The Continuum of Computing: a seamless ubiquitous service-oriented platform to unite the Edge to the Cloud

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

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

Abstract

Dramatic improvements in mobile connectivity combined with the productivity of Web-level programming have caused the Internet of the past to turn into a ubiquitous platform where a multitude of web services are developed, deployed and employed at various locations. The resulting multitude of heterogeneous computing nodes, ranging from clusters of compute nodes via consumer devices to industrial sensors and actuators, compels envisioning a seamless integration of the Cloud and the Internet of Things into a single Continuum.

Although already spoken of in some vision statements, the Continuum still is a very novel notion for technology. Its concrete realisation demands cutting-edge solutions from various ambits of Cloud, Edge and Web software development. This paper presents a high-level reference architecture for the Continuum accompanied with a Proof of Concept implementation of an infrastructure layer for it, which helps gauge the distance between various state-of-the-art technologies and their integration, with the proposed vision.

The findings presented in this work show that several technology solutions have organically sprouted in recent years, which very well fit the Continuum, suggesting natural convergence towards it. However, a substantial gap in terms of viability still remains, whose bridging calls for serious community effort.

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.

Similarly, dramatic improvements in mobile connectivity occurred in the last two decades in terms of ubiquity, reliability, and affordability, have allowed anyone to access the Internet from anywhere and at any time. 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. As such "things" comprise a multitude of heterogeneous devices ranging from consumer devices, like mobile phones and wearables, to industrial sensors and actuators [3], this step of evolution has given exponential growth of the Internet of Things (IoT), which spans smart transportation systems, grids [4], and even cities [5].

In this arrangement, a more decentralised solution is required. The Continuum is a collective infrastructure where data processing may take place *dynamically* where it is deemed most convenient under any of the criteria of interest (latency, privacy, energy, etc.). The concept enables 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 [6]. The Cloud and the Edge can both benefit from forming the Continuum together, allowing Cloud-alike virtualized 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 in literature 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 aggiungere riferimenti recenti. The Continuum concept goes beyond merely connecting network nodes to allow computation to happen at *predetermined* locations in the computing space.

Such vision can bring novelty to a variety of application areas, from managing extreme events (e.g. environmental monitoring [7]) to optimising everyday processes (e.g. manufacturing [3]) and improving life quality (e.g. healthcare [8] and smart cities [9]). Figure 1 attempts to capture said vision pictorially.

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.

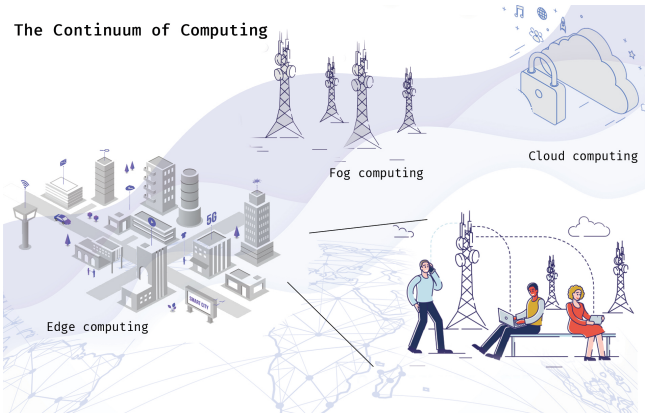

Figure 1: Pictorial view of the Continuum of Computing comprising the Edge, Fog and Cloud computing. The Edge of the network is where physical reality begins, whereas Fog devices operate as the bridge between the physical and the digital worlds (Edge and Cloud respectively). Fog computing denotes computational resources situated at a few, usually one, hop away from end users.

Contribution. The concept of Continuum is not completely novel, as some authors have already described a similar vision during the last decade [10, 6]. No matter how attractive this vision may be, however, only a few authors have carried out to date, to the best of our knowledge, efforts to explore the implementation of technologies for integrating the Cloud and Edge device into a seamless system, [11]. Our work aims to provide a foundation to fill such gap.

Our research comprises a proposition of a reference architecture for the research community (Section 2), an in-depth analysis of the Continuum's problem space (Section 3) and a a Proof of Concept (PoC) implementation that adapts and integrates various state-of-the-art technologies (Section 5). We adapted both well-known tools developed for the Cloud and technologies still at an early stage of development, in order to gauge the distance between the industry and the Continuum vision. The result of our work and the utilised tools are all open-sourced and available for any future work that aims to move forward the implementation of a Continuum system.

To the best of our knowledge, such adaptation and integration of open-source tools has only been assumed to be possible by the research community [12] and wrongly deemed as straightforward in our opinion.

We first explored the feasibility of adapting mature Cloud tools like the Kubernetes [13] orchestrator to span over the whole Cloud-Edge continuum. Notably, we explored the discoverability of Edge nodes in a Kubernetes cluster and the ubiquitous interoperability of web services. We also thoroughly experimented with the integration of the nascent sandboxing standard WebAssembly [14] to bring container-like virtualisation and portability on both comparatively powerful machines and constrained devices.

In our findings, we noticed that many of today's technologies fit rather well,

in principle, in our vision of the Continuum. We take this as a sign of convergence in the organic trends of evolution of areas 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 [13] are still figuring out how to align their Cloud-centric interface to modern use cases like Edge computing. On the other hand, nascent solutions like WebAssembly [14] are very promising on paper in terms of designs and features but still in a state of infancy in terms of practicality. More in-depth conclusions are outlined in Section 7.

2. A system-level view of the Continuum

2.1. 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, as depicted in Figure 2.

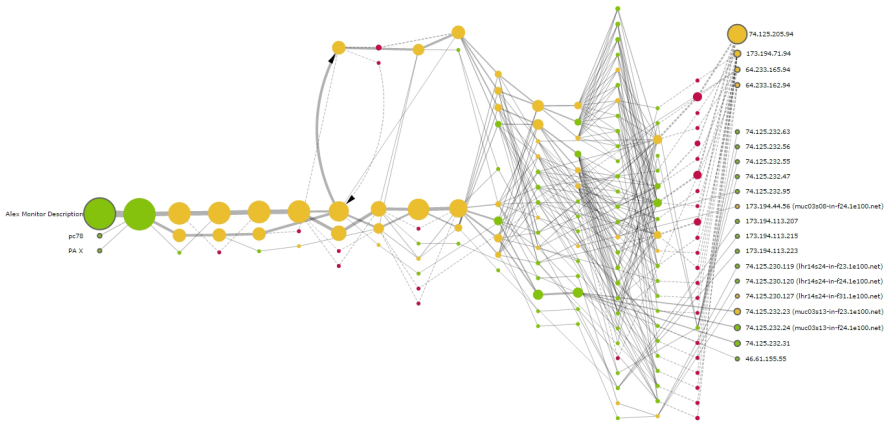

Figure 2: Traceroute virtualisation of an IP packet reaching google.com. The left green nodes are the source nodes, while packets travel across to the extreme right to servers located in data centers. Source: [15].

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, the Continuum computing has the responsibility to determine the preferable trade-off between the two, leveraging resources across the whole path 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.

2.2. Cluster federation

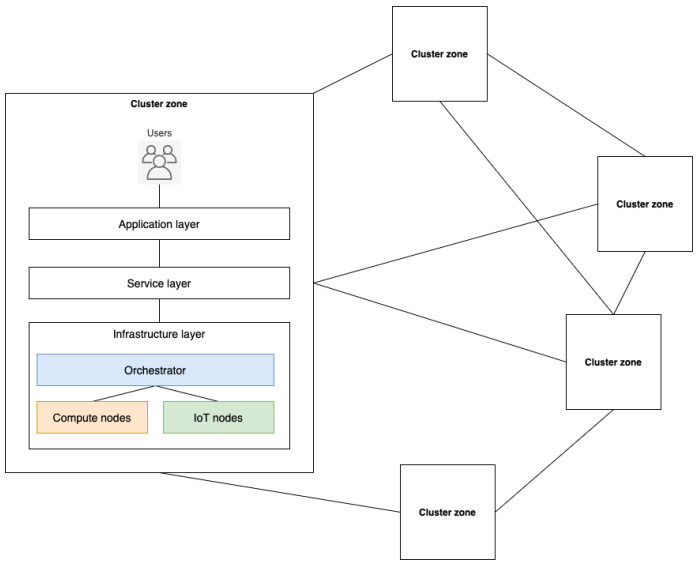

Figure 3: A high-level view of a federated set of cluster nodes.

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 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.

2.3. 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.

2.3.1. 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 the application requirements, e.g. latency, geographical constraints, availability or utilisation.

2.3.2. 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

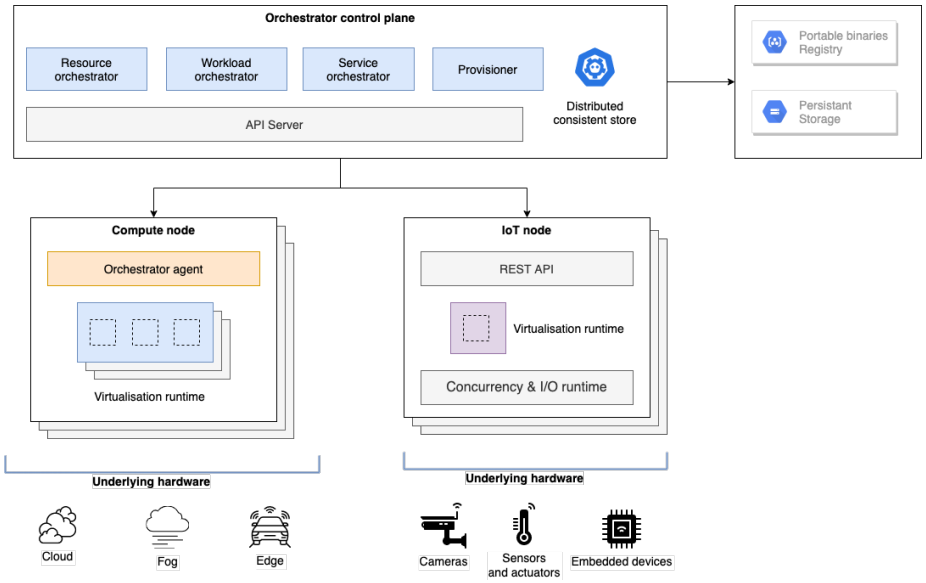

Figure 4: Reference architecture for the infrastructure.

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.

A critical requirement of the virtualization runtime on the Compute nodes is 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. This aspect is fundamental due to the extreme heterogeneity of the devices in the Continuum.

2.3.3. IoT nodes

IoT nodes are embedded devices that act as sensors or actuators, provided as services to the cluster (more on this to follow). 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 §4.2. Besides, the embedded devices support dynamic configuration by running arbitrary virtualisation modules in a lightweight runtime.

Besides the requirement of interoperability between Compute nodes, the IoT runtime must as well be compatible with the application format accepted by the Compute nodes, assuming the module size and the hardware requirements can be satisfied by the limited device. Such extended service interoperability enables greater flexibility and novelty in deciding where some aspects of IoT computing, such as controlling and preprocessing, happens.

Solving how to run arbitrary computation 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.

2.3.4. 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.

2.4. 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 both mainstream areas like flood warning [7] and novel fields like electrical load forecasting [16] and precision agriculture [17].

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. Gathering and preprocessing this data showcases how essential it is, for many services in the Continuum, that arbitrary code may execute safely and swiftly.

Additionally, 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, thus, an additional point in favour of sharing the data and computation on the Edge.

Finally, in a flood warning system, the telemetry stations acquire data (e.g. air humidity, soil moisture) from wireless sensors networks. Besides the opportunity of sharing this data with the above applications, the sensor networks also 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 implemented a Proof of Concept system based on the architecture proposed in Figure 5:

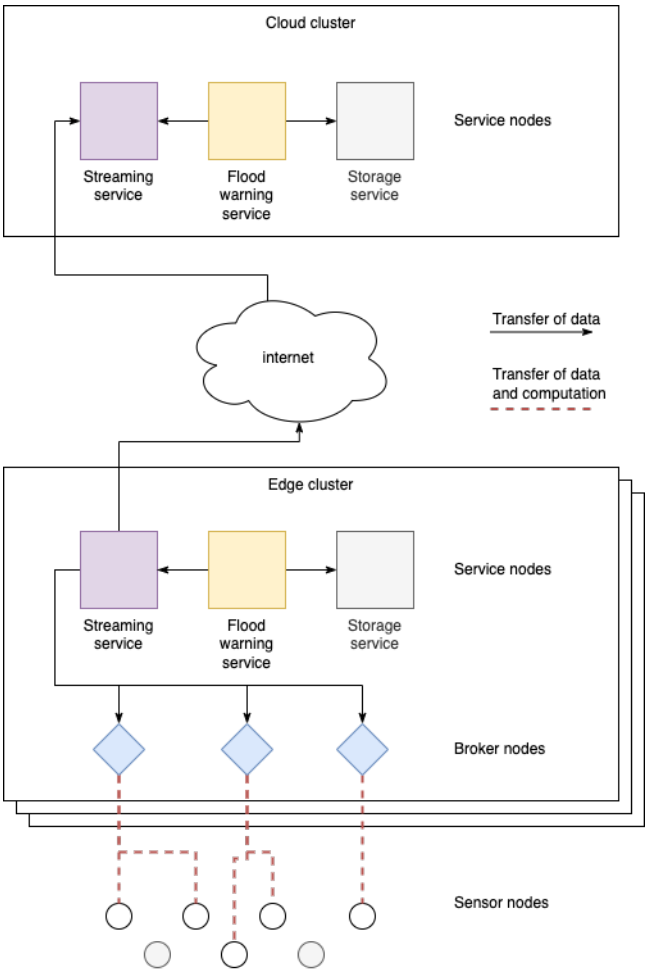

Figure 5: Architecture for the flood warning system.

- 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 internal application-specific 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

Table 1: Preliminary overview of the challenges and candidate technologies. WebAssembly is evidently a key enabling technologies in our opinion.

Challenge	**Technology**
Service orientation	RESTful web services Open Service Broker CoAP
Orchestration	Kubernetes, Akri
Virtualisation	WebAssembly
Dynamic configuration	WebAssembly
Interoperability	WebAssembly
Portability and Programmability	WebAssembly, Rust

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 prediction model benefits from more knowledge derived from multiple streams geographically distributed. A flood risk assessment model running in the Cloud can 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. All these examples of computing dynamism advocate that the Continuum shines compared to relatively static Cloud-only, Edge-only, or pre-defined Cloud+Edge architectures.

3. The challenges ahead

Several challenges lay ahead in the realisation of the Continuum infrastructure. Besides featuring extreme heterogeneity, current Edge technology most notably lacks support for service orientation, interoperability, orchestration, reliability, efficiency, availability, and security. We now briefly discuss each problem in isolation and for each we later propose a candidate technology. Figure 1 presents a preliminary overview.

Service orientation. We argue that service orientation is fundamental 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 [18].

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 [6].

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

First and foremost, there is a lack of vendor-neutral, trustworthy and widely accepted service intermediaries. Their availability is critical 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 [19].

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 [20], a Google technology for serialising structured data, in their service APIs. A plausible implementation of the Continuum should map high-level descriptions (e.g. 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 [18]. 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. Effective M2M communication is paramout in our vision of the Continuum, as binding a consumer to a particular service interface should entail minimal direct interaction with the provider's infrastructure.

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 [21].

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), net-

work 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. Additionally, it requires mobility handover and service migration at local and global scales Riferimento.

TODO: leggere Orchestration in fog computing: a comprehensive survey

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. [22] and [23]) argue the feasibility of container virtualisation applied to cheap low-powered devices, such as the Raspberry Pi [24].

Thanks to the underlying Docker image technology [25], 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 [26] 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. Besides, containers incur latency between hundreds of milliseconds and seconds [27], 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 [28], 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 [29] 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 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 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 [7] 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 [30]. are popular in the wireless sensor networking area, while OPC [31] 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 architectures, 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 [32]. 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.

4. Technology selection

In order to address the described problems in our Proof of Concept (PoC), we adapted and integrated a variety of technologies from the industry. The resulting selection include a mixture of both well-known mainstream tools and experimental tools still under initial development. The usage of existing tools is intentional to allow a realistic evaluation of the state-of-the-art to prove that there is an undeniable organic trend towards the Continuum paradigm. Our goal is also to provide a baseline technology selection for future works in the Continuum. For each candidate technology, we assess its maturity in terms of fitness to the goals of the Continuum and simplicity of adaption or integration. While many technologies look promising on paper, they may prove to be still in infancy in reality. To the best of our knowledge, we are the first to provide an extensive collection of tools publicly available for research and development in the Continuum.

4.1. 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) [33] 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 4.1) are emerging to extend the web to the Internet of Things and HTTP itself is undergoing revisions (e.g. HTTP/3 or QUIC [34]).

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 the sensor nodes in our PoC 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, IoT 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 to use a uniform interface across the Continuum: 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.

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.

We used these features to bring IoT nodes into the Continuum as any other service and to enable the coexistence of multiple equivalent services offered by different Cloud providers. We mapped provider-specific interfaces to uniform RESTful interfaces.

Open Service Broker. In our PoC, we realised a web-based service platform that implements the RESTful Open Service Broker (OSB) interface [35]. 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

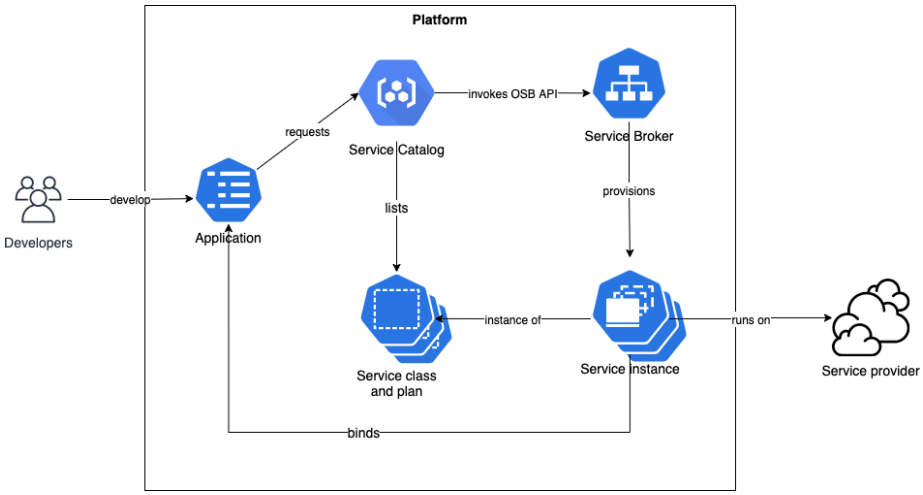

Figure 6: The Open Service Broker architecture.

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 architecture we devised for our use case (Figure 5). 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.

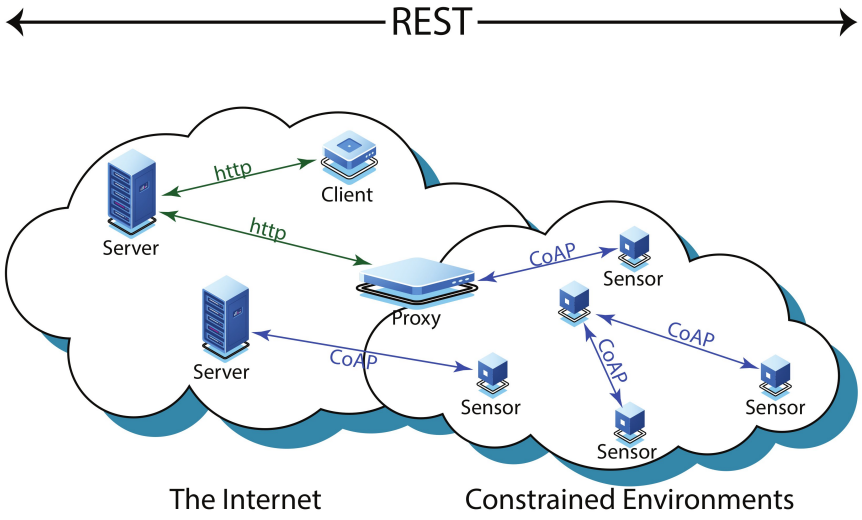

Figure 7: The REST architecture enhanced with CoAP. Source [36].

CoAP. To include IoT nodes in our REST architecture, we adopted CoAP [36], 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.

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 §4.2) 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 intermediary can discover CoAP resources and make them available at regular HTTP URIs, enabling web services in the Continuum to access CoAP servers transparently in the OSB service platform.

4.2. Orchestration

Kubernetes [13] 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 [37]. This extensibility allowed us to leverage a uniform virtualisation platform between the Compute nodes of the Continuum.

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 [38] was lack of multitenancy in the latter. Docker Swarm is a popular open-source orchestrator often cited for Edge orchestration (e.g. [23], and [39]) due to its simplicity. However, support for multitenancy is a must for our service platform.

Akri. To register the IoT devices on the Kubernetes cluster, we adopted Akri [40], a Microsoft 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.

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. We devised the broker as a web server that abstracts the actual communication between devices and applications behind the RESTful API described Previously.

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

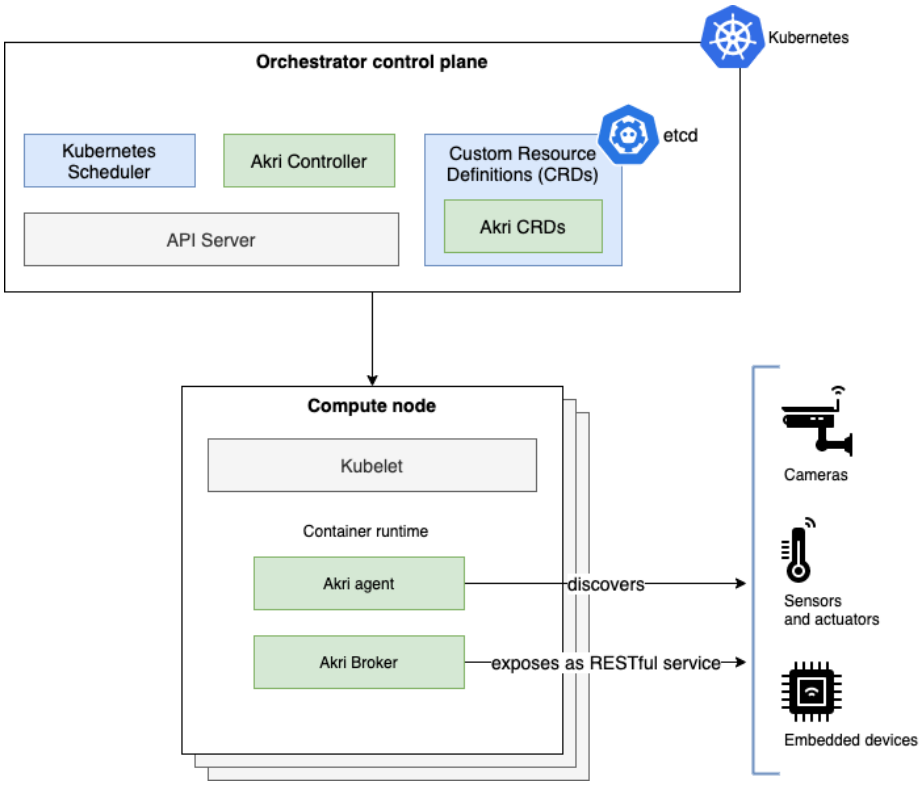

Figure 8: The Akri architecture can be divided into four main components: the agents, the controller, the brokers and the configuration. A configuration extends the Kubernetes API with new communication protocols and the related metadata, such as the protocol discovery parameters or the Docker image for the agent container. The Akri agent is a Pod responsible for discovering devices according to a communication protocol. It keeps track the device's state and keep the Akri controller updated with the status.

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.

4.3. 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.

We picked WebAssembly as the technology enabling virtualisation, interoperability and portability in the Continuum for two fundamental reasons. First, 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. Second, WebAssembly is advertised as safe *and* fast to execute. A program code cannot corrupt their execution environment, jump to arbitrary locations, or perform other undefined behaviour (which memory-safe languages such as Rust, riferimento, 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. 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. Performance wise, benchmarks of Wasm runtimes on modern browsers have shown a slowdown of approximately 10% compared to native execution, typically within 2x [14, 12].

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 [41], [42] and [43], especially for serverless computing.

4.3.1. Dynamic configuration

Another strong point of WebAssembly is enabling arbitrary code execution on highly constrained devices on the Continuum. The authors of eWASM [44] 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 [45]) 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 [46], they help dynamically update system code and debugging but are not yet mature in terms of 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 [7]. The WebAssembly standard has many features that make it attractive for embedded devices [44]. First, WebAssembly can be generated from different source languages and run on many CPU architectures, thus both on the Compute nodes and the IoT nodes of our proposed architecture. 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 9 summarises the key technologies we employed in the reference architecture of the Continuum infrastructure.

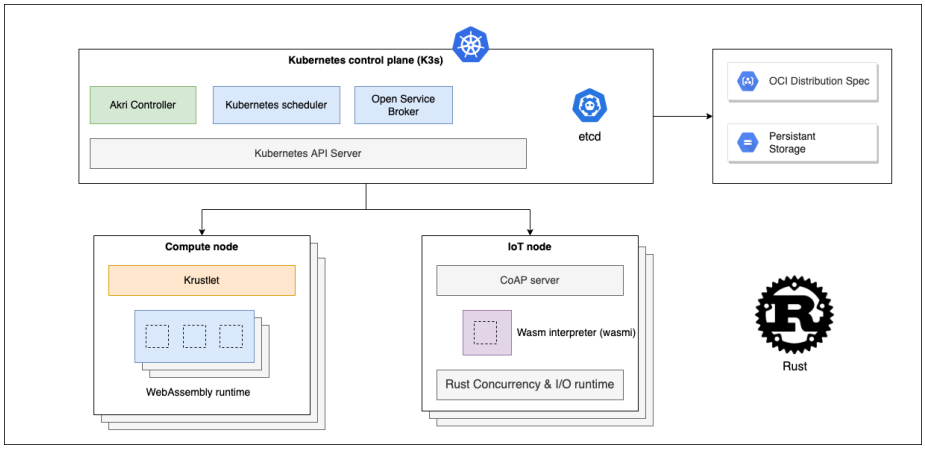

Figure 9: Technology baseline for the reference infrastructure architecture.

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;
- 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.

5. Evaluation

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.

We also have limited our work to Wasm-related benchmarks. The rationale is to provide benchmarks that can be adequately compared against in future works of the research community. Additionally, WebAssembly is the enabling technologies that we deem most promising and important for the Continuum among our the experimental ones of the presented technology selection.

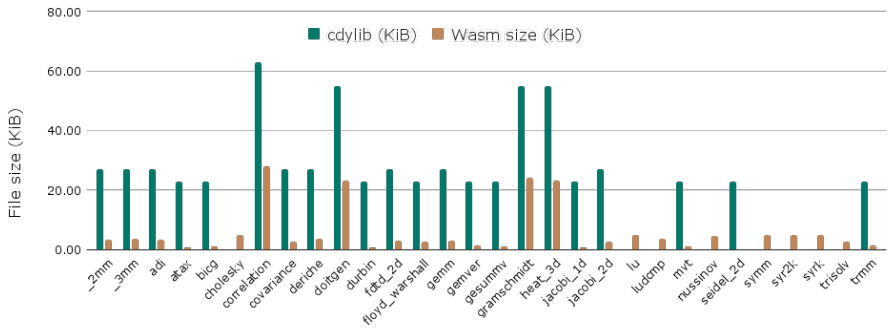

Figure 10: Comparison of Wasm size (KiB) and C dynamic library size (KiB).

5.1. Wasm for IoT devices

Figure 10 presents a comparison of the sizes of different Wasm binaries compiled from the Polybench [47] 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 from the same Rust source code and 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.

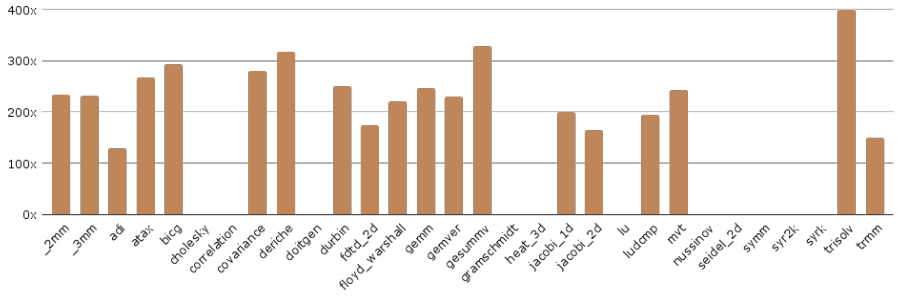

Figure 11: Comparison of Wasm interpreter performance and Rust native performance.

Figure 11 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 [46]. 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 [48], was adapted to work on embedded devices and was not designed for highly constrained devices. Alternative embeddable interpreters, implemented in the C language, shows a much inferior execution penalty, in the order of 30-60x slower than native [44]. Arguably, however, a 30x execution penalty can still seriously deter the usage of interpreters in microcontrollers.

Another crucial concern we have found in our work is that the heap overhead of using Wasm interpreters 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 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.

Finally, running Wasm on resource-constrained microcontrollers 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.

5.2. Wasm on the Cloud

We successfully integrated WebAssembly into Kubernetes with Wasm Pods running on Krustlet [49], while the container Pods are scheduled on K3s [50] 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 [45]).

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 [37].

At the time of writing, it has not been however possible to implement a portable web server (e.g. to act as an electrical load forecasting service) and compile the application to Wasm. 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 [51]). 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.

On the other hand, we have successfully compiled the weather-based flood prediction model to Wasm. The flood prediction model is a conventional Deep Neural Network constituted by three dense layers. The model has been trained on the Cloud using the traditional machine learning framework Keras [52].

As of the time of writing, TensorFlow [53] provides a WebAssembly backend for both the browser and Node.js [54]. However, the library itself cannot be easily compiled to WebAssembly yet in order to run the trained model. As a result, the model is saved to the alternative ONNX [55] standard format and executed by a Rust neural network inference library (*tract* [56]) that can read and run Tensorflow or ONNX models. The library in turn is compiled to Wasm as target.

6. Related Work

TBD
- [11] è solo per data processing

7. 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.

The experience reported in this paper allows us to conclude that many solutions needed by the Continuum are organically sprouting in the areas of Cloud and Edge development, which is very encouraging.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.

Our empirical results show that present WebAssembly is nearly exclusively suited for pure compute functions, which is too little for the Continuum. 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, however, we trust these problems should be overcome given sufficient time and effort by the developers' community. On this account, we anticipate WebAssembly to receive significant attention in the coming years.

[1] P. Mell, T. Grance, et al., The NIST definition of cloud computing, https://csrc.nist.gov/publications/detail/sp/800-145/final, 2011.

[2] Ericsson, Ericsson mobility report, https://www.ericsson.com/en/mobility-report, 2022. Accessed: 2022-11-19.

[3] 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.

[4] I. Mugarza, A. Amurrio, E. Azketa, E. Jacob, Dynamic software updates to enhance security and privacy in high availability energy management applications in smart cities, IEEE Access 7 (2019) 42269–42279.

[5] N. Mitton, S. Papavassiliou, A. Puliafito, K. S. Trivedi, Combining cloud and sensors in a smart city environment, 2012.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] J. Ménétrey, M. Pasin, P. Felber, V. Schiavoni, Webassembly as a common layer for the cloud-edge continuum, in: Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge, FRAME '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 3–8.

[13] The Linux Foundation, Kubernetes, https://kubernetes.io/, 2021. Accessed: 2022-11-19.

[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] dotcom monitor, Visual traceroute, https://www.dotcom-monitor.com/wiki/knowledge-base/visual-traceroute-graphical-tool/, 2021. Accessed: 2022-11-19.

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

[17] 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.

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

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

[20] Google, Protocol buffers, https://developers.google.com/protocol-buffers, 2021. Accessed: 2022-11-19.

[21] 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.

[22] 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.

[23] 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.

[24] Raspberry Pi, Products, https://www.raspberrypi.org/products/, 2021. Accessed: 2022-11-19.

[25] Docker, Docker image specification 1.0.0, https://github.com/moby/moby/blob/master/image/spec/v1.md, 2021. Accessed: 2022-11-19.

[26] 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).

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

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] R. Fielding, Representational state transfer (rest), https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm, 2000. Accessed: 2022-11-19.

[34] 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.

[35] Cloud Foundry, Open service broker, https://www.openservicebrokerapi.org/, 2016. Accessed: 2022-11-19.

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

[37] The Linux Foundation, containerd, https://containerd.io/, 2021. Accessed: 2022-11-19.

[38] Docker, Swarm mode overview, https://docs.docker.com/engine/swarm/, 2021. Accessed: 2022-11-19.

[39] 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.

[40] Deis Labs, Akri, https://github.com/deislabs/akri, 2021. Accessed: 2022-11-19.

[41] 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.

[42] 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.

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

[44] 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.

[45] Bytecode Alliance, wasmtime, https://github.com/bytecodealliance/wasmtime, 2021. Accessed: 2022-11-19.

[46] wasm3, wasm3 performance, https://github.com/wasm3/wasm3/blob/main/docs/Performance.md, 2021. Accessed: 2022-11-19.

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

[48] Parity, wasmi, https://github.com/paritytech/wasmi, 2021. Accessed: 2022-11-19.

[49] Deis Labs, Krustlet, https://github.com/deislabs/krustlet, 2021. Accessed: 2022-11-19.

[50] Rancher, k3s, https://k3s.io/, 2021. Accessed: 2022-11-19.

[51] Node.js, The node.js event loop, https://nodejs.dev/learn/the-nodejs-event-loop, 2021. Accessed: 2022-11-19.

[52] Keras, Keras, https://keras.io/, 2021. Accessed: 2022-11-19.

[53] Google, Tensorflow, https://www.tensorflow.org/, 2021. Accessed: 2022-11-19.

[54] Tensorflow, Introducing the webassembly backend for tensorflow.js, https://blog.tensorflow.org/2020/03/introducing-webassembly-backend-for-tensorflow-js.html, 2020. Accessed: 2022-11-19.

[55] T. L. Foundation, Onnx, https://onnx.ai/, 2021. Accessed: 2022-11-19.

[56] I. Sonos, tract, https://github.com/sonos/tract, 2021. Accessed: 2022-11-19.