reference architecture of the Continuum infrastructure.

We briefly also revisit our previous system for weather-based service as a 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 they agree to communicate following the REST architecture. Service nodes typically use REST over HTTP, while sensor nodes prefer CoAP;

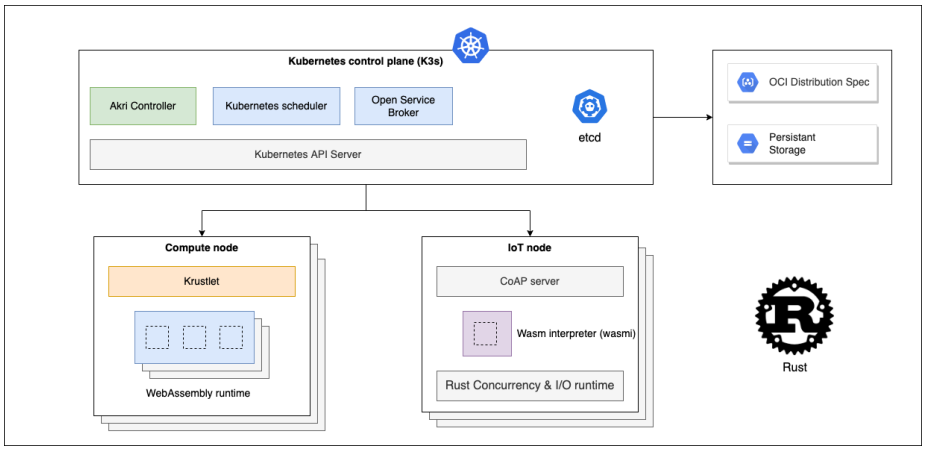

Figure 15: Technology baseline for the reference infrastructure architecture.

- 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 an electrical load forecasting service) and compile the application to Wasm. We successfully compiled a neural network inference model to WebAssembly. However, there is an underlying issue with implementing network servers as there is neither sufficient network API nor multi-threading support in the standard yet. On the one hand, 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.
On the other hand, 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 [55]). This limitation severely limits the workload capabilities of the server. Lately, the WebAssembly specifications have outlined a thread and atomics proposal intending to speed up multi-threaded applications. At this time of this writing, that proposal is still in the early stage, and it is implemented only in web browsers, behind an experimental flag.

6. Conclusion

In this paper we have presented a Continuum of Computing constituted by software and hardware resources provided as a service and delivered anywhere

the user is, independently of their respective location. In the intent of evaluating its viability for real-world industrial applications, we uncovered the difficult challenges that the research community will have to face to make the Continuum real.

For many of such challenges, we have shown that present-day technologies align well in principle with the envisioned needs of the Continuum, but they are still very far from sufficient maturity for industrial use. Projects like Akri for Kubernetes still are in their infancy and rated as explorative by some of the major players in the Cloud industry, notably Microsoft and Huawei. Akri, in particular, still has to solve the challenge of bringing Cloud capabilities to IoT devices in addition to merely exposing them as a service. Even relatively well-adopted standards like the Open Service Broker API are still incapable of solving common problems that different stakeholders are facing, such as managing the lifecycle of chains of service dependencies.

The experience reported in this paper allows us to conclude that many solutions needed by the Continuum are organically sprouting from many areas of software development like Cloud, Edge and Web, which is very encouraging. Notably, several independent Cloud providers have remarkably similar ideas about the concept of service platforms. However, each of them goes about solving issues like service dependency on their own, which is not very satisfactory. The Continuum is still a very novel idea and its realisation demands cutting-edge solutions from the just mentioned areas of software development. Problems like IoT orchestration, service composition and multi-platform virtualisation are prominent hot challenges. Naturally, each of them has attracted interest in the last few years and nascent solutions have emerged, but we have not observed any governance over the evolution of these areas as the Continuum will need it.

The two ambits of technology that we deem most worthy of future effort are (1) ways to extend current orchestrator capabilities beyond Cloud boundaries and (2) bringing the WebAssembly virtualisation to a mature level for usage both in the high-powered Cloud machines and in constrained Edge devices. As things stand today, our empirical results show that WebAssembly is nearly exclusively suited for pure compute functions. The lack of multi-threading and a mature network interface severely limit the space of real-world applications that wasm can serve. Likewise, the results we obtained from execution benchmarks run on microcontrollers discourage the idea of using Wasm interpreters on resource-constrained nodes as it would be necessary to incorporate the Edge in the Continuum for real. On the bright side, we reckon that these problems should be surmountable given sufficient attention, time and effort by the developers' community. On this account, we anticipate IoT/service orchestration and WebAssembly to receive significant attention in the coming years.

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] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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

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

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

[17] 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.

[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] MITRE, Cwe - common weakness enumeration, https://cwe.mitre.org/, 2022. Accessed: 2022-03-05.

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