The continuum of computing

Giovanni Jiayi Hu[a,c]

[a]*Department of Mathematics, University of Padua, Italy I-35121*

ARTICLE INFO

Keywords:
continuum of computing
edge computing
cloud computing
webassembly

ABSTRACT

Abstract

ORCID(s):

1. Introduction

The Internet has evolved significantly since its inception. It has grown into a ubiquitous platform for everyday services from just a simple communication layer for information sharing between researchers. Many changes and infrastructure trends drive this transformation.

First, during the past decade, the sharing of server and networking capabilities - known as the cloud computing paradigm - has become a reality, giving users and companies access to virtually unlimited amounts of storage and computing power. Nowadays, many constantly connected mobile devices, servers, and network components offer their virtualised capabilities to their users.

Second, as connectivity improved dramatically in affordability, bandwidth, reliability and reachability, people started bringing their devices with them and accessing the Internet anywhere and anytime. This trend, in turn, boosted the evolution of mobile computing and led to the emergence of richer client-side web applications. The uptake of 5G connectivity will bring an even more massive boost to this trend. By the end of 2026, Ericsson estimates that over 3.5 billion people, or 45 per cent of the world's population, will have a 5G coverage subscription [33].

Nowadays, we are amid the so-called Internet of Things (IoT), where everyday objects (things) are connected to the Internet and each other. These "things" comprise a multitude of heterogeneous devices ranging from consumer devices, such as mobile phones and wearables, to industrial sensors and actuators [22], smart transportation, grids [88], and cities [85]. There are two denominators of this diversity of devices. First, they operate at the leaf nodes of the Internet network. Second, they are "equipped with identifying, sensing, networking and processing capabilities that will allow them to communicate with one another and with other devices and services over the Internet to achieve some objective" [119].

Thus, it is natural that as consumers, we want our Internet-capable devices (e.g., mobile phones, thermostats, electric vehicle) to adapt seamlessly to our changing lives and expectations, regardless of location [14].

Besides, our capacity for collecting data is expanding dramatically. Nevertheless, our ability to manage, manipulate, and analyse this data to transform it into information and act upon it has not kept the pace [1]. Data sources and the size of the generated information are growing exponentially. They have outpaced the Internet ability to transport this data in a reliable and timely manner to the cloud. IoT applications demand a short response time, involve private data, and produce a large quantity of data which could be a heavy load for the network transport layer. Cloud computing and Internet infrastructure are not capable of supporting these applications and their exploding multitude.

In the author's vision, the computing will converge to an information-rich pervasive continuum that seamlessly combines this data and computing power to model, manage, control and make use of virtually any realisable sub-system of interest [1]. Use cases exist in diverse application areas, from managing extreme events (e.g. environmental monitoring [20]) to optimising everyday processes (e.g. manufacturing [22]) and improving life quality (e.g. healthcare [92] and smart cities [55]). Fig. 1 is a pictorial view of such vision..

A natural requisite of this vision is that computational capabilities are gradually being introduced at the network's edge, where devices are connected to the Internet and communicate with each other. This inter-communication constitutes in edge networks. New approaches that combine distributed services close to the data sources (i.e. edge nodes) with resources in the cloud and along the data path can establish a computing continuum and effectively process this data. Depending on the use case and service level requirements, user applications may require processing and storage locally, in the cloud, or somewhere in between. Fog computing [23] is the term coined by the research community to identify computational resources situated at a few, usually one, hop away from end-users. Fig. 1 represents the fog computing as base stations, to which users connect to in everyday activities.

This trend lays the groundwork for novel and ubiquitous services in a wide range of application domains. To cite Haller et al. [54], this thesis believes that we are close to "a world where physical objects are seamlessly integrated into the information network, and where the physical objects can become active participants in business processes. Services are available to interact with these 'smart objects' over the Internet, query their state and any information associated with them, taking into account security and privacy issues."

This thesis proposes a vision that follows naturally from the previous premises - the **continuum of computing** - a ubiquitous system where distributed resources and services on the whole computing continuum are dynamically aggregated on-demand to support different ranging services.

In this thesis's vision, there will be pervasive service platforms anywhere the user is and a multitude of services available over the Internet. The service providers will provide services, and consumers use them to accomplish any high-level application-specific goal. Services will be composed based on other existing services, leveraging the principle of reuse. These services' granularity will be very different, ranging from high-level business services to low-level sensor services provided by the Internet of Things.

It is fundamental that software and hardware provided as-a-service is first-class citizen, allowing such services to be dynamically placed by decoupling them from their location. The continuum must also facilitate the elastic provisioning of virtualised end-to-end service delivery infrastructures. Virtual capabilities are leased, dynamically configured and scaled as a function of user demand and service requirements. End-to-end network configuration is a primary example of a virtual capability that needs to be scaled elastically according to the execution's dynamics. The continuum must behave as a fluid, which continuously adapts its shapes to fit the surroundings, as first described in [75]. In Fig. 1, the three spaces of computing (cloud, fog, and edge) are all

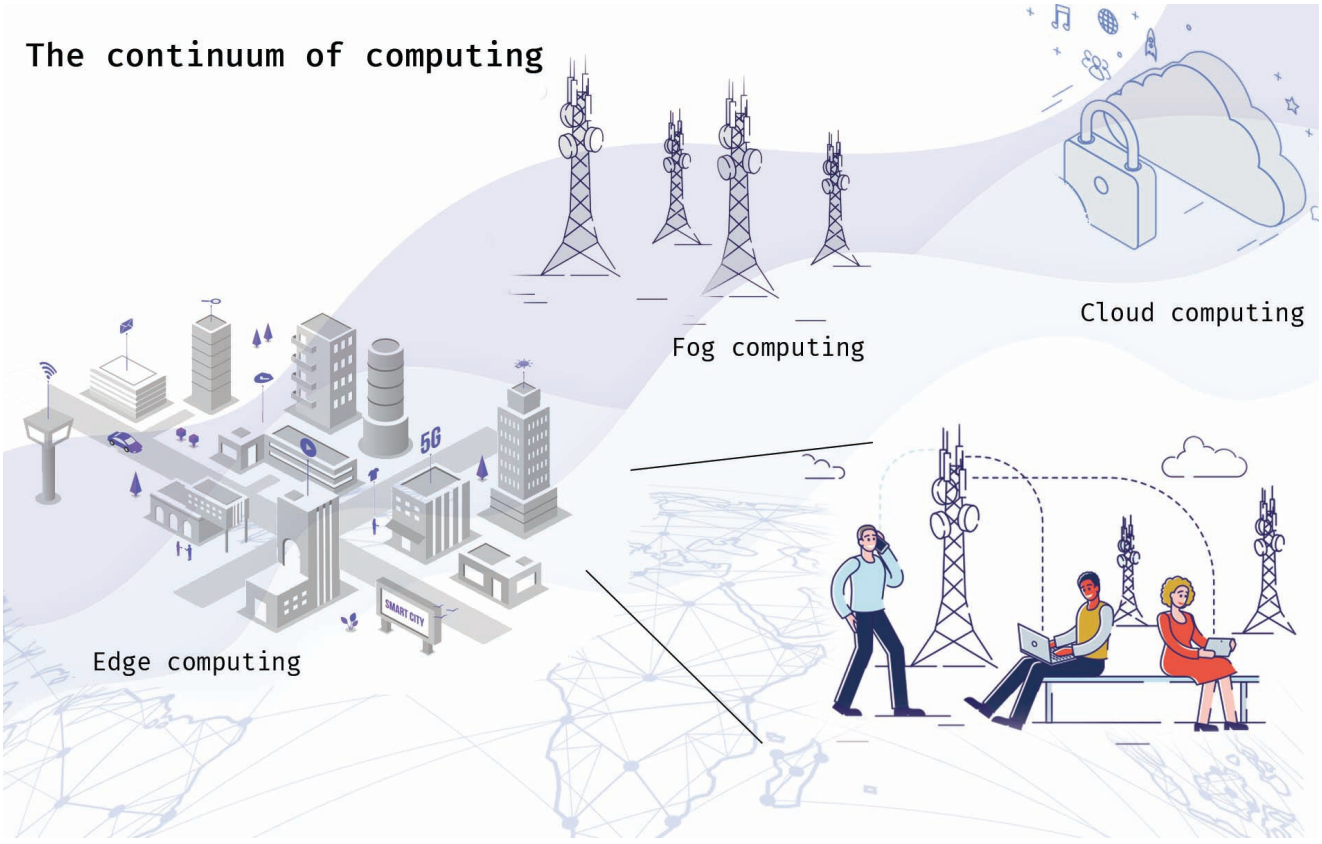

Figure 1: Pictorial view of the continuum of computing

touched by the fluid computing continuum, where computation can freely flow like a water droplet. This freedom of movement is an evolution to the common practice of merely connecting the network nodes to allow computation to happen at pre-determined locations in the computing space.

However, this vision goes beyond the traditional elasticity of clouds. In addition to computational, network and storage resources, the managed capabilities include end-to-end network configuration, high-level service and device functionalities.

The notion of "the continuum of computing" is not entirely novel and has been anticipated before. The authors of [75] envision a Fluid Internet, which "seamlessly provisions virtualised infrastructure capabilities, adapting the delivery substrate to the dynamic requirements of services and users, much like a fluid adapting to fit its surroundings". The researchers in [1] present the notion of computing in the continuum as well, as "a fluid ecosystem where distributed resources and services are programmatically aggregated on-demand to support emerging data-driven application workflows". Similarly, the authors of [14] define the continuum of computing and seek to develop approaches that include the entire computing continuum as a collective whole.

The vision of continuum would not be feasible without the previous groundwork done in the research areas of cloud computing and IoT. These two topics have gained extreme popularity in the last decade. The number of papers in both areas has been steadily increasing since 2008. More recent works like [19] focus on integrating the cloud and IoT but keeping the two as distinct spaces where the latter sends data and offloads computation to the former. The few publications mentioned in the previous paragraph are the only ones to foresee a continuum of computing encompassing the whole Internet as a substrate for ubiquitous services. Unfortunately, however, to the best of the author's knowledge, very few efforts have been carried out on developing platforms to make this vision possible and optimised.

The remaining parts of this thesis are organised as follows. Chapter 2 discusses the present trends which lead to the continuum and presents the challenges of this vision. Chapter 4 shows the problem space addressed by this thesis and the technology baseline. Chapter 5 contains an evaluation of the architecture presented in the preceding section and, lastly, the thesis concludes in Chapter 6 with the final remarks.

2. The trend to decentralisation

In the past decade and now more than ever, cloud computing has provided users with the potential to perform computing tasks utilising resource physically distant. A popular definition of cloud computing [81] is "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can

be rapidly provisioned and released with minimal management effort or service provider interaction."

The cloud architecture can be split into four layers: datacentre (hardware), infrastructure, platform, and application [125]. The resulting model is divided into three types of service: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Accordingly, to the developers' needs, IaaS can be chosen if one wishes to customise every aspect of how an application gets deployed and executed, leaving the responsibility for provisioning the hardware or virtual machines to the cloud provider. Otherwise, in PaaS and SaaS, the developers have no control over the infrastructure but can access prepackaged components or complete applications, respectively. Thanks to such flexibility in the service layer, the cloud computing paradigm has been highly successful. By 2023, Gartner predicts that 40% of all enterprise workloads will be deployed in cloud infrastructure and platform services, up from 20% in 2020 [45].

Conventional single provider infrastructures hosting cloud services offer undoubtedly many benefits (e.g. affordability, scalability, and efficiency) but not without challenges. A large data centre's energy consumption is high to keep it operational, and like any other centralised computing model, the issues resulting from sudden interruption would be adverse. The downtime cost of a single IT incident can range from a median of $600,000$\$ to a value of $2,400,000$\$ [58].

Another issue of cloud-only solutions is that the required data must be transferred and stored in separate places from the source. Data centres are often geographically distant from the application users, notably when the data is generated at edge locations. However, the private exchange of sensitive or personal data is critical for applications. For instance, in healthcare, to ensure patient's privacy, data cannot be stored in the public cloud.

Besides, the evolution of the Internet of Things is having a significant impact on cloud computing generally, and it is exposing the limitations of the latter. The number of connected devices increases exponentially with estimations of dozens of billions of "things" going live in the coming years [44].

As a consequence of connecting these "things" to the Internet, large volumes of data are being generated at unprecedented volumes, variety and velocity. This data is currently transferred and stored in the cloud in a centralised architecture. Data transfer, especially in these volumes, is costly in networking pricing (e.g. ingress traffic to the cloud and network transportation) and retards computational performance.

Thus, it becomes evident that traditional data processing methods where data is collected at the edge and processed in a central instance will not suffice. A more decentralised solution is required where data processing could take place before transfer and storage. The key is to reduce the number of messages and the amount of data transmitted throughout the continuum so that data should be processed locally or only where it makes sense.

With these premises, in this thesis's vision, we shall apply the cloud computing definition as mentioned above to the Internet of Things and the network along the data path (fog and edge). The rationale for achieving cloud-like capabilities in the continuum is two-fold. First, it offers on-demand virtual capabilities regarding storage, memory and processing units that augment IoT devices and components with limited computation capabilities due to form factors. Note, however, that fog and edge nodes cannot scale "infinitely" as cloud data centres do.

Second, scaling fog and edge infrastructure is arduous and time-consuming, even more as in cloud infrastructure, due to additional factors such as heterogeneity, network unreliability and difficulty to predict capacity needs in advance. Underprovisioning means potentially losing business, while overprovisioning means wasting money on unused infrastructure [90]. Moreover, user demand at the edge can be very often spiky [90], meaning that companies would need to provision for anomalous peaks like gatherings and events, investing in significant infrastructure that sits underutilised most of the time. The overprovisioning also has an environmental cost when underutilised infrastructure consumes significant amounts of power.

The cloud computing model applied to the fog and edge as services is attractive since it allows businesses to decrease the required capital expenditure and frees them from the need to invest in infrastructure. Businesses can rent resources according to their needs and only pay for the usage [19]. Moreover, it reduces operating costs, as service providers do not have to provision capacities according to maximum peak load. Resources are released when service demand is low.

On the other hand, the cloud can benefit from the fog and edge by extending its scope to reach real-world things in a more distributed and dynamic manner and delivering new services in a large number of real-world scenarios. Such an extension will impact future application development, where this new flow of information gathering, processing, and transmission will generate new challenges (described in Chapter §2).

2.1. Decentralization

As stated before, highly distributed networks are the most effective architecture for the continuum, particularly as services become more complex and more bandwidth-hungry. Although often referred to as a single entity, the Internet is actually composed of a variety of different networks. The net result is that content generated at the edge must travel over multiple networks to reach its centrally hosted data centre.

Unfortunately, inter-network data communication is neither an efficient nor reliable operation and can be adversely affected by a number of factors.

The peering points, depicted nodes in Fig. 9, are where networks exchange traffic and 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 [90]. The economic incentive flows in at the first mile (cloud data centres) and at the last mile (IoT), with

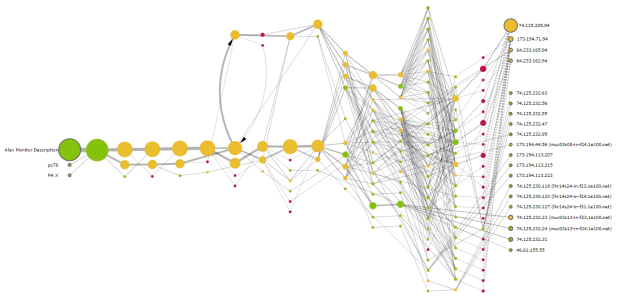

Figure 2: Traceroute virtualisation of an IP packet reaching google.com . The left green nodes are the source nodes and travel throught to the extreme right, where servers are located in datacentres. Source [87].

however very little interest to invest in the middle network composed of peering points. These points thus become the cause for packet loss and increase latency.

Across the Internet, outages are happening all the time, caused by a wide variety of reasons such as cable cuts, misconfigured routers, DDoS attacks, power outages, or even earthquakes and other natural disasters. While failures vary in scope, large-scale occurrences are not uncommon [113].

When dealing with a pervasive system such as the continuum, it is therefore no longer sufficient to simply have enough server and network bandwidth resources. One must consider the throughput of the entire path from IoT devices to data centres to end-users. The bottleneck is not likely to be at just the extreme ends of the path. It could be at a peering point, as mentioned, or due to the network latency between server and device.

Autonomous vehicles are an evident example of these issues. One Gigabyte of data will be generated by autonomous cars every second, and they require real-time processing for the vehicle to make correct decisions [105]. Such a situation can quickly challenge the network bandwidth and reliability to support many vehicles in one area. Moreover, if all the data were to be sent to the cloud for processing, the response time would be too long. In such cases, the data needs to be processed at the edge for shorter response time, more efficient processing and lesser network pressure.

Secondly, in many other cases like environmental monitoring, most of the end nodes in IoT (e.g. sensors and actuators) are energy-constrained things, so offloading some computing tasks to the edge could be more energy efficient. Even mobile phones benefit significantly from reduced energy consumption, especially for tasks like Mobile Augmented Reality [12].

For these reasons, alternate models such as fog computing and edge computing (shortly presented) have emerged in recent years. Such computing models arise from the general idea of processing the information at the nearest spot. A fog and edge-based platform, with servers anywhere the end device, can achieve the scale needed as well. Each location supports higher orders of throughput, low response times, and higher energy efficiency.

However, information processing at the nearest physical node may not always be a good option. If the task requires long computation, as in big-data analysis, it will be better to offload to the more capable but distant node till the cloud at the opposite extreme point. A request's latency is the sum of two components: the computing latency and the transmission latency. High computing latency can outweigh transmission efficiency. Therefore, edge computing has the responsibility to determine the ideal tradeoff between computing latency and transmission latency, leveraging resources on the whole continuum to achieve the best optimisation.

Similar attention must be dedicated to the energy issue. The battery is the most precious resource for things at the edge of the network, but the wireless communication module is usually very energy-hungry [105]. For a given workload, it is not always the case that the most energy-efficient solution is to offload the whole workload to the edge rather than compute locally. The key is the tradeoff between computation energy consumption and transmission energy consumption. Offloading to another node is preferable only if the latter overhead is more negligible than computing locally.

It shall be clear by now that the vision of a computing continuum is not that one form of computing (e.g. edge computing) will supplant another (e.g. cloud computing), but that we must try to harness the entire computing space as a whole. Some trending computing models relevant to the decentralisation above are now presented.

2.1.1. Fog Computing

Fog computing [23] is a decentralised computing infrastructure that is used mainly as a complement to cloud computing. It leverages the compute resources at the edge network and brings the computational processing closer to the data source by offloading workload to edge nodes from cloud data centres. The network nodes near the edge providing these resources are called fog nodes, and they are usually a single hop away from the end-users. An example of such nodes is the base stations depicted in the bottom-right illustration in Fig. 1.

Overall, any device with computing, storage and network connectivity can constitute a fog node. Examples include switches and routers inside buildings (pictured in the smart city in Fig. 1), industrial controllers, embedded servers and video surveillance cameras [61]. In recent years, the spread of fog computing has been facilitated by the availability of suitable hardware in small, affordable, low-power computers (e.g. Raspberry Pi [98]). Improvements in virtualisation technologies that enable slicing resources between different users to provide isolated environments (see Section §3.3) have also helped.

The Cloudlet [104] architecture is one of the first approaches that offer an example of fog computing with virtual machine (VM) techniques. Its computing power is much less than a conventional cloud infrastructure as cloudlets are composed of less powerful processors and are significantly smaller in size.

2.1.2. Mobile Edge Computing

Edge computing [105] denotes any computing and network resource that is located only at the extreme edge of the network. The rationale of edge computing is that computing should happen in the proximity of data sources, thus ideally *at* the data source, eliminating the costly data transfer to a remote data centre. The result significantly improves Quality of Service (QoS) as, similar to fog computing, and the end-user earns considerable network latency reduction and bandwidth consumption.

When the end-users are mobile devices, edge computing is referred to as mobile edge computing (MEC) [13] as well. In contrast to the broader term edge computing, MEC focuses on co-locating computing and storage resources at base cellular networks stations. Being co-located at base stations, MEC servers' computing and storage resources are available close to mobile users like smartphones and smart vehicles and can support mobility between cellular cells. MEC is seen as a promising approach to increase the quality of experience in cellular networks and a natural direction for the evolution to 5G networks [121].

2.1.3. Serverless Computing

Serverless computing [62] involves building, running and providing applications and services without considering the server-side. "Serverless" does not mean that there is no server usage, but rather that the application developers focus on the application itself rather than what happens on the physical infrastructure.

Serverless computing is synonymous with Function-as-a-Service (FaaS) and event-based programming as underlying provisioning and instantiation of the resources will happen only when necessary and not all the time. Notably, provisioning happens on the fly when an event triggers the execution of a function. In particular, FaaS triggers a server's infrastructural provisioning only when a function is requested, then executes the expected operations and frees the resources above as soon as they are deemed to be not necessary.

The major benefits of this model for cloud consumers are increased scalability and applications' infrastructure independence, and lower costs. Similarly, cloud vendors also have strong incentives for services to be built on serverless architectures as opposed to following a fixed-price model for long-running VMs. VM deployments tend to be very static as users deploy them for long periods. In contrast, the FaaS computing model tends to offer more variation in memory and CPU utilisation. As a result, the serverless computation gives a better opportunity to optimise resource usage and bill users for function invocation and network utilisation.

Recently, the leading cloud vendors (e.g. Amazon and Microsoft) are putting effort into making it easier to integrate edge functionality into their clouds, mainly by providing serverless computing and tight integration with their other cloud services. As an example, AWS Lambda@Edge [10] allows using serverless functions at the AWS edge location in response to CDN event to apply moderate computations. On the other hand, AWS Greengrass [8], and Azure IoT Edge [83] are commercial solutions specifically targeting serverless for IoT.

Recent works in the literature have argued the advantages of edge computing too. An instance, the authors of [12] present an augmented reality use case for mobile edge computing, in which computation and data-intensive tasks are offloaded from the devices to serverless functions at the edge, outperforming the cloud alternative up to 80% in terms of throughput and latency.

3. The challenges ahead

After discussing the decentralisation trend, this section presents the challenges in implementing such a vision. As an example, the heterogeneity of the connected devices is immense. It can be seen from different perspectives, such as computing performance, storage and network requirements, communication protocols, energy consumption, to name just a few. Such diversity poses questions about handling the virtualisation of the underlying resources and the nodes' interoperability. Both challenges are explored in Chapter §3.3 and §3.5 respectively.

To enable the continuum, this thesis identifies the following challenges as important ones to be addressed. The list should not be considered exhaustive by any means, nor the most significant ones. Dynamic network configuration and distributed persistent storage are two examples of critical technical challenges, which the author has not explored yet in his thesis' work.

Besides being characterised by very high heterogeneity, the edge lacks important features such as service orientation, interoperability, orchestration, reliability, efficiency, availability, and security. Therefore, it is natural that many challenges in the continuum's realisation overlap with open research problems in edge computing. At the same time, other issues are originated, or at least exacerbated, by the desire for holistic integration of cloud, fog, and edge computing.

1. *Service orientation*: as stated in the introduction of this thesis, services must be treated as first-class citizens, allowing them to be dynamically placed and decoupling them from their location. Consumers should only be concerned with what they want to do and accomplish and providers with how that could be done and provided to the user;
2. *Orchestration*: advanced orchestration systems are required to support the resource management of heterogeneous devices and be adapted to many applications running on the edge connected devices;
3. *Virtualisation*: the system has to provision resources and provide access to heterogeneous IoT resources and hardware such as GPUs. Besides, today's common practice in the cloud assumes enumerated resources and predictable computing capabilities. However, in the continuum, the capabilities and numbers of components change dramatically over time and require virtualisation solutions to provide a reliable service. Lastly, edge computing must support multi-tenancy, and the

virtualisation techniques are likely to be asked to deliver execution environments even more hardened than in cloud centres;

4. *Dynamic configuration*: dynamic configuration on constrained devices would allow dynamically adapting to environmental context changes based on the application requirements. The benefits are increased accuracy in machine-learning predictions, lower response times and more flexible data transformation at the data source;
5. *Interoperability*: applications on a continuum should be able to amalgamate services and infrastructure from different locations seamlessly. Every node on the Internet should additionally expose a uniform interface to its resources and services;
6. *Portability and Programmability*: developers should be provided with the languages and tools to program applications and services to run in such a dynamic environment without having to reinvent themselves;
7. *Mobility*: efficient migration of each application and service has to be supported from platform to platform and to follow the users' movements in the network (roaming);
8. *Reliability*: services in the continuum could fail due to various reasons, notably at the edge. The establishment of reliable communication between nodes must be supported;
9. *Security and Privacy*: personal and sensitive user data are subjected to high risk while many users access Internet services from anywhere. Security and privacy matter even more when personal data may have to be stored closer to the users/devices to facilitate computing and processing on the edge or fog layer;
10. *Context awareness*: services need to be aware of different aspects of the environment they are acting in, whereas today's services are rarely context-sensitive;
11. *Energy efficiency*: obtaining energy efficiency in both data processing and transmission is an important open issue;

3.1. Service orientation

This thesis envisions service-oriented paradigms as necessary for organising and utilising distributed capabilities that may be under the control of different ownership domains. A Service-Oriented Architecture (SOA) provides "a uniform mean to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations" [79].

According to this model, the major components of a basic SOA and their possible interactions are constituted by a service provider that publishes its service interface (how other services access their functionalities) via a service registry where a service requester/consumer can find it and subsequently bind to the service provider [54].

The SOA reference model's central concept lies in services that provide access to capabilities to be exercised by following well-defined interfaces and a service contract with constraints and policies. This enables the loose coupling of services and minimisation of mutual dependencies.

Services are provided by the service providers and used by others, the service consumers, a role usually taken on by application developers. Services may be composed based on other existing services, thereby adhering to the principle of reuse. They are responsible only for the logic and information they encapsulate, uniformly described and publicly retrievable via service discovery mechanisms. This design is in net contrast to today's practice. We continue to build ad-hoc programs confined in single points of the continuum, defining individual device computing behaviours [14].

However, to enable the seamless and agile interoperation of services in the continuum, several challenges still exist concerning their organisation and implementation.

First of all, the lack of neutral, trustworthy and widely accepted service intermediaries on the Internet of today still prevents the establishment of a continuum described above. Service intermediaries are necessary to facilitate the efficient retrieval of services that match a given user need and provide service performance by monitoring service quality and availability. Unfortunately, so far, single cloud providers such as Amazon Web Services, Google Cloud Platform or Microsoft Azure have shown no interest in inter-cooperation [49].

Secondly, the lack of means of composing higher-order services from various services imposes additional challenges to the emergence of the continuum of computing. The lack of interoperability prevents services from being set up quickly and easily. Cloud providers and their respective systems adhere to different conventions with respect to interfaces and communication protocols. For instance, Google Cloud Platform services heavily use Protocol Buffers [47] in the services' APIs, a Google technology for serialising structured data. For this reason, a significant effort will be required to uniformly provide the service semantics and interfaces in a comprehensible way for both humans and machines. For example, the system would need to map high-level descriptions to vendor-specific implementations (e.g. "key-value store" would map to different AWS or Google Cloud products).

Lastly, services on the Internet of today can be considered mute and unresponsive. In the future, services are expected to become communicative and reactive to their respective environments, particularly in two respects [54]. Existing service interfaces (e.g., REST-based) are not yet designed to be interpreted and used by machines but rather by humans. This issue frequently prevents the systems from easily discovering and interacting with them. Methods are needed to improve both user-service interaction and machine-to-machine communication. Today's services are unfortunately clearly built with human interaction in mind. Fortunately, recent REST community efforts are trying to compensate for the lack of M2M communication [109].

3.1.1. Service-oriented architectures

The idea of service orientation can be partially traced back to the object-oriented programming (OOP) literature [31]. However, the evolution of objects into services, and

the relative comparisons, has to be treated carefully since the first focus on encapsulation and information hiding in a shared-memory scenario. In contrast, the second is built on the idea of independent deployment and asynchronous communication via well-defined interfaces. However, both paradigms share the common idea of componentisation.

The last decade has seen a further shift towards the concept of service first and the natural evolution to microservices afterwards. SOA has been introduced to harness distributed systems' complexity and integrate different software applications. In SOA, a service offers functionalities to other components, accessible via message passing. Services decouple their interfaces from their implementation.

This design is in striking contrast to monolithic architectures, where the application is composed of a single program, typically providing a user interface and data access through a database. However, monolithic architecture poses real challenges and difficulties when growing exponentially because of the need to scale up [31].

SOA is built around services designed to work in an orchestrated manner to modularise the system. It is more challenging to divide the application into multiple services, but it enables greater flexibility, extensibility and reusability of existing services for multiple use cases. The model's benefits are application modularity, and service reusability [78], which are both very important to enable a computing continuum. However, a major disadvantage is the complexity of orchestrating and monitoring all the services and the underlying infrastructure, especially when the project is complex and the components' amount is huge.

The first generation of SOA architectures defined daunting and nebulous requirements for services (e.g., discoverability and service contracts) [31], which hindered the adoption of the model. Microservices are the second iteration, introduced as a solution for the gaps in the SOA approach.

The microservices approach divides applications into more granular components by distributing them into small self-contained services. Each service implements and attends to separate business functions and capabilities to maintain independence from other services. They are mainly deployed in an automated manner, through a container (more on that in §3.3), and communication happens through REST APIs [36], therefore making the impact of the programming language insignificant.

In terms of the continuum, a SOA realised through the *as-a-service* model allows consumers to be only concerned with what they want to do and accomplish, and providers with how that could be done and provided to the user. A successful interface establishment between those two actors can lead to minimal direct consumer interaction with the provider's infrastructure, allowing complete control to the provider and no cost of ownership for the former.

Furthermore, this architecture results in a system where various service implementations should already exist, maybe provided in the same fashion of service marketplaces (e.g. Shopify App Store [107]). The consumer himself does not have to be an expert and may use high-level languages to specify requirements, constraints and workflow to achieve arbitrary goals. Even the infrastructure's physical resources should not be consumer-aware, and there may be several diverse implementations to meet specific service demands. These implementations can differ in hardware type or service level requirements and could be characterised by different price and performance attributes.

3.2. Orchestration

The transition to the continuum will require coordinating and scheduling the operation of a myriad of distributed service components, whose complexity makes the necessity of service orchestration paramount. Besides, it will also be a crucial feature for many IT organisations and DevOps adopters to speed the delivery of services, simplify optimisation, and reduce costs [90]. However, the continuum's orchestration is challenging due to the scale, heterogeneity, and diversity of resource types and the uncertainties of the underlying environments. The uncertainties arise from a number of factors, including resource capacity demand (e.g. bandwidth and memory), failures (e.g. failure of a network link), user access pattern (e.g. the number of users and location) and lifecycle activities of applications.

Another difficulty caused by the underlying resources' heterogeneities is providing proper pricing models that take into account locations, resource types, and sizes. Considering such variables would enable, for instance, to dynamically provision resources of different pricing models to the cluster to satisfy application-specific needs with minimum cost.

Generally speaking, orchestrating the services in the continuum is a considerable challenge. It encompasses different technologies from different fields, including, among others, wireless cellular networks, distributed systems, virtualisation, and platform management. It imposes new models of interaction among different heterogeneous clouds, requires mobility handover and migration of services at both local and global scale.

3.3. Virtualisation

The rapid pace of innovation in data centres and software platforms has transformed how companies build, deploy, and manage online applications and services. Until the last decade, basically, every application ran on its physical machine.

The high costs of buying and maintaining large numbers of machines, and the fact that each was often under-utilised, led to a great leap forward in virtualisation.

Virtual Machine Monitors (VMM) [101] were born at the end of the 1960s as a software abstraction layer that partitions a hardware platform into one or more virtual machines (VM) to run existing software unmodified. Nowadays, however, a VMM is more a solution for security and reliability. Functions like migration and security, which are difficult to achieve in modern operating systems, seem much better suited to implementation at the VMM layer. KVM (Kernel-based Virtual Machine) [69] is an example of VMM built into the Linux kernel that allows a host machine to run multiple, isolated virtual machines.

On the contrary, Linux Containers (LXC) [16] are an OS-level virtualisation method for running multiple isolated applications sharing an underlying Linux kernel. A container consists of one or more processes, with reduced privileges, having restricted visibility into kernel objects, and host resources.

With containers, applications share an OS. As a result, these deployments will be significantly smaller in size than VM deployments, making it possible to store a few thousand containers on a physical cloud host (versus a strictly limited number of VMs). Adopting containers can ideally lead to a future where the end-user can deploy services and applications on edge computing platforms on heterogeneous devices with minimal efforts. Several works in literature (e.g. [93] and [15]) have argued the feasibility of container virtualisation applied to cheap low-powered devices, namely the Raspberry Pies [98].

Containers provide resource isolation, self-contained packaging, anywhere deployment, and easiness of orchestration. All these features make containers a very compelling technology for the continuum.

To be precise, however, some of these characteristics are a merit of the underlying Docker image technology [28] rather than the container virtualisation per se. Images package applications with individual runtime stacks, making the resultant software independent from the host operating system. This self-containment makes it possible to run several instances of an application on different platforms. Many leading cloud providers like Amazon, Google, Microsoft, IBM and others use this technology for enabling PaaS and FaaS.

Existing serverless platforms such as Google Cloud Functions [46], Azure Functions [82] and AWS Lambda [9] isolate functions in ephemeral, stateless containers. However, the use of containers as an isolation mechanism causes latencies up to hundreds of ms or even 5 seconds [86], which are unacceptable for latency-sensitive services operating ad the edge. These platforms cache and reuse containers for multiple functions call within a given time window, notably 5 minutes, to achieve better efficiency. Similarly, some users send artificial requests to keep the containers alive. In the edge environment, the long-lived and over-provisioned containers can quickly exhaust the limited node resources and become impractical for serving many IoT devices. Therefore, supporting a high number of serverless functions while providing a low response time (say 10ms [32]) is one of the main performance challenges for resource-constrained edge computing nodes.

In addition to the latency matter, containers have a not negligible resource footprint and isolation issues. Containers offer relatively weak isolation to the point where containers are run in per-tenant virtual machines to achieve proper isolation. Unfortunately, virtual machines' resource requirements are too demanding for the edge or fog nodes. A lightweight yet robust isolation solution is, therefore, a hot research question.

Accordingly, recently lightweight isolation platforms have been introduced as a bridge between containers and full system virtualisation. Both Firecracker and gVisor [21] rely on host kernel functionality. Firecracker provides a narrower interface to the kernel by starting minimalistic guest VMs and providing full virtualisation, whereas gVisor has a wider interface. They both have low overhead in terms of memory footprint, in any case. For instance, Amazon Firecracker has a memory footprint of 5MB, and startup latency of about 125ms [7].

Lastly, it is worth mentioning that Unikernels [80] are another novel virtualisation technique. Unikernels comprise a minimal operating system and a single application, making them a natural alternative to containers. Unikernel-based serverless experiments yield at least a factor of 6 better latency and throughput than mainstream solutions [66]. Unikernel implementations lack the maturity required for production platforms, e.g. missing the needed tooling and a way for non-expert users to deploy custom images. This disadvantage makes porting containerised applications very challenging. Besides, container users can rely on a large ecosystem of tools and support to run unmodified existing applications.

As a concluding note, undeniably, the basic units of execution have continuously shrunk in the past years: from VMs to containers, Unikernels, and serverless functions (at least in a virtual sense, as serverless functions are currently built into vendor-specific container images behind the scenes). Such smaller execution units are dictated by the need for increased flexibility execution environment and strong economic incentives to increase resource utilisation. Consequently, applications are increasingly becoming comprised of distributed components, and the number of lines of code associated with each unit of execution has dramatically decreased.

3.4. Dynamic configuration

Edge systems and IoT nodes must promptly recognise the environmental context changes under which they are subject. Such changes are often critical to many applications like video analysis [61]. This failure is because the service controller running on the edge can validate the data's usefulness (e.g. video frame) only after receiving the devices. Meanwhile, the IoT device will continue to perform poorly (or even uselessly) until the controller makes adjustments.

Dynamic configuration on constrained devices would allow dynamically adapting to environmental context changes based on the application requirements. This goal is usually achieved by running an application-specific computation on the node itself. The experimental results in [61] show that adapting multiple IoT cameras to serve applications and dynamically reconfiguring the cameras based on application desiderata can significantly enhance object detection accuracy, computational cost, and response time.

At the same time, opening the edge devices to arbitrary code execution exposes the system to malicious intents, meaning that even a single buggy line of code can lead to system compromise, threatening the controlled physical assets and potentially human safety. Current software isolation stacks are complicated to use in trustworthy embedded systems as

the machine might lack the necessary hardware (e.g. Memory Protection Units), have insufficient memory or missing the required software (e.g. Linux kernel functions to operate containers).

Hardware support is often used to isolate non-embedded applications from each other, and the OS from applications [96]. Dual-mode protection enables the kernel code to be inaccessible from applications at the cost of requiring mode transitions. In addition, Memory Protection Units (MPUs) enable the partitioning of memory into subsets, each accessible by separate applications. On the other hand, MPUs require OS, hardware, and build-system support. Thus the adoption of hardware memory isolation has been limited in microcontrollers as they typically lack the former two requirements.

A further challenge of IoT devices' dynamic configuration is allowing isolated execution with an acceptable compromise in efficiency loss and energy consumption. A common memory-safe execution technique is adopting interpreted languages (notably Lua) that provide type safety and ensure that all memory accesses adhere to the proper data type. Thus, memory accesses are constrained only to memory provided by the language runtime, usually via an interpreter, and are prevented from corrupting the device. For instance, the authors of [20] have ported interpreters of high-level languages (Lua and Python) to C as a means to support dynamic reconfiguration of the internal logic in telemetry sensors.

As an alternative, Software Fault Isolation (SFI [114]) techniques provide memory safety instead by constraining loads and stores to a sandbox. When this data sandboxing is paired with control flow integrity (CFI), which ensures that execution can only follow paths intentionally generated by the compiler, the application's execution and memory accesses are constrained to the sandbox. Unfortunately, both the overheads of interpreting in the language runtime and performing the necessary memory checks in SFI are a concern, given the limited resources of microcontrollers.

3.5. Interoperability

There are and will be many different technologies for connecting and integrating all the things into the continuum. For example, ZigBee, IPv6 over Low-Power Wireless Area Networks (6LoWPAN), MQTT and CoAP [89] are all gaining popularity in the wireless sensor networking area, and OPC [50] is well accepted in factory automation. However, the technologies are too different from expecting any standard to be able to cover them all.

For this reason, dealing with heterogeneity when building the continuum's edge infrastructure is necessary. Standards are undoubtedly helpful but will be hardly achieved. The key is to separate the functionality from its technical implementation, thus asking for interoperability instead of standardisation. Service-oriented architectures are ideal for this since they encapsulate functionality in services with a common interface, abstracting from the underlying hardware and protocols.

Having an infrastructure that allows connecting and integrating a diverse set of technologies is not just a "necessary evil" but rather a strength since it offers two key benefits. First, it allows for applying different solutions to different applications. Depending on the application requirements, the best-fitting technology can be used.

Secondly, an infrastructure where diverse technologies can easily be integrated into will be more future-proof. Especially in edge computing and IoT, the technical developments are not complete, and we can expect that new technologies and protocols will arise. An infrastructure built with technology diversity in mind will allow interoperability with existing and already deployed devices and networks.

With that said, cloud platforms heterogeneity is a non-negligible concern as well. Cloud services typically come with proprietary interfaces, causing resource integration to be adequately customised based on specific providers. This issue can be exacerbated when services in the continuum can depend on multiple providers to provide the different resources an application may require or to improve application resilience in case of single-provider failure [49].

Overall, cloud, IoT services, and applications are nowadays typically conceived as isolated vertical solutions. All system components are tightly coupled to the specific application context or the cloud provider. The interoperability challenge involves several aspects, where solutions need to be investigated in terms of uniformly access to cloud services [49], programming interfaces, and means for coping with data diversity.

3.6. Portability and Programmability

In cloud computing, users program their code and deploy them on the cloud. The cloud provider is the one in charge to decide where the computing is conducted in a cloud. Users have only partial knowledge of how the application runs. This easiness is one of the benefits of cloud computing: the infrastructure is transparent to the user. Usually, the program is written in the programming language the developer is most familiar with and between a limited selection (e.g. JavaScript, Java, Python, .NET). The application is then compiled for a specific target platform (typically x86_64 GNU Linux) since the program only runs in the cloud.

As mentioned before in Section §3.3, cloud developers typically use containers spawned from images that include everything needed to run them. This package includes code, libraries, settings, and system tools. More importantly, these images can be constructed from minimal filesystem layers, and read-only layers can be shared notably the base OS layer. Images hence are lightweight and use considerably less space than VMs. They offer benefits to developers in terms of ease of deployment, testing, and composition. Notably, they can be used to capture a development environment known to work for each application revision and share this environment among a team of developers. Organisations are widely using containers to deploy their increasingly diverse workloads to the cloud datacentres.

However, in edge computing and by extension in the continuum, the nodes have diverse platforms. The diversity of

CPU architectures (namely x86_64, ARM32, ARM64, and RISC-V) give programmers a hard time compiling for the different platforms. The nodes' runtime differs from each other, and the programmers face considerable difficulties in writing a service that may be deployed in the continuum paradigm.

Docker images attempt to overcome this issue by defining multiple variants (usually referred to as tags in Docker terminology) of the same image to target multiple architectures. A single image may contain variants for different architectures or even for different operating systems, such as Windows. Most of the official images on Docker Hub [29] provide a variety of architectures. For example, official images usually support amd64, arm32v5, arm32v6, arm32v7, arm64v8, i386, ppc64le, and s390x. As an instance, when running an image on an x86_64 / amd64 machine, the x86_64 variant will be pulled and run. Despite this feature, a significant pain point is that developers are still required to configure and build their applications multiple times for each platform. Moreover, the absence of an OS on embedded devices or the limitation on resource capacity block any hope of running containers on such machines, mining the idea of containers as a unified solution for portability in the continuum (at least in their conventional form).

The theme of portability is often strongly connected to programmability. Making the proper technological choices (e.g. the programming language) is necessary to grant portable execution. To address the programmability in the context of edge computing, the authors of [105] propose the concept of computing stream defined as a serial of functions applied to the data along the data propagation path. Likewise, the work in [14] proposes to use a simple FaaS abstraction to program the continuum so that new algorithms and deep learning models can be pushed to appropriate locations (i.e., edge, fog, cloud, or HPC computing resources).

Serverless architecture naturally solves two critical problems for portability and programmability [120]. First, the serverless programming model dramatically reduces users or developers' burden in developing, deploying, and managing applications. There is no need to understand the complex underlying procedures and distributed system to run the applications. Second, the functions are flexible to run on either edge or cloud, which helps achieve the desired portability.

However, in this thesis's view, the serverless computing model can satisfy only a limited subset of services, notably those with event-driven and request-reply nature. However, a common type of applications is long-running services that require high availability and must handle latency-sensitive requests. Examples include user-facing service or industrial control loops. Another type of applications is batch jobs, which have a limited lifetime and are more tolerable toward performance fluctuations. Examples include scientific computations or MapReduce jobs [27]. Batch jobs are more suitable to be run in the cloud as they require high computational capacity. However, there are literature examples that argue the advantages of big-data analysis on the edge [55], notably to achieve faster response times.

3.7. Mobility

In the continuum, services can be reallocated to follow the end users' movements, and the data and state should be reallocated as well. Therefore, the collaboration issues (e.g., synchronisation, data/state migration) have to be addressed across multiple infrastructure layers.

When mobility is required, provisioning data and services also needs to be performed with high reactivity and reliability. For instance, in the context of smart mobility, vehicles are often on the move, and vehicular networking and communication are often intermittent, or unreliable [56].

Being co-located at base stations, MEC servers are available anywhere the user moves and can support roaming between cellular cells.

3.8. Reliability

When applications are deployed in resource-constrained environments, many challenges related to device failure or unreachability exist. Things at the edge of the network could fail due to various reasons, and they could additionally report failure to report data under unreliable conditions such as low battery level.

As mentioned before, various new communication protocols for IoT data collection have been proposed in the last years (e.g. CoAP, MQTT, AMPQ) [89]. These protocols serve well for the support of low energy and highly dynamic network condition. However, their connection reliability is not as good as conventional BlueTooth or WiFi. For instance, if both sensing data and communication are not reliable, the system must leverage multiple reference data sources and historical data records to provide a reliable service.

3.9. Security and Privacy

The integration of edge computing, fog computing and cloud computing will raise some new and unforeseen security issues. Unique and unstudied 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 [122].

Like the health record data, end-user data collected at the edge of the network should be stored at the edge, and the user should be able to control if service providers should use the data. However, as edge computing stores and processes data at the edge, the privacy-sensitive information associated with end-users could be exploited and be more vulnerable than within cloud servers.

Moreover, for applications like smart grids or sensor networks, an adversary could report false data, modify the other user data, tamper with their smart meter, or spoofing IP addresses, further disrupting sensor management's effectiveness IoT systems.

3.10. Context awareness

Today's web services are rarely context-sensitive. However, to support context-sensitive services at the edge, they need to be aware of different aspects of the environment they are acting in. Context services can provide such information to the application services that implement local control

loops and trigger specific actions based on context events. Such context services can be composed of lower-level services which deliver individual sensor readings.

On this matter, the MEC architecture provides the advantage of data locality, which lets a given application store the context data only regarding the devices within the region covered by the base station. Such an advantage is twofold: providing location-aware services is more accessible, and less data has to be persisted in each zone. For example, MEC servers in [12] can retrieve the machine learning features in an augmented reality scene, match them against a local database, and return the corresponding data (information about monuments, buildings and other points of interest) to the client application.

3.11. Energy efficiency

Obtaining energy efficiency in both data processing and transmission is an essential open issue. Several directions have been proposed for handling such issue: more efficient data transmission and compression technologies, data caching mechanisms for reusing collected data in time-tolerant applications, middlewares to improve availability and compress data in case of continuous and long-duration monitoring of data.

4. System design

Having illustrated the challenges of this thesis' vision, this chapter presents the issues mentioned above tackled by the author's work and a technology baseline proposition.

4.1. Addressed challenges

This thesis attempts to provide some groundwork on how to address the following challenges within the list discussed in Section §2:

1. *Service orientation*: the web architecture is proposed as the substrate for the pervasive communication between services in the continuum, and the Representational State Transfer (REST) (§4.2) is the natural architectural style to follow so that services can communicate uniformly. Furthermore, the author presents Open Service Broker (§4.3) platform to allow independent vendors and developers to provide services running on popular service orchestrators;
2. *Orchestration*: Kubernetes (§4.4) is proposed as an orchestrator for resource orchestration, workload orchestration and service orchestration of the continuum. On this matter, Kubernetes is extended to support the orchestration of IoT devices with an attempt to overcome the heterogeneity and unreliability of the underlying environment;
3. *Virtualisation*: WebAssembly (§4.6) is used as platform-agnostic virtualisation technology to provide lightweight sandboxing capabilities on every node of the continuum based on Software Fault Isolation and Control Flow Integrity. Importantly, WebAssembly is a portable bytecode that is a compilation target of potentially any programming language, both the ones that use garbage collection (e.g. Go, JavaScript, and Python) and those that do not (e.g. C, C++, and Rust);
4. *Dynamic configuration*: the combination of a small WebAssembly interpreter and the low-overhead binary format of the bytecode result in intriguing opportunities for dynamic configuration of IoT devices;
5. *Interoperability*: REST is a natural choice to support high-level interoperability between services as well. A contribution of this thesis is the implementation of the CoAP (§4.5) protocol to expose constrained devices' capabilities as REST resources. Such resources are then made available within the Kubernetes cluster, easily accessible as RESTful services. The Akri project (§4.4.1) is introduced to extend Kubernetes to support IoT devices. Its broker-based architecture offers a compelling solution to the heterogeneity of the Internet of Things. Similarly, the Open Service Broker allows a similar opportunity to work around the interoperability issue between cloud providers via a broker-based architecture;
6. *Portability and Programmability*: the thesis suggests using recent efforts in extending and standardising Docker images to support the distribution of arbitrary binary files, notably WebAssembly applications instead of container filesystems. This solution significantly improves the integration with existing development tools and lowers the entry barrier for developing applications in the continuum. The Rust (§4.7) programming language is then argued as the most sensible choice for providers to implement the infrastructure and developers to realise their applications, especially because of the tight integration to the aforementioned WebAssembly technology;

Due to the ample problem space, mobility, reliability, security, privacy, context awareness, and energy efficiency are not addressed in this thesis and are left as open research questions. Besides, by any mean, it is also not in the intentions of this thesis to be assertive in the technical solutions proposed here. The author proposes a technological ground for future work and later realises a proof of concept (POC) to argue the continuum's viability with today's technologies.

4.2. Web services

Tim Berners-Lee's original vision for the web focused on documents and their inter-relationships, but it has become more and more a web of applications in the last decade. Nowadays, the web has become the world's most successful vendor-independent application platform.

The web is a loosely coupled architecture and applications today depend on the web architecture, using HTTP to access information and perform updates. The most dominant architectural style is Representational State Transfer (REST) [36] that makes information available as resources identified by URIs. Applications communicate by exchanging representations of these resources using a transfer protocol, HTTP precisely. HTTP is the most popular application protocol

on the Internet and the pillar of web architecture. However, new communication protocols (e.g. CoAP in Section 4.5) are emerging to extend the web to the Internet of Things and HTTP itself is undergoing revisions (e.g. HTTP/3 or QUIC [74]).

The most prominent characteristics of REST that makes it relevant as technology to enabling the continuum are:

- Resource orientation: resources are an information abstraction that allows servers to make any information or service available, identified via Uniform Resource Identifiers (URIs). The REST architecture allows the server to own the original state of a resource, and the client negotiates and accesses a representation of it. The representation negotiation is suitable for interoperability, caching, proxying, and redirecting requests and responses. Such features enable seamless interoperation of services in the continuum through proxies, scalable architectures and better availability. Besides, under the REST architecture, web resources often advertise links to other resources creating a distributed web and resulting in an even more scalable and flexible architecture;

- Uniform interface: clients access these server-controlled resources in a request-response fashion using a small uniform set of methods with different semantics (GET, PUT, POST, and DELETE). This well-defined, limited set of methods makes REST significantly different from Remote Produce Calls (RPC). 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. Such characteristic has been fundamental for the Internet-scale, multi-domain, scalable information system called web. It will be increasingly necessary as the latter expands its scope to continuum computing. RPC mechanisms, in contrast, are defined in terms of language APIs, not network-based applications [36];

- High-level interoperability: besides the representation negotiation, the REST architectural style enables further interoperability between RESTful protocols through proxies or, more generally, intermediaries that behave like a server to a client and play a client with respect to another server. REST intermediaries play well with the assumption that not every device must offer RESTful interfaces directly. In several cases, notably in the Internet of Things, it may not be possible to change the underlying communication protocol. However, intermediaries allow exposing the resources through a RESTful API nevertheless. The interactions behind that RESTful interface are invisible and may include highly specialized protocols for the specific implementation scenario (e.g. OPC UA or MQTT) [51];

- Communication protocol independence: REST needs to be based on a protocol different from HTTP for constrained environments. HTTP is a powerful and well-tried protocol, but it is relatively expensive for both binary overhead and network resource usage. Part of the problem is that HTTP has undergone more than a decade of organic growth, leading to considerable implementation baggage that overwhelms small devices [18]. Fortunately, the REST architectural style is not strictly tied to HTTP, and new communication protocols like CoAP (§4.5) can offer a very similar interface while guaranteeing more lightweight implementation, less demand for network bandwidth, lower latency and more energy efficiency among many features;

- Machine-to-Machine communication (M2M): communication protocols must be specifically designed for efficient M2M communications without introducing overhead in network load, delay, and data processing. In the M2M environments typical of IoT applications (and of the continuum by extension), devices must discover each other and their resources. Resource discovery is common on the web. One form of web discovery occurs when a user access a server's default resource (such as *index.html*), which often includes links to other web resources available on that or related servers. Machines can also perform web discovery if standardized interfaces and resource descriptions are available. New approaches from the IETF include the well-known resource path */.well-known/scheme* (RFC 5785) and the HTTP link header (RFC 5988). In the IoT, we are dealing with autonomous devices and embedded systems. Therefore the importance of uniform, interoperable resource discovery is much greater than on the current web;

- Stateless: REST requires requests from clients to be self-contained, in the sense that all information to serve the request must be part of the request. Statelessness helps applications to scale, as persisting state in a highly distributed system brings challenges like consistency and hinders support for user mobility in the continuum space.

From the previous points, it is clear that the design goals of RESTful web systems and the benefits for a decentralized and massive-scale service system align well with the field of pervasive computing. For these reasons, REST an ideal candidate to build a universal API for services and devices in the continuum, letting developers of web-based applications continue using their existing skills.

4.3. Open Service Broker

Application developers require services and managed ones, whose infrastructure is handled by a vendor for the whole service lifecycle, enable developers to concentrate on their application code rather than operating these services. On this matter, on-demand marketplace services can increase developer velocity and minimize time to deliver value to the

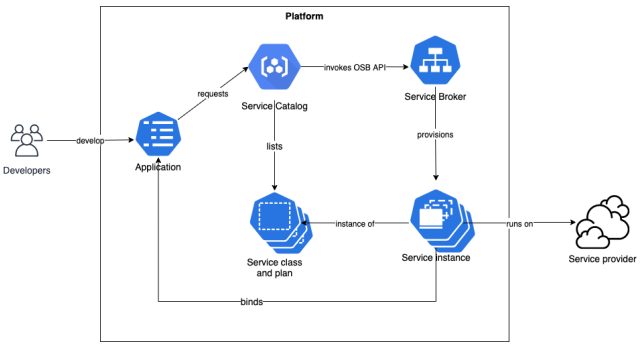

Figure 3: The Open Service Broker architecture

market. In the author's view, the continuum will benefit from a marketplace where vendors develop and add services based on user demand. This idea is not novel, and the most notable example is Cloud Foundry's Marketplace [42], now dismissed, unfortunately. A very akin one, still active and thriving, is the Shopify App Store [107] where developers can extend their e-commerce with third-party services. In the continuum's marketplace, providers control the access to the services and payment plans but allow developers to bring their own services to the catalogue. Over the years, a rich ecosystem of services will be developed and accessible via simple well-documented APIs (namely RESTful interfaces).

Contrary to this vision, cloud standards have failed to gain traction. Therefore, the need to find mechanisms for bridging the heterogeneity gap between platforms and enabling data integration is more relevant than ever. Most orchestration technologies working across administrative domains use a broker to orchestrate resources at different levels within a provider (e.g. the cloud and the edge network) and across providers [49]. As the number of cloud vendors is limited, it is possible to build adapter and brokering layers that align access to different clouds.

On this matter, several stakeholders (namely the Kubernetes community, OpenShift, IBM and Google) have shown interest and contributed to working on a standard Open Service Broker (OSB) API [41]. Components that implement the OSB REST endpoints are referred to as service brokers and can be hosted anywhere the application platform can reach them. Service brokers offer a catalogue of services, payment plans and user-facing metadata. The main components of the OSB architecture are presented in Fig. 3.

However, the actual value of the broker API is in abstracting service-specific lifecycle operations from the platform. The service broker translates RESTful requests from the platform to service-specific operations such as creation, update, deletion and generation of 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 marketplace so that the final catalogue of services is the aggregate of all services. The platform is thus able to provide a consistent developer experience for application developers consuming these services.

4.4. Kubernetes

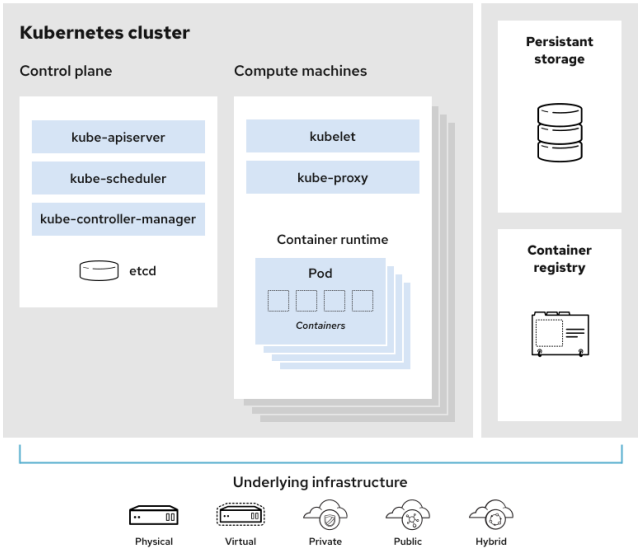

Figure 4: The Kubernetes architecture . Source [100].

Kubernetes [39] is an open-source framework designed to manage containerized workloads on clusters and originated from Google's experience with cloud services. Fig. 4 illustrates the main components of the architecture.

The basic building block in Kubernetes is a Pod. A Pod encapsulates one or more tightly coupled containers that are colocated and share the same set of resources. Pods also encapsulate storage resources, a network IP, and a set of options that govern how the Pod's container(s) should run. A Pod is designed to run a single instance of an application enabling horizontal scaling by replicating multiple Pods across the nodes. The amount of CPU, memory, and ephemeral storage a container needs can be specified when creating a Pod. This information can then be used to make decisions on Pod placement. These compute resources can be specified as both requests and limits.

Kubernetes allows for various container runtimes to be used 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] or Firecracker and gVisor mentioned before. Moreover, etcd [25] is an open-source highly available distributed key-value store and a fundamental part in supporting Kubernetes in managing the cluster nodes and jobs. Specifically, etcd is used to store all of the cluster's data and acts as the single source of truth for all of the framework's components.

Overall, Kubernetes is a highly mature system; it stemmed from 10 years of experience at Google. It is the leading container-based cluster management system with an extensive community-driven support and development base. It provides users with a wide range of options for managing their Pods and how they are scheduled, even allowing for pluggable customised schedulers to be easily integrated into

the system.

Besides the extensible scheduler, which makes Kubernetes already an intriguing choice as a potential orchestrator for the continuum, it also supports label-based constraints for the Pods' deployment. Developers can define their labels to specify identifying attributes of objects that are meaningful and relevant to them but that do not reflect the characteristics or semantics of the system directly. An example would allow specifying that an IoT device component must be reachable from the host or persistent storage available, e.g. a Solid State Drive. Nevertheless, more importantly, labels can be used to force the scheduler to colocate services that communicate a lot into the same availability zone, improving the latency drastically and paving the way for context-aware services. However, to realize the heterogeneity goals, it is required to filter unqualified resources continuously. New resource affinity models should be proposed to rank the resources when provisioning for different applications.

Finally, Kubernetes is also designed with multitenancy in mind, compared to Docker Swarm [30]. Docker Swarm is another popular open-source orchestrator often cited as the main competitor for edge orchestration due to its simplicity (e.g. [15], and [60]). Support for multitenancy is a must and well-aligned with their goal of executing heterogeneous services on a set of shared resources.

4.4.1. Akri

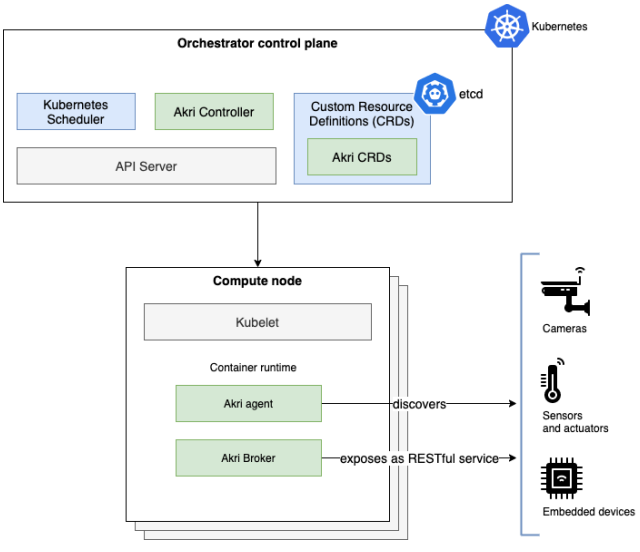

Figure 5: The Akri architecture

Akri [70] is an open-source project which extends the Kubernetes Device Plugins API [67] to allow visibility to IoT devices from applications running within the cluster. The Device Plugin API is designed initially to expose hardware resources attached to a Kubelet, such as GPUs or Solid State Drives, and it is still at the experimental stage. Akri stretched the design further so that it would be possible to implement the discovery of IoT devices, with support for the diversity of communication protocols and the ephemeral availabilities.

Akri's architecture, presented in Fig. 5, can be divided into four main components: the agents, the controller, the brokers and the configuration. A configuration is a Kubernetes Custom Resource Definition (CRD) to extend the Kubernetes API with new types of resources. Specifically, a configuration defines a communication protocol (e.g. OPC UA [50]) and the related metadata, such as the protocol discovery parameters or the Docker image to use as agent.

The Akri agent is a Pod responsible for discovering devices according to a communication protocol. It can be easily developed and deployed to the cluster to support new protocols in the system. The agent will track the state of the device and keep the Akri controller updated with the status. At the time of writing, the project has built-in support for ONVIF [91], udev [5] and OCP UA [50] discovery handlers, with an incoming proposal for CoAP [18] written by this thesis' author.

By 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.

Lastly, the Akri controller advertises discovered leaf devices to the Kubernetes resources manager, making them visible to any application that requires IoT devices to work. The controller 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, but Akri does not constraint. For example, Akri could be configured to deploy a broker that exposes a gRPC [26] interface and translate the requests to the underlying IoT communication protocol. In this thesis' view, the broker should be a web server that abstracts the actual communication between devices and applications behind a RESTful API. Such perspective is in line with the reasons presented before to choose the REST architecture as a uniform interface for services in the continuum. Akri should automatically find all the devices in the environment and make them available as web resources. As an instance, the agent discovers the devices regularly by scanning for CoAP (Section §4.5) resources.

Besides, the broker could offer local aggregates of device-level services. For example, it can offer a service that returns the combined temperature measurements of all the things connected to it at any given time.

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

The broker architecture has the advantage of fully decoupling the leaf node from the cluster workload. It only needs to send an update packet with a frequency short enough to

ensure data validity. On the other hand, the retrieved data's staleness will depend on the device's frequency of updates. Forwarding the HTTP request (adequately translated) has the advantage of always returning the most recent sensor reading when the request is processed. The cache mechanism cannot be applied for non-cacheable requests (e.g. HTTP POST) that must be sent to devices, such as turning on LEDs or changing the application state.

As many distributed monitoring applications are usually read-only during their operation (e.g. sensors collecting data), this architecture exhibits a greater scalability level. A potential goal is to enable new types of services where physical sensors can be shared with thousands of users with little, if any, impact on the latency and data staleness. A representative of such a case is tracking public transportation with sub-second accuracy [32].

4.5. CoAP

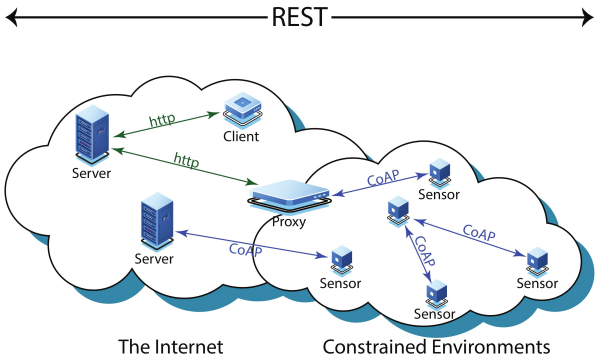

Figure 6: The REST architecture enhanced with CoAP . Source [18].

CoAP [18] (RFC 7252) is a web communication protocol for use with constrained nodes and constrained (e.g. low-power, lossy) networks. The protocol is designed for M2M applications and provides a client-server architecture between IoT nodes but supports built-in discovery of resources.

A central element of CoAP's reduced complexity is that, instead of TCP, it uses the UDP transport protocol and defines a very simple message layer for retransmitting lost packets. The rationale for UDP is that TCP carries significant overhead and can have suboptimal performance for links with high packet loss, both of which are common across the Internet [90] and especially on remote locations where sensors might be deployed. Besides, the HTTP protocol adds further burden, as the additionally multiple round trips required for the requests can quickly add up, affecting the device performance and energy efficiency.

CoAP uses a four-byte binary header within UDP packets, followed by a sequence of options, each up to two bytes. The protocol's specification also defines the usual four request methods: GET, PUT, POST, and DELETE. Similarly, response codes are patterned after the HTTP response codes.

The protocol is built upon key concepts of the web such as URIs and RESTful interaction [97]. As a result, CoAP easily interfaces with HTTP for integration with web services while meeting specialized IoT requirements such as multicast support, very low overhead and simplicity for constrained environments.

The URI format allows exposing device data as resources and the use of standard and specialized service endpoints. For instance, CoAP servers are encouraged to provide resource descriptions via the well-known URI */.well-known/core* to achieve resource discovery. Clients then access this description with a GET request on that URI, usually via an IPv4 or IPv6 broadcast message. The description format is based on the CoRE Link Format (RFC 6690), which is simple and easy to parse. Ease of parsing allows more efficient M2M discovery and inter-communication between the nodes themselves.

The CoAP protocol supports different resource representations, in line with the REST architecture's representation negotiation. The default format is textual for its convenience when reading and parsing. The binary format is efficient to communicate but requires external tools to make it readable by human users. XML is understandable and very well structured, but the size of its messages is significant, and it is much worse to parse compared to binary formats. Lastly, JSON is understandable, well structured and compact, but may still put an unnecessary parsing burden on the limited device. With all constrained devices, flash or memory consumption is one of the biggest problems, notably on devices where network connectivity already claims significant buffer memory.

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

Interoperability with the rest of the continuum can be achieved by following the REST architecture's proxy pattern, as depicted in Fig. 6. We can generally build intermediaries that speak CoAP on one side and HTTP on the other without encoding specific application knowledge. This flexibility allows deploying new applications without having to upgrade the intermediaries involved. On the other hand, an intermediary can perform the translation between CoAP and HTTP without posing further requirements either on the client or server. Because equivalent methods, response codes, and options are present in HTTP and CoAP protocols, the mapping between the two is straightforward. As a result, an intermediary (e.g. an Akri broker) can discover CoAP resources and make them available at regular HTTP URIs, enabling web services to access CoAP servers transparently.

However, the HTTP client-initiated interaction model may be unsuited for many event-based and streaming systems in the IoT. Data is sent asynchronously to the clients as soon as it is produced. CoAP uses the Observe approach (RFC 7641) to support pushing information from servers to clients

to overcome this issue. A client can indicate its interest in further updates from a resource by specifying the "Observe" option in a GET request. If the server accepts this option, the client becomes an observer of this resource and receives an asynchronous notification message each time it changes. This kind of communication, combined with an intermediary broker, allows streaming data updates via WebSocket (RFC 6455) and overcoming the client-pull interaction model of HTTP. The broker can also help achieve more reliable communication by transparently changing the underlying sensor in case of unavailability or by just avoiding closing the connection in case of temporary loss of connectivity.

4.6. WebAssembly

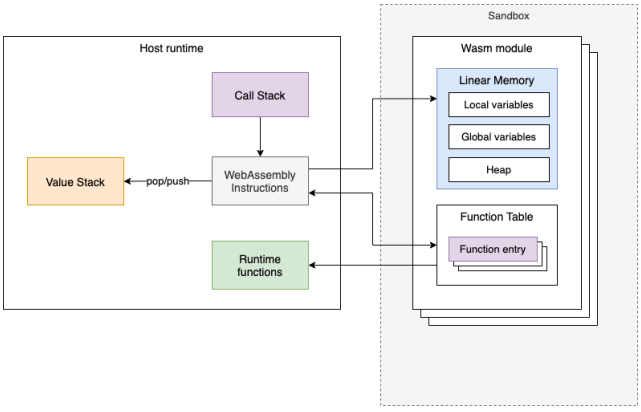

Figure 7: WebAssembly stack machine .

WebAssembly (Wasm) [52], 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. The project is developed by the World Wide Web Consortium (W3C) with support from all major web browser vendors (Mozilla, Google, Microsoft, and Apple), thus increasing the likelihood that it will avoid serving the purposes of a single entity (e.g. Java and .NET). 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, illustrated in Fig. 7. Even though WebAssembly technically is a binary code format, it can be presented as a language with syntax and structure. This design choice was intentional since it makes it easier to explain and understand without compromising compactness or decoding ease.

A Wasm binary takes the form of one or more modules. It contains definitions for functions, globals, tables, and memories. The computation is based on a stack machine: code for a function consists of instructions that manipulate values on an implicit operand stack, popping argument values and pushing result values. However, WebAssembly represents control flow differently from most stack machines. It does not offer simple jumps but instead provides Structured Control Flow (SCF) constructs more akin to a programming language. This design ensures by construction that control flow cannot contain arbitrary branches. The SCF allows WebAssembly code to be validated and compiled in a single pass as well. SCF can also be disassembled to a WebAssembly text format (.wat) that is easier to read and often overlooked but crucial human factor on the web.

A WebAssembly program's memory is a large array of bytes referred to as a linear memory or simply memory. All memory access is dynamically checked against the memory size, and out of bounds access results in a trap. Linear memory is disjoint from code space, the execution stack, and the engine's data structures. Each Wasm memory access addresses linear memory at an offset from the base, n, of the linear memory. Thus, some address virtualisation as an address b in the sandbox is located at $b+n$ in physical memory. The Wasm runtime is responsible for translating linear memory accesses and bounds-checking them to prevent accesses outside the sandbox.

Therefore, compiled programs cannot corrupt their execution environment, jump to arbitrary locations, or perform other undefined behaviour, especially when the original language is memory-safe (e.g. Rust §4.7). At worst, a buggy or exploited WebAssembly program can mess up the data in its memory.

This memory and state encapsulation are applied at the module level rather than at the application level, meaning that a module's memory and functions cannot leak information unless explicitly exported/returned. This granularity in sandboxing is extremely important as security incidents have increasingly exploited vulnerabilities in the dependency chain. Reuse of third-party software is pervasive in modern languages like JavaScript, Rust or Go. As an example, JavaScript (and consequently TypeScript [84]) strongly relies on its package manager NPM and the latter has been increasingly subject to security incidents in recent years [124].

The granular memory encapsulation means that even untrusted modules can be safely executed in the same address space as other code. Additionally, it allows a WebAssembly engine to be embedded into any other language runtime without violating memory safety and enabling programs with many independent instances to exist in the same process. These sandboxing features make WebAssembly a compelling technology upon which to implement a virtualisation stack for the continuum. On the web, the substrate of the continuum, code is fetched from untrusted sources, and it is vital that it can be safely executed.

Another critical safety feature of Wasm is the type system. Code must be validated before it can be executed safely. Validation rules for WebAssembly are defined succinctly as a type system. This type system is, by design, simple. It is designed to be efficiently checkable in a single linear pass to allow parallelisable binary decoding and compilation. Moreover, function pointers cannot be dereferenced directly. A call to a function pointer is translated into a function in a runtime table of valid entry points and types. The type of the

function is checked dynamically against the expected type in the said entry. The dynamic signature check protects the integrity of the execution environment. In case of a type mismatch or an out of bounds table access, a trap occurs.

With that being said, the design goals of WebAssembly strongly advocate for its suitability for this thesis:

- *Safe to execute*: protection for arbitrary code has traditionally been achieved by providing a managed language runtime that enforces memory safety, preventing programs from compromising user data or system state. However, managed language runtimes have traditionally not offered much for portable low-level code, such as C/C++ applications, that do not need garbage collection;
- *Fast to execute*: low-level code like that emitted by a C/C++ compiler is typically optimized ahead-of-time. On the contrary, managed runtimes and sandboxing techniques have typically imposed a steep performance overhead on low-level code. WebAssembly is, however, very competitive with native code, and benchmarks using Wasm runtimes on modern browsers have shown slowdown within 10% compared to native execution and almost always within 2x [52];
- *Language, hardware, and platform independency*: as mentioned before, the web spans not only many device classes but different machine architectures, operating systems, and browsers. Code targeting the web must therefore be independent of any underlying hardware or platform to allow applications to run across all software and hardware types with the same behaviour;
- *Deterministic and easy to reason about*: WebAssembly has been designed with formal semantics from the start for both execution and validation. Besides, the Wasm binary can be compiled into a .wat file to simplify learning and debugging;
- *Simple interoperability*: WebAssembly is similar to a virtual ISA in that it does not define how programs are loaded into the execution engine or how they perform I/O. The embedder (i.e. the host runtime) defines how modules are loaded, how imports and exports between modules are resolved, provides foreign functions to accomplish I/O, and specifies how traps are handled. It is possible, by design, to link multiple modules that different authors have created from likewise different source languages. However, as a low-level language, WebAssembly does not provide any built-in object model. It is up to developers to map their data types to numbers or memory. This design is supposed to provide maximum flexibility to developers, and unlike previous VMs like Java or .NET, it does not privilege any specific programming or object model while penalizing others. The downside of this design is that interoperability with object references is cumbersome. It involves exchanging references between the Wasm application and the host code or between modules that originated from different languages. A step in easing this issue is the recent introduction of Reference Types [117]. On the other hand, since WebAssembly is an abstraction over hardware, not over a programming language, a WebAssembly module may be compiled once and moved freely between different hardware architectures with no recompilation;
- *Compact*: the binary code is designed to be compact, especially compared to bytecode in text format. For this reason, it is recommended to distribute the binary code on the Internet code and using the text format only for learning and debugging. Code transmitted over the network has to be as compact as possible to reduce load times, save potentially expensive bandwidth, reduce memory usage on constrained network-attached devices and improve overall responsiveness;
- *Easy to validate and compile*: validation proceeds by checking on the fly while the incoming bytecodes arrive, with no intermediate representation (IR) being constructed. Benchmarks run on mainstream browsers in [52] prove that validation can be fast enough to be performed at a full network speed of 1Gib/s;
- *Streamable and parallelisable*: a Wasm runtime can minimize latency by starting streaming compilation as soon as function bodies arrive over the network. It can also parallelize the compilation of consecutive function bodies. For instance, each function body is preceded by its size so that a decoder can skip ahead and parallelize even its decoding.

Despite the name WebAssembly, there has been a significant effort in the last years in adopting Wasm for native execution, as it is a portable target for the compilation of various high-level languages. Wasm standard does not necessarily make browser-specific assumptions, and there has been substantial work to standardize the WebAssembly System Interface (WASI) to run Wasm outside the browser. To the best of the author's knowledge, the original design goals assumed a browser-based execution context. Nevertheless, such goals fit perfectly to the continuum's needs, where the web is the infrastructure upon which applications and services are deployed. On this matter, browsers are very much akin to an Operative System for client web applications, and WebAssembly is thus unsurprisingly being adopted on conventional OSes. Native execution and browser execution share many problems, like isolation, portability and interoperability, to name just a few.

At the time of writing, there are several Wasm runtimes for programs written in different languages capable of embedding Wasm applications. On the other hand, multiple compilers can compile languages to Wasm, as shown in Fig. 8. Notably, the Wasm back-end for LLVM [77] works for C, C++, and Rust. Commercial solutions have also been slowly but steadily gaining popularity. Cloudflare's Service

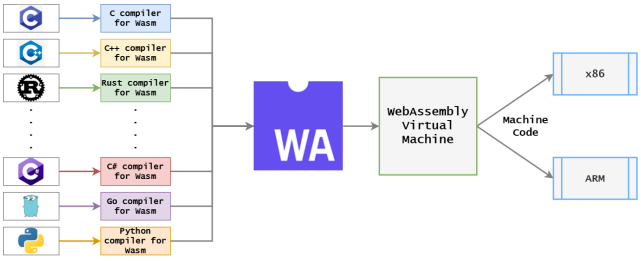

Figure 8: WebAssembly is a portable binary format .

Workers [24] began to offer support for creating and hosting serverless functions in WebAssembly. In March 2019, the edge-computing platform Fastly had announced an open-sourced Lucet [35] that provides a compiler and runtime to enable native execution of Wasm applications and can instantiate a Wasm module within $50\mu s$, with just a few kilobytes of memory overhead. Parity is also actively experimenting with writing smart contracts using Wasm [94].

WebAssembly is currently under experimentation as a new method for running portable applications without containers because of its design features. This method leverages its binary format a portable vehicle to inherent memory and execution safety guarantees via its language features and a runtime with strong sandboxing capabilities. The Wasm security model enables the execution of multiple untrusted modules in the same process, thereby providing significantly more lightweight isolation compared to VMs and containers for multi-tenant serverless execution [43].

This idea is still in its infancy, but there has been some interest in recent years, as shown by the works done in [53] and [106]. All these works focus primarily on serverless computing via WebAssembly, demonstrating the gain in popularity of such a computing model. As this thesis has stated before, however, if serverless were the only execution model, it would severely limit the variety of applications in the continuum.

4.6.1. WebAssembly for embedded microcontrollers

An emerging use case for WebAssembly is arbitrary code execution on microcontrollers, and, as a result, interpreters are gaining popularity on resource-limited devices. wasm3 [115] is a popular open-source interpreter capable of running on any system with at least 64KiB of storage and 10KiB of memory. The authors of eWASM [96] have also explored various mechanisms for memory bounds checking and have evaluated the trade-offs between processing efficiency and memory consumption. Just-In-Time compilers for WebAssembly exist (e.g. Wasmtime [4]) and receive more attention from the community, but their size and complexity make them unsuitable for microcontrollers.

Although WebAssembly interpreters can often be approximately 11x slower than native C [116], they help dynamically update system code and debugging but may not be applicable for code on devices susceptible to performance and energy efficiency.

In the author's view, interpreting WebAssembly on microcontrollers remains an intriguing technology nevertheless worth the research community's attention. They offer a persuasive alternative to other language runtimes, e.g. Lua, which are commonly used on embedded devices to support dynamic configuration [20]. The WebAssembly standard has many features that make it appealing for embedded devices [96].

- Broad support: there is a steadily increasingly ecosystem of vendors, tools, and languages providing WebAssembly support. With the ability to leverage WebAssembly, the embedded system community would benefit from a broader ecosystem;
- Portability: WebAssembly is a platform-independent Intermediate Representation that can be generated from different source languages and can run on many CPU architectures. Solving how to run WebAssembly on microcontrollers effectively would open the possibility to include the embedded world to the continuum as an additional place of intelligent computing, rather than only as a mere data collector and dummy actuator;
- No mandatory garbage collection: many broadly used language runtimes such as JavaScript, Lua, or python cannot provide predictable execution and may require excessive memory for a microcontroller;
- Lightweight runtime: WebAssembly mandates only a small number of runtime features around maintaining memory sandboxing. These light requirements help in an embedded adaptation.

4.7. Rust

As the continuum integrates compute support across the network, the attack surface greatly increases consequently. Security has to be a top priority during software development, yet most of the infrastructure is programmed using the C, C++ programming languages. These two programming languages are chosen due to the low overhead on memory and the high processing performance. Embedded systems have favoured the two languages because of the need for low-level control over the hardware. However, C and C++ are not particularly renowned for producing secure software, as evidenced by the large number of vulnerabilities reported against software written in them. Notably, the leading cause of security vulnerabilities are memory safety bugs like data races and buffer overflows. Even higher-level languages suffer from these kinds of safety issues. The Go language, upon which most modern distributed systems are built (namely Kubernetes), is not exempt from exploits based on data races [17].

On the other hand, Rust is a strongly-typed, compiled language that uses a lightweight runtime similar to C. Unlike many other modern languages, Rust is an attractive choice for predictable performance because it does not use a garbage collector. It provides strong memory safety guarantees by focusing on "zero-cost abstractions", meaning that safety checks

are done at compile-time and runtime checks (e.g. out-of-bounds access) have the minimum overhead and come with a predictable cost.

Not all operations can be verified to be memory safe at compile time. One example is indexing a slice, a partial view into an array. In this case, the indexing operation contains a runtime check to check if the index is within bounds. If the index is out of bounds, then the result is panic. A panicking condition can result in either aborting the whole program or just unwinding the stack of the thread that ran into the panic. The unwinding process walks up the stack freeing all live resources before terminating the thread, ensuring a clean memory state afterwards.

In the Rust community, there is a strong emphasis on building safe abstractions with almost zero runtime cost. Safe Rust code is guaranteed to be free of null or dangling pointer dereferences, invalid variable values (e.g. casts are checked), reads from uninitialized memory, unsafe mutations of shared data, and data races, among other misbehaviours. The borrow checker, the most innovative feature of the language compiler, runs as part of the compilation process and catches bugs like just mentioned misbehaviours. Security cannot be achieved without memory safety, so it is fundamental that the latter is a property of the language and not a developer's concern only.

In the author's view, Rust is a much recommended choice as the implementation language for the continuum's infrastructure. Rust is one of the few alternatives to C and C++ that can equally compete in performance, low-level control and cross-platform support. Its use cases are not limited to system development, however. Although initially advertised as a system language, Rust is nowadays described as a language to build *any* reliable and performant software. Use cases range from embedded systems and Operative Systems to web servers and user interfaces (e.g. via WebAssembly on the browser). Thanks to its vast community of open-source libraries and productivity tools, it can be a sensible alternative even for high-level services.

Its toolchain is backed by the LLVM modular compiler infrastructure, which Rust relies on for translating high-level code from languages such as C++ to its intermediate representation (IR). In addition to code translation, LLVM also provides several tools that aid in optimisation and dead code elimination. Thanks to the integration of LLVM, the Rust compiler rustc can transform this IR to generate WebAssembly binary code. The union of Rust and WebAssembly constitutes a powerful combination. Developers can write source code in Rust to achieve high productivity and efficient memory-safe applications. On the other hand, WebAssembly can contribute to the universal portability of such binaries without compiling or distributing multiple versions (e.g. Docker image versions).

4.7.1. RTIC

Rust is gaining popularity as a language for embedded systems as well. The language provides a rich standard library, which relies on dynamic memory allocation provided by the host operating system, not available on embedded devices. Fortunately, the standard library is built on a core library that neither requires dynamic allocations nor any host operating system. These minimal dependencies make it possible for Rust applications to be built and executed on bare metal targets. Likewise, many Rust libraries that may be needed on embedded environments come with support for "no_std", i.e. the library can be compiled without the standard library.

The Rust compiler rustc supports a wide range of platforms and architectures as well. Passing a compilation target to rustc allows compiling the source code to machine code optimized for the target platform.

A further step towards the maturity of Rust as a language for embedded systems is the framework Real-Time Interrupt-driven Concurrency (RTIC) [102], previously known as Real-Time For the Masses (RTFM) [34]. RTIC is a framework developed by the Embedded Systems group at the Luleå University of Technology (LTU) for leveraging the interrupt controller (shortly explained) to schedule tasks of different priority. At the same time, the risk of deadlock is mitigated thanks to all resource-accesses being handled by the Stack Resource Policy (SRP) [11].

The Stack Resource Policy is a method to handle resource access in real-time systems with a single shared stack. RTIC uses SRP to handle access to resources, and it guarantees a race and deadlock-free preemptive execution, along with bounded priority inversion. Additionally, the model does not use excessive memory for the stack since all the tasks share a common stack space.

RTIC implements the SRP using hardware interrupts. An interrupt signal is a hardware-generated event that usually occurs asynchronously with respect to the program's execution. Examples of signals include an electrical pin changing its logical state from 0 to 1 or vice versa, a timer reaching a specific value, or data becoming available on some communication interface. In response to these signals, the processor executes an interrupt handler, a special subroutine. Interrupts also have an associated priority level. In the ARM Cortex-M architecture case, all interrupts have higher priority than the thread mode, reserved for the common procedures, so handlers will preempt code running in thread mode.

Preemption works as follows: the processor suspends the execution of the current subroutine, saves the state of that subroutine (the processor registers) onto the stack and then jumps to the interrupt handler. The interrupt handler runs to completion and returns. Upon returning from the interrupt handler, the suspended routine's state is restored by popping the registers off the stack, and the preempted routine is resumed.

The Nested Vector Interrupt Controller (NVIC) manages all these interrupts peripheral [63]. The NVIC is a standardized interrupt controller found on all Cortex-M devices. The NVIC provides an interface to control and configure interrupts in the form of memory-mapped registers.

The way the NVIC handles interrupts [64] exactly matches how the RTIC task model prioritizes tasks. This fact is used

to provide a highly efficient implementation: the RTIC runtime lets the NVIC, the hardware, do all the task scheduling, so no bookkeeping needs to be done in software. Thus, the task's priority is bound to the interrupt service routine (ISR), matching that priority, associating an ISR to each interrupt handler, represented by a task. This alternative to time-sliced threads provides several advantages like reduced overhead, efficient memory utilization, predictable performance (constant time overhead), deadlock freedom, and suitability for static analysis [34].

4.8. Architecture

To the best of the author's knowledge, this thesis is the first to combine the vision of the computing continuum and the proposal of a technology baseline, whose soundness is demonstrated with a Proof of Concept (POC). Notably, the POC's goal is to prove that the technologies mentioned above can be integrated to support the realisation of the continuum's infrastructure layer.

The infrastructure layer is composed of data and computational resources. The data can be generated by various 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 edges.

To support a genuinely ubiquitous system where services are everywhere and anywhere the end-user is, for instance, following the movements across geographical zones, further architectural considerations have to be made. As an example, a federation layer may fit as a solution. The federation layer would be responsible for orchestrating the geographically distributed resources composing the underlying infrastructure. However, this thesis will not explore such a federation layer, and it is left as an open research question of future works on the continuum.

With the goal being said, a reference architecture of the infrastructure is shown in Fig. 9. From a high-level perspective, users submit applications to the orchestrator. The orchestrator provisions the necessary resources and services according to the application's requirements then assigns the latter to compute nodes in the cluster, where they are executed. The compute cluster is an abstraction of interconnected nodes on geographically distant clusters, such as clouds and fog nodes.

Users submit their applications in the form of a manifest. The manifest describes resources requirements in terms of the amount of CPU and memory they will require, for example. Likewise, the manifest contains the service requirements as well. Services may range from common dependencies like a database and persistent storage to data sources, which the cluster will provide based on the stated requirements. Service requirements allow developers to describe constraints on the services, e.g. latency needs and service plans. Unfortunately, this thesis has not explored standard manifest formats to describe such service requirements as of the time of writing. The POC merely leverages the Kubernetes YAML format, which specifies information like the

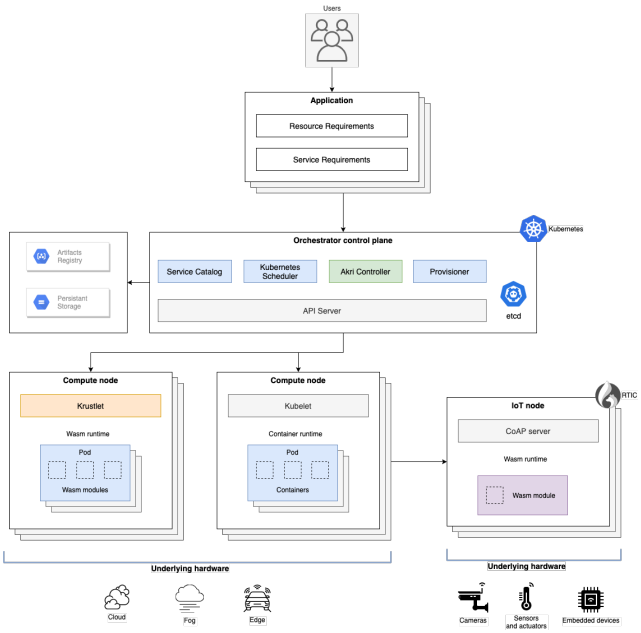

Figure 9: Reference architecture

container image, the placement constraints, and the resource requests.

4.8.1. *Orchestrator control plane*

The Kubernetes 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. In case resources are insufficient, applications can be rejected or put on weight until the resources are freed. Another possible solution would be to increase the number of cluster nodes to place the incoming application. Such nodes may be provisioned from local machines or anywhere in the network path from the end-user to the cloud datacentre. After determining if requirements can be satisfied, the scheduler maps application components, specifically Kubernetes Pods, onto the cluster resources. This deployment is done by considering several factors, e.g. the availabilities, the utilization of the nodes, affinities, priorities, or constraints.

In cases where the addition of new cluster nodes is possible, an automatic process is necessary. The provisioner is responsible for dynamically adding nodes (physical or virtual) to the cluster when the existing resources are insufficient to meet the applications' demands. It will also decide when nodes are no longer required in the cluster and shut the nodes down to prevent additional costs.

4.8.2. *Compute nodes*

Each machine in the cluster that is available for deploying Pods is a compute node. Each of these nodes has an

implementation of the Kubelet APIs with various responsibilities. First, it collects local information such as resource consumption metrics that can be periodically reported to the control plane. Second, it starts and stops Pods and manages local resources via a container runtime or a Wasm runtime. Finally, it monitors the Pods deployed on the node, sending periodic status to the control plane.

The reason for the presence of two types of compute nodes lies in the lack of maturity of the Wasm ecosystem at the time of writing. As previously mentioned, Krustlet is still far from reaching feature parity compared to a standard Kubelet implementation. For example, support for the Kubernetes networking is not implemented yet, nor are machine metrics collected and sent to the control plane. Besides, as stated before, the current WebAssembly System Interface (WASI) standard does not have complete networking support yet. Adding support for connecting to sockets is fundamental to allow Wasm modules to connect to web servers, databases, or any service. Because of the lack of network support, both as API in the Wasm module and as network configuration in the Kubernetes cluster, the POC uses a hybrid approach where conventional Kubelets with container runtimes are used as a fallback.

4.8.3. IoT nodes

IoT nodes are embedded devices that act as sensors and actuators, provided as service to the cluster. The IoT nodes are heterogeneous in runtime implementation and communication protocols. Therefore applications in the cluster interface with them via Akri brokers as described before. However, in the POC of this thesis, embedded devices use the RTIC runtime implemented in Rust and support dynamic configuration by running arbitrary Wasm modules in a lightweight Wasm runtime.

The Wasm runtime is implemented via an interpreter capable of executing any regular Wasm binary, assuming the file size and the hardware requirements can be satisfied by the limited device.

4.8.4. Underlying infrastructure

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

5. Evaluation

In this section, the author presents a performance evaluation of the Proof of Concept (POC) that implements the reference architecture presented in Fig. 9. The evaluation uses a local cluster of 4 machines and a microcontroller constituted by:

- 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, with only 1GB of total memory;
- A STM32F407 microcontroller with ARM Cortex-M4 core, 512KiB flash storage, and 128KiB of memory. The device is also capable of many 32-bit floating-point operations.

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

On the software layer, this thesis uses Rust 1.53.0-nightly (07e0e2ec2 2021-03-24) and Alpine Linux aarch64 v3.12.1 as OS. Alpine Linux has been chosen as it is a much more lightweight alternative to Ubuntu, and it offers a built-in Raspberry Pi version available for download. An idle Ubuntu Server 20.0 consumes about 420MB, whereas Alpine Linux incurs only 50MB of memory.

5.1. The application

As a practical example to guide the architecture's implementation, the thesis applies the continuum concept to a flood warning system. The existence of preprocessed data in the local sensor-actuator network can be leveraged to support rescue teams. Thus, the distributed architecture of the continuum is proposed as a viable architecture for advanced telemetry systems with distributed intelligence.

In a flood warning system, the telemetry stations acquire data from wireless sensors networks, process the data in a distributed manner, and locally determine potential levee breaking. Sensor networks represent a crucial component in IoT environments [122]. For example, they can cooperate with RFID systems to better track things, get information about the position, movement, temperature, etc. Sensor networks are typically composed of a potentially high number of sensing nodes, communicating in a wireless multi-hop fashion. Such networks can provide various valuable data and are being utilised in several areas like healthcare, government and environmental services (natural disaster relief), defence (military target tracking and surveillance), hazardous environment exploration, and seismic sensing.

The ideas and a large part of the implementation, which the thesis applies to the smart levee monitoring system, can also be applied to other modern data acquisition and distributed processing systems with similar requirements, including smart cities and smart agriculture. Risk assessment and early warning systems play a crucial role in minimizing the negative results of floods. Such systems may significantly contribute to detect a potential threat of a breach. Therefore, local flash floods may be dangerous because they occur in a short time, usually as a result of sudden heavy rainfall.

The application implements a system based on the architecture proposed previously in Fig. §9:

- Measuring nodes: it is composed of sensor devices that collect data from the measuring layer, preprocess it and transmits it to the central part of the system for further processing. One challenging task of this layer is implementing the dynamic reconfiguration of the internal logic, as preprocessing can result in more accurate predictions. However, the optimal parameters are continuously evaluated by the cloud machine learning models. In this thesis, the software runtime is implemented with RTIC. The interrupt-driven Stack Based Policy allows the definition of several tasks (e.g. network handling and data collecting), establishing the proper priorities depending on the requirements, and requiring a minimal stack size. Most of the memory is reserved for the WebAssembly interpreter. The reconfiguration can thus be done by replacing the application via over-the-air by transmitting a minimal Wasm binary file;

- Edge computing nodes: it provides a local database to store the information for the analysis and forecasting, which also happens locally. In a real-world environment, especially during flood threat scenarios, the communication channels may become unavailable, so the system must perform a localised flood risk assessment. Moreover, the gathered data can be processed in place. Only results can then be transmitted to the cloud to reduce the required bandwidth or minimize redundant information submitted.

- Cloud nodes: it stores the information for the machine learning training of the flood prediction model, a task which is typically delegated to the cloud datacentre because of the availability of powerful computational resources, such as specialized GPUs.

On the other hand, the machine learning inference can happen on each node of the system. On normal operational conditions, the inference could be executed on the cloud to provide more accurate results. However, when an incident happens and the Internet connection is lost, the edge nodes must be ready to provide the analysis locally. In some cases, it may additionally be reasonable to execute the prediction on the measuring node themselves, assuming enough hardware capability. Such execution would achieve the lowest latency in critical scenarios. As a result, the computation must be capable of flowing dynamically in the continuum space.

5.2. Evaluation of WebAssembly for microcontrollers

Fig. 10 and Fig. 11 present a comparison of the sizes of different Wasm binaries compiled from the Polybench [123] modules. The Polybench benchmark suite offers relevant functions to embedded systems as it includes common matrix and statistical operations.

Fig. 10 offers a direct comparison of binary size and source size. However, comparing with source code is rarely meaningful due to factors including the widespread use of

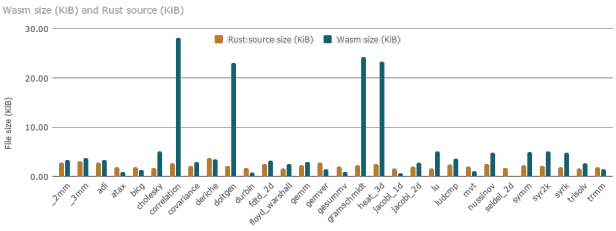

Figure 10: Comparison of Wasm size (KiB) and Rust source size (KiB)

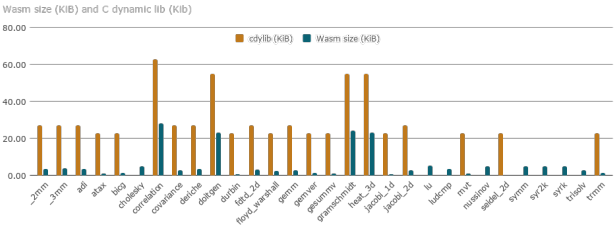

Figure 11: Comparison of Wasm size (KiB) and C dynamic library size (KiB)

conditional compilation and complex build systems. The purpose is to achieve an initial grasp of the Wasm binary size. The author has attempted to control for these effects by configuring the Rust toolchain to reasonable defaults. The compiler is configured to rely on LLVM to optimize the resulting code for code size. The toolchain also uses LLVM's link-time optimization (LTO) to ensure further size optimisation.

Once the compilation process is done, the result is a self-contained, sandboxed Wasm binary file ready for deployment onto the continuum.

The results show how the final Wasm binary size has no predictable correlation to the source size, despite being usually about 2x larger, with some exceptions. Modules like "correlation" show a significant increase in the Wasm size relative to source code lines. Such a variety of differences depends on the source code's specific instructions and the toolchain configuration. Notably, loops, inlined function and math functions can result in substantial differences in the output size because of the different optimisation techniques the compiler may take. Such unpredictability means that the Wasm toolchain is not immune to the compilers' famous idiosyncrasy, and developers must be careful always to check the final output.

The author has also chosen the C dynamic library size as a more meaningful comparison since it is a close alternative to Wasm binary files. Both outputs have been compiled using the same LLVM toolchain and with the same optimisation flags. In such a case, the results undeniably favour the Wasm binary format as the C dynamic lib is often many times larger. The author has even removed some of the Polybench modules (namely *cholesky, lu, ludcmp, nussinov, symm, syr2k, syrk, and trisolv*), which had a significantly larger C dynamic library size of 1.5MB (compared to about

5-10KiB in Wasm), which would make the other modules in the chart visually hard to evaluate. The comparison of Wasm files to containers would be even more favouring the former, as containers package a whole operative system filesystem. Even the smallest image base, "Alpine Linux Mini Root Filesystem", has an additional size of about 5.5MB uncompressed.

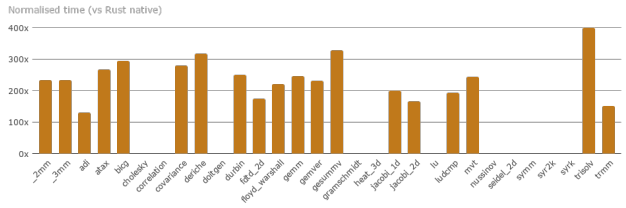

Figure 12: Comparison of Wasm interpreter performance and Rust native performance

Fig. 12 offers an overview of the Wasm interpreter's execution performance on the STM32F407 microcontroller. Each Polybench benchmark has been run 15 times, following the same methodology described in [116]. The results show a dramatic slowdown compared to the Rust native alternative, with a factor of 100-400 times. Such results are discouraging for the idea of using Wasm interpreters on microcontrollers to support dynamic reconfiguration.

However, it is fair to note that the Wasm interpreter wasmi [95] was adapted to work on embedded devices and was not meant to design highly constrained devices. wasmi is developed by Parity, a blockchain company, and is thus used to offer a deterministic sandboxed execution context. As a result, execution performance is not paramount, unfortunately. The alternative interpreter wasm3, implemented in the C language, shows a much more minor execution penalty, in the order of 30-60x slower than native [96]. Thus, it is reasonable to believe that future efforts may allow a comparable result for the Rust-built wasmi interpreter. Nevertheless, 30x times execution over time can still arguably deter the usage of interpreters in microcontrollers. Future work should also provide a benchmark with respect to other popular interpreted languages like Lua, Python, and JavaScript.

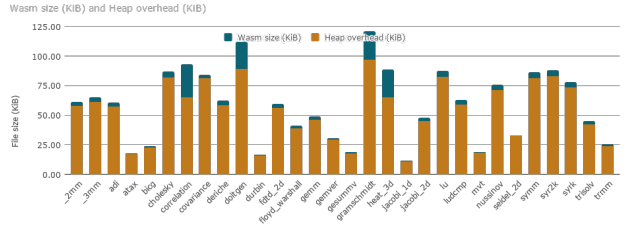

Figure 13: Comparison of Wasm size (KiB) and heap overhead (KiB)

As an additional benchmark, the author has evaluated the heap overhead of interpreting Wasm on microcontrollers. Fig. 13 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 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 and read-only data on the more capable flash storage. Writable data is saved in the stack instead, and it can be calculated with accuracy in many production-grade toolchains like C and Ada.

Generally speaking, running Wasm on resource-constrained microcontrollers has also presented other challenges.

- Memory consumption: Wasm's 64 KiB pages are too large for microcontrollers that often have between 16-256 KiB SRAM. Dynamic allocation is a common requirement even for embedded systems. However, Wasm specifies that the sandbox should expand the memory by 64KiB chunks, which is not granular enough for constrained embedded systems. As a consequence, the wasmi interpreter has been modified 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. There is a need for a Wasm embedded standard that supports smaller granularity of memory pages;
- Performance: although the bounds checking on linear memory and the indirect checking of function pointer invocations are necessary for proper isolation, they add overhead over non-sandboxed code. The authors in [96] provide a deep evaluation of different SFI alternatives in WebAssembly;
- Assumed ISA features. Wasm specifically is a 32-bit virtual architecture and supports floating-point, but not all Cortex-Ms do. In such cases, floating-point emulation must be provided by the programmer. In Rust, this solution is typically accomplished utilising the *libm* library, a port of MUSL's libm containing the most popular math operations. MUSL is a lightweight alternative to the conventional libc and employed as an alternative in Alpine Linux. However, the Rust ecosystem is less mature and complete on this matter compared to the C version, and the many floating-point functions may still be missing in the port.

5.3. Evaluation of CoAP for microcontrollers

Fig. 14 and Fig. 15 present the experimental evaluation of the CoAP implementation in Rust. The STM32F407 is equipped with a LAN8720 ETH 10/100Mbs module connected to the microcontroller via cheap jumper wires. On the software stack, the device offers a CoAP server running

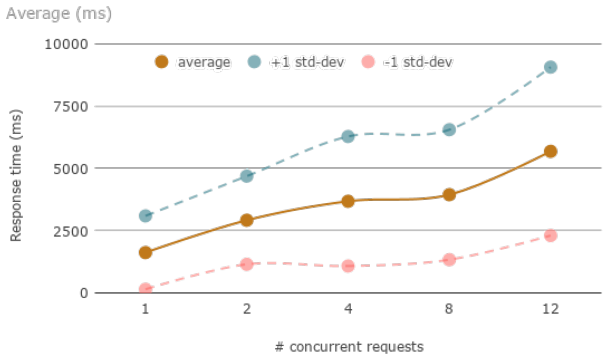

Figure 14: Average response time for concurrent CoAP requests

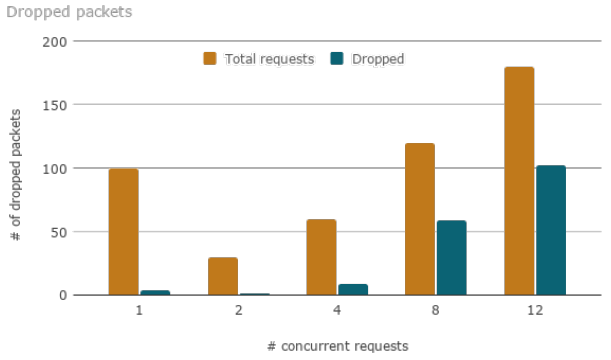

Figure 15: Number of dropped packets as concurrency increases

on the RTIC runtime. Such a server is put under stress by firing concurrent requests from a machine connected to the same network, and the test is repeated 15 times. As expected, the response times increase as the number of concurrent requests rises. However, such escalation of response time is not exponential and presents a good smoother curve. The standard deviation range is also significantly large, as the embedded device cannot provide timely responses. Further experiments should compare the CoAP server with an equivalent HTTP or MQTT server running on the same RTIC runtime.

On a different note, the number of dropped packets dramatically increases as the concurrency grows. Such behaviour is expected, and the experimental logs provide a plausible rationale. As the number of concurrent requests increases, the microcontroller network buffers cannot cope with the volume, and the packets are even truncated before being parsed. A deeper analysis should check whether the issue can be solved by allocating more buffer space or, on the contrary, is a hardware-related limit. Admittedly, the hardware network connection via jumper wires is not the most robust solution.

5.4. Evaluation of Wasm as virtualisation technology

The following benchmarks measure how long it takes to create and boot a Wasm-based Pod, how the memory scales as the number of running Pods and concurrent instantiation increase, and how both of these compare to containers. The benchmark is a simple DateTime application that logs the current system time upon creation and goes into sleep. By going to sleep, the Pod does not complete, and the resources remain allocated. The log is later retrieved using the Kubernetes API to calculate the boot time.

Wasm Pods run on Krustlet [72], whereas container Pods are scheduler on the K3s [99] Kubelet. Krustlet (a Kubernetes-rust-kubelet) is an experimental implementation of the Kubelet APIs that supports Wasm as virtualisation technology. It is designed to run as a Kubernetes Kubelet. Therefore, it listens to the Kubernetes API event stream for new Pods and runs them under a WASI-based runtime (notably, Wasmtime [4]).

On the other hand, K3s is a fully certified Kubernetes distribution geared towards resource-constrained environments backed by the Rancher company. As the Kubernetes community started seeing real demand for orchestration on edge, the Rancher team decided to make it a first-class open-source project. K3s is implemented in Go and packaged as a single binary of about 50MB in binary size. It bundles everything needed to run Kubernetes, including the container runtime (namely containerd [37]), a Container Network Interface implementation (Flannel [59]), a DNS server (CoreDNS [38]), and a simple ingress load balancer (Traefik [73]). The OS dependencies are also minimal and well supported by all the modern Linux kernels.

Krustlet has been customised to measure the time it starts to instantiate the Pod's necessary Wasm runtime. On the other hand, K3s and Kubernetes do not provide tools explicitly meant to obtain such value, and the author does not have the Go knowledge to modify and re-compile K3s. As a result, the start time is collected from the Kubernetes Pod metrics, which provides the time at which the Kubelet receives the request to provision the container. The provision request is followed by the container image's pull and then instantiation of the container itself. To avoid incurring the additional image pull time, the author explicitly sets the Pod configuration to use the cached image and ensure that the Kubelet always has such cache. Lastly, the granularity of this start time metric is only up to seconds.

Fig. 16 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 the memory utilisation as the number of running Pods increases is also collected.

The experimental results show that a Wasm-based virtualisation strategy incurs less boot time. However, the more mature container Kubelet presents a more linear curve and minor standard deviation as the number of concurrent de-

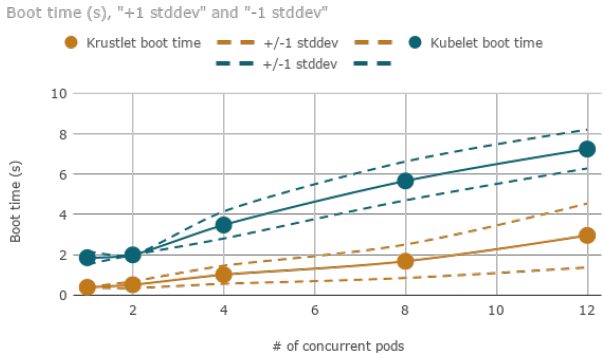

Figure 16: Average boot time for concurrent Wasm Pods

ployments increases. Because efficiency concurrency is essential as much as a fast boot time, there is no clear winner. Nevertheless, such preliminary results are encouraging the adoption of Wasm as an alternative to the container technology since efficiency is not a paramount design goal in the early implementations of Krustlet and Wasmtime. Future versions will probably provide even more competitive results.

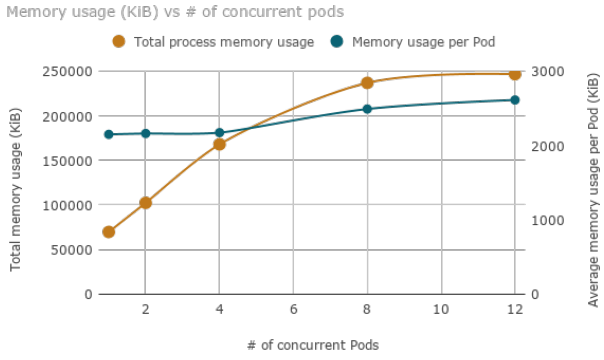

Figure 17: Average memory usage of concurrent Wasm Pods

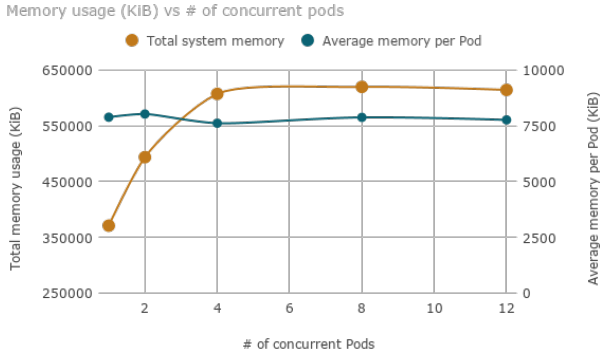

Figure 18: Average memory usage of concurrent K3s Kubelet Pods

Fig. 17 and Fig. 18 offer an overview of the memory overhead of the two different virtualisation solutions. The memory utilisation has been tracked using the Linux *pmap* utility, which allows the analysis of the system memory allocated by a process granularly. In Krustlet's case, the task is trivial as Wasm modules run in the same process as the host, and only additional Pods can significantly increase the heap memory, as Rust is not garbage collected. On the contrary, K3s instantiates a process for each new container. It is much more difficult to evaluate each Pod's memory cost, as the Go language uses the heap for every aspect. Therefore, in the latter case, we resort to measuring the system memory using the Linux *free* program. Fortunately, since the author uses Alpine Linux with a minimal configuration to support just enough to run Kubernetes, the system measurements are relatively free from other processes' contamination.

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

On a different note, as Go is a garbage-collected language, the heap utilisation is highly unpredictable. The Go Garbage Collector (GC) uses a pacer to determine when to trigger the next GC cycle [111]. Pacing is modelled like a control problem where it is trying to find the right time to trigger a GC cycle to hit the target heap size goal. Go's default pacer will try to trigger a GC cycle every time the heap size doubles. The net result is that the K3s shows an increasing memory usage even during idle periods, as it processes the ordinary Kubernetes operations to maintain functionality. Besides, as the GC kicks in only when the heap size doubles, the hardware memory is underutilised. The freeable memory should be used for running additional pods, achieving better Pod packing. Such efficiency is crucial for edge nodes that have already limited hardware capabilities but must support multiple workloads. This consideration is another point favouring the adoption of the Rust language to develop the continuum infrastructure.

Lastly, in both technologies, the memory overhead per Pod is relatively constant. However, the Wasm Pod incurs approximately 2x-3x less overhead, allowing more efficient packing of applications on the same machine. For example, the Raspberry Pi 3B+ can support up to 80 Wasm Pods and still have 500MB of free memory (although no more Pods can be allocated). In contrast, the K3s Kubelet achieves

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

maximum memory utilisation of about 55 container Pods, and the entire node becomes completely unresponsive. The author believes that such results pledge to favour Krustlet, as the premature Out Of Memory error can be fixed on later Wasmtime versions. In contrast, the container overhead is already the state of the art of a decade of research in container technologies.

5.5. Evaluation of Wasm for Machine Learning

In the field of machine learning, numerical precision matters. WASM natively supports floating-point arithmetic, whereas other popular machine learning backends like WebGL requires hardware extensions. Not all devices support this extension, which means a GPU-accelerated machine learning is not supported on some devices (e.g. older mobile devices which can run Wasm instead).

Moreover, GPU drivers can be hardware-specific and different devices can have precision problems. On iOS, 32-bit floats are not supported on the GPU. Consequently, TensorFlow [48], the popular machine learning library developed by the Google Brain team, falls back to 16-bit floats, causing precision problems. In WASM, computation always happens in 32-bit floats and thus have precision parity across all devices. Such numerical predictability is also paramount in blockchain applications, as proved by Parity's interest in WebAssembly [94].

The author has compiled the flood prediction model to Wasm, and Fig. 19 evaluates the model's response time with increasing concurrency. The flood prediction model is a conventional Deep Neural Network constituted by three dense layers of 32, 16 and 1 units. The model has been trained on the cloud using the traditional machine learning framework Keras [65]. Keras provides a Python interface for artificial neural networks on top of the TensorFlow library.

As of the time of writing, TensorFlow provides a WebAssembly (WASM) backend for both the browser and Node.js [110]. However, the library itself cannot be easily compiled to WebAssembly yet, to the best of the author's knowledge. As a result, the trained Keras model is saved to the ONNX [40]standard format and executed by the *tract* library [108]. tract is a Rust neural network inference library that can read Tensorflow or ONNX models and run data through them. It also supports compilation to Wasm as a target. The resulting Wasm module exports a public function through which Krustlet can pass the model inputs.

At the time of writing, it has not been possible to implement a web server directly on top of the inference model and compile the application to Wasm. There is an underlying issue with implementing network servers with Wasm as there is neither complete network nor multi-threading support yet.

First, the current WebAssembly System Interface (WASI) standard only contains a few methods for working with sockets, namely *sock_recv*, *sock_send*, and *sock_shutdown*, 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. As a workaround, this thesis has implemented the capability to invoke functions in Wasm Pods on top of the Kubernetes *exec* API [68], which is however designed to execute commands in a running container from the Kubernetes command line, and it is an undocumented interface.

Second, the lack of concurrency primitives means that a server running in WebAssembly is single-threaded, or its implementation has to be significantly more complex (e.g. Node.js's event loop). This limitation severely narrows the workload capabilities of the server. The WASM spec recently got a thread and atomics proposal with the goal to speed up multi-threaded applications. The proposal is still in the early stage, meant to seed a future Working Group, and it is only implemented under the experimental flag in web browsers.

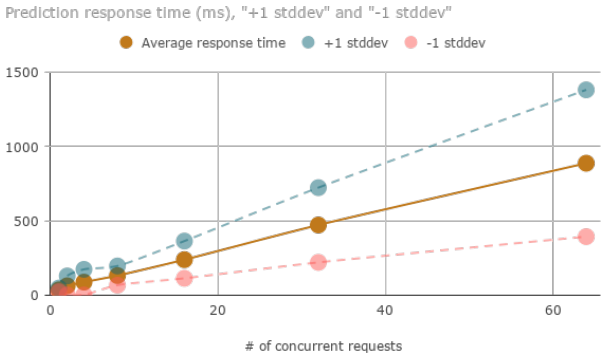

Figure 19: Latency of the Wasm flood prediction model with varying concurrency

Fig. 19 illustrates the flood prediction response times within a Wasm Pod, in the same fashion of serverless functions. As expected, the response times increase linearly with varying concurrency. Because the WASI standard does not support multi-threading, the requests are completed sequentially.

6. Conclusion

This thesis has presented a computing continuum vision and the challenges that the industrial and academic community will face implementing such a paradigm. The author has also presented an effort to suggest a potential architecture for the infrastructure and implemented a Proof of Concept of such architecture based on the technologies available at the time of writing. Such POC has been applied to a real-world use case of flood prediction as well. The net result is a working prototype based on the presented architectural and

software concepts. The implementation is publicly available on GitHub [57].

In the author's view, we are clearly at a convergence point after the last decades of technological advances. The computation must be free to flow in the continuum between the ends of the network. However, a significant number of challenges (§2) lie ahead, and an even greater effort from all the stakeholders is required.

The WebAssembly standard is a potential enabling technology for the realisation of the continuum, but it is still in its infancy. As shown by the experimental results, WebAssembly is mainly suited for pure computational functions. The lack of multi-threading and a mature network interface severely limit the space of real-world applications. Most of the production use cases of WebAssembly involve pure computation scenarios, typically machine learning and relatively simple serverless functions. Other projects like wasmcloud [76] (previously named wasCC) try to overcome the API limitations by providing custom system capabilities, to the detriment of the platform-independency feature of WebAssembly, unfortunately.

However, it is undeniable that the Wasm bytecode format has attracted interest from multiple parties like Cloudflare, Fastly, and Parity, despite WebAssembly was initially conceived for efficient execution in a browser context. These commercial companies are interested in its performance, portability and sandboxing capabilities for different scopes than browser execution. Other companies, namely Mozilla, Intel, Red Hat and Fastly again, have joined the efforts to the birth of the Bytecode Alliance [2]. As stated by the members, the official goal of the alliance is to "establish a capable, secure platform that allows application developers and service providers to confidently run untrusted code, on any infrastructure, for any operating system or device, leveraging decades of experience doing so inside web browsers."

Seeing such public announces from notorious industry leaders may bring excitement at first, but it is hard to state if such commitment is real or just a side project between many. In August 2020, Mozilla laid off most of its employees [112], notably the whole Wasm team and a large slice of the Rust core team. Many developers have been employed by other companies interested in Rust and Wasm, e.g. Fastly and Amazon. However, it is hard to tell if they will be able to maintain the commitment. For example, active development of wasm-bindgen [103] has ceased, even though wasm-bindgen is the most popular tool for high-level interaction between Wasm and the JavaScript browser runtime.

This evident interest pattern but lack of financial and stable support from commercial companies is typical in WebAssembly and Rust. Companies are still gaining interest in hiring Rust developers [6] to modernise their existing infrastructure rather than committing to new technologies like WebAssembly. For example, despite Intel being part of the Bytecode Alliance and author of a Wasm runtime for microcontrollers (wasm-micro-runtime [3]), there has not been progressing in discussing and proposing memory pages smaller than 64KiB in the WebAssembly standard, despite the issue being well-known for years [118].

Microsoft has been notably providing more financial support compared to the other cloud providers. The development of both the Krustlet project and the Akri project is lead by developers of the Deis Labs team [71], whose members maintain several vital projects in the cloud community (notably Helm, the package manager for Kubernetes). As a result, the future perspective for Krustlet and Akri is brighter than other projects like wasm-bindgen. Both projects are still highly immature, but they are expressive in depicting the future direction of Wasm and Kubernetes as critical technologies of the computing continuum.

Results show that Wasm-based Pods in Krustlet incur fixed low overhead in memory usage and can offer competitive boot times compared to more mature container solutions. The outcome is intriguing as these technologies (Krustlet and Wasmtime) are still in their infancy. Therefore performance or resource optimisation is not paramount.

Throughout this thesis, Rust has been a constant underlying technology in every aspect of the infrastructure. In the author's view, when there is a need to push the boundaries of application, be it compute, memory, or IO, there is no substitute for lifting the hood, understanding the low-level behaviours, and figuring out how to run efficiently and safely. The Rust language offers a precious language built with such requirements in mind.

As a concluding note, although the proposed architecture and the related technologies are still very immature, they open up a set of exciting opportunities for the continuum's realisation, which the author plans to investigate further in his future work.

A. My Appendix

Appendix sections are coded under `\appendix`.

`\printcredits` command is used after appendix sections to list author credit taxonomy contribution roles tagged using `\credit` in frontmatter.

References

[1] AbdelBaky, M., Zou, M., Zamani, A.R., Renart, E., Diaz-Montes, J., Parashar, M., 2017. 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.

[2] Alliance, B., 2021a. Bytecode alliance. https://bytecodealliance.org/. Accessed: 2021-05-28.

[3] Alliance, B., 2021b. wasm-micro-runtime. https://github.com/bytecodealliance/wasm-micro-runtime. Accessed: 2021-05-28.

[4] Alliance, B., 2021c. wasmtime. https://github.com/bytecodealliance/wasmtime. Accessed: 2021-05-28.

[5] archlinux, 2021. udev. https://wiki.archlinux.org/index.php/udev. Accessed: 2021-05-28.

[6] AWS, 2020. Why aws loves rust and and how we'd like to help. https://aws.amazon.com/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/. Accessed: 2021-05-28.

[7] AWS, 2021a. Firecracker. https://firecracker-microvm.github.io/. Accessed: 2021-05-28.

[8] AWS, 2021b. Iot greengrass. https://aws.amazon.com/greengrass/. Accessed: 2021-05-28.

[9] AWS, 2021c. Lambda. https://aws.amazon.com/lambda/. Accessed: 2021-05-28.

[10] AWS, 2021d. Lambda@edge. https://aws.amazon.com/it/lambda/edge/. Accessed: 2021-05-28.

[11] Baker, T.P., 1990. A stack-based resource allocation policy for realtime processes, in: [1990] Proceedings 11th Real-Time Systems Symposium, IEEE. pp. 191–200.

[12] Baresi, L., Mendonça, D.F., Garriga, M., 2017. Empowering low-latency applications through a serverless edge computing architecture, in: European Conference on Service-Oriented and Cloud Computing, Springer. pp. 196–210.

[13] Beck, M.T., Werner, M., Feld, S., Schimper, S., 2014. Mobile edge computing: A taxonomy, in: Proc. of the Sixth International Conference on Advances in Future Internet, Citeseer. pp. 48–55.

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

[15] Bellavista, P., Zanni, A., 2017. 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.

[16] Bernstein, D., 2014. Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Computing 1, 81–84.

[17] blog, S., 2015. Golang data races to break memory safety. https://blog.stalkr.net/2015/04/golang-data-races-to-break-memory-safety.html. Accessed: 2021-05-28.

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

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

[20] Brzoza-Woch, R., Konieczny, M., Nawrocki, P., Szydlo, T., Zielinski, K., 2016. 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.

[21] Caraza-Harter, T., Swift, M.M., 2020. Blending containers and virtual machines: a study of firecracker and gvisor, in: Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 101–113.

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

[23] Cisco, 2015. Cisco fog computing solutions: Unleash the power of the internet of things. https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-solutions.pdf. Accessed: 2021-05-28.

[24] Cloudflare, 2018. Webassembly on cloudflare workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/. Accessed: 2021-05-28.

[25] CNCF, 2021a. etcd. https://etcd.io/. Accessed: 2021-05-28.

[26] CNCF, 2021b. grpc. https://grpc.io/. Accessed: 2021-05-28.

[27] Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Elmeleegy, K., Sears, R., 2010. Mapreduce online., in: Nsdi, p. 20.

[28] Docker, 2021a. Docker image specification 1.0.0. https://github.com/moby/moby/blob/master/image/spec/v1.md. Accessed: 2021-05-28.

[29] Docker, 2021b. Hub. https://hub.docker.com/. Accessed: 2021-05-28.

[30] Docker, 2021c. Swarm mode overview. https://docs.docker.com/engine/swarm/. Accessed: 2021-05-28.

[31] Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L., 2017. Microservices: yesterday, today, and tomorrow. Present and ulterior software engineering , 195–216.

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

[33] Ericsson, 2020. Ericsson mobility report. https://www.ericsson.com/en/mobility-report. Accessed: 2021-05-28.

[34] Eriksson, J., Häggström, F., Aittamaa, S., Kruglyak, A., Lindgren, P., 2013. Real-time for the masses, step 1: Programming api and static priority srp kernel primitives, in: 2013 8th ieee international symposium on industrial embedded systems (sies), IEEE. pp. 110–113.

[35] Fastly, 2019. Announcing lucet: Fastly's native webassembly compiler and runtime. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime. Accessed: 2021-05-28.

[36] Fielding, R., 2000. Representational state transfer (rest). https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. Accessed: 2021-05-28.

[37] Foundation, T.L., 2021a. containerd. https://containerd.io/. Accessed: 2021-05-28.

[38] Foundation, T.L., 2021b. Coredns. https://coredns.io/. Accessed: 2021-05-28.

[39] Foundation, T.L., 2021c. Kubernetes. https://kubernetes.io/. Accessed: 2021-05-28.

[40] Foundation, T.L., 2021d. Onnx. https://onnx.ai/. Accessed: 2021-05-28.

[41] Foundry, C., 2016. Open service broker. https://www.openservicebrokerapi.org/. Accessed: 2021-05-28.

[42] Foundry, C., 2017. Launches online marketplace for expanding ecosystem. https://www.cloudfoundry.org/blog/cloud-foundry-launches-online-marketplace-expanding-ecosystem/. Accessed: 2021-05-28.

[43] Gadepalli, P.K., McBride, S., Peach, G., Cherkasova, L., Parmer, G., 2020. Sledge: a serverless-first, light-weight wasm runtime for the edge, in: Proceedings of the 21st International Middleware Conference, pp. 265–279.

[44] Gartner, 2017. Leading the iot. https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf. Accessed: 2021-05-28.

[45] Gartner, 2021. Predicts 2021: Cloud and edge infrastructure. https://www.gartner.com/en/doc/735107-predicts-2021-cloud-and-edge-infrastructure. Accessed: 2021-05-28.

[46] Google, 2021a. Cloud functions. https://cloud.google.com/functions. Accessed: 2021-05-28.

[47] Google, 2021b. Protocol buffers. https://developers.google.com/protocol-buffers. Accessed: 2021-05-28.

[48] Google, 2021c. Tensorflow. https://www.tensorflow.org/. Accessed: 2021-05-28.

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

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

[51] Guinard, D., Trifa, V., Wilde, E., 2010. A resource oriented architecture for the web of things, in: 2010 Internet of Things (IOT), IEEE. pp. 1–8.

[52] Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J., 2017. 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.

[53] Hall, A., Ramachandran, U., 2019. 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.

[54] Haller, S., Karnouskos, S., Schroth, C., 2008. The internet of things in an enterprise context, in: Future Internet Symposium, Springer. pp. 14–28.

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

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

[57] Hu, J., 2021. fedra-thesis. https://github.com/jiayihu/fedra-thesis. Accessed: 2021-05-28.

[58] Institute, P., 2016. Cost of data center outages. http://files.server-rack-online.com/2016-Cost-of-Data-Center-Outages.pdf. Accessed: 2021-05-28.

[59] flannel io, 2021. flannel. https://github.com/flannel-io/flannel. Accessed: 2021-05-28.

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

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

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

[63] Joseph, Y., 2014a. The definitive guide to arm cortex-m3 and cortex-m4 processors. ISBN-13 , 978–0124080829.

[64] Joseph, Y., 2014b. The definitive guide to ARM Cortex-M3 and Cortex-M4 processors. chapter Interrupt inputs and pending behaviors. pp. 978–0124080829.

[65] Keras, 2021. Keras. https://keras.io/. Accessed: 2021-05-28.

[66] Koller, R., Williams, D., 2017. Will serverless end the dominance of linux in the cloud?, in: Proceedings of the 16th Workshop on Hot Topics in Operating Systems, pp. 169–173.

[67] Kubernetes, 2021a. Device plugins. https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/. Accessed: 2021-05-28.

[68] Kubernetes, 2021b. Get a shell to a running container. https://kubernetes.io/docs/tasks/debug-application-cluster/get-shell-running-container/. Accessed: 2021-05-28.

[69] KVM, 2021. Kernel virtual machine. https://www.linux-kvm.org/page/Main_Page. Accessed: 2021-05-28.

[70] Labs, D., 2021a. Akri. https://github.com/deislabs/akri. Accessed: 2021-05-28.

[71] Labs, D., 2021b. Deis labs. https://deislabs.io/. Accessed: 2021-05-28.

[72] Labs, D., 2021c. Krustlet. https://github.com/deislabs/krustlet. Accessed: 2021-05-28.

[73] Labs, T., 2021d. Traefik. https://github.com/traefik/traefik. Accessed: 2021-05-28.

[74] Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J., et al., 2017. 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.

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

[76] LLC, W., 2021. Wasmcloud. https://wasmcloud.com/. Accessed: 2021-05-28.

[77] LLVM, 2021. Llvm. https://llvm.org/. Accessed: 2021-05-28.

[78] Lynn, T., Mooney, J.G., Lee, B., Endo, P.T., 2020. The cloud-to-thing continuum: opportunities and challenges in cloud, fog and edge computing. Springer Nature.

[79] MacKenzie, C.M., Laskey, K., McCabe, F., Brown, P.F., Metz, R., Hamilton, B.A., 2006. Reference model for service oriented architecture 1.0. OASIS standard 12.

[80] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., Crowcroft, J., 2013. Unikernels: Library operating systems for the cloud. ACM SIGARCH Computer Architecture News 41, 461–472.

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

[82] Microsoft, 2021a. Azure functions. https://azure.microsoft.com/en-us/services/functions/. Accessed: 2021-05-28.

[83] Microsoft, 2021b. Azure iot edge. https://azure.microsoft.com/en-us/services/iot-edge/. Accessed: 2021-05-28.

[84] Microsoft, 2021c. Typescript. https://www.typescriptlang.org/. Accessed: 2021-05-28.

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

[86] Mohanty, S.K., Premsankar, G., Di Francesco, M., et al., 2018. An evaluation of open source serverless computing frameworks., in: CloudCom, pp. 115–120.

[87] dotcom monitor, 2021. Visual traceroute. https://www.dotcom-monitor.com/wiki/knowledge-base/visual-traceroute-graphical-tool/. Accessed: 2021-05-28.

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

[89] Naik, N., 2017. 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.

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

[91] ONVIF, 2021. Onvif. https://www.onvif.org/. Accessed: 2021-05-28.

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

[93] Pahl, C., Helmer, S., Miori, L., Sanin, J., Lee, B., 2016. 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.

[94] Parity, 2020. Write wasm smart contracts with ink! 2.0. https://www.parity.io/write-wasm-smart-contracts-with-ink-2-0/. Ac-

cessed: 2021-05-28.
[95] Parity, 2021. wasmi. https://github.com/paritytech/wasmi. Accessed: 2021-05-28.
[96] Peach, G., Pan, R., Wu, Z., Parmer, G., Haster, C., Cherkasova, L., 2020. ewasm: Practical software fault isolation for reliable embedded devices. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39, 3492–3505.
[97] Pereira, P.P., Eliasson, J., Kyusakov, R., Delsing, J., Raayatinezhad, A., Johansson, M., 2013. Enabling cloud connectivity for mobile internet of things applications, in: 2013 IEEE seventh international symposium on service-oriented system engineering, IEEE. pp. 518–526.
[98] Pi, R., 2021. Products. https://www.raspberrypi.org/products/. Accessed: 2021-05-28.
[99] Rancher, 2021. k3s. https://k3s.io/. Accessed: 2021-05-28.
[100] RedHat, 2020. Introduction to kubernetes architecture. https://www.redhat.com/en/topics/containers/kubernetes-architecture. Accessed: 2021-05-28.
[101] Rosenblum, M., Garfinkel, T., 2005. Virtual machine monitors: Current technology and future trends. Computer 38, 39–47.
[102] RTIC, 2021. Rtic. https://rtic.rs/0.5/. Accessed: 2021-05-28.
[103] rustwasm, 2021. wasm-bindgen. https://github.com/rustwasm/wasm-bindgen. Accessed: 2021-05-28.
[104] Satyanarayanan, M., Bahl, P., Caceres, R., Davies, N., 2009. The case for vm-based cloudlets in mobile computing. IEEE pervasive Computing 8, 14–23.
[105] Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L., 2016. Edge computing: Vision and challenges. IEEE internet of things journal 3, 637–646.
[106] Shillaker, S., Pietzuch, P., 2020. Faasm: lightweight isolation for efficient stateful serverless computing, in: 2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20), pp. 419–433.
[107] Shopify, 2021. App store. https://apps.shopify.com/. Accessed: 2021-05-28.
[108] Sonos, I., 2021. tract. https://github.com/sonos/tract. Accessed: 2021-05-28.
[109] Swagger, 2021. Openapi specification. version 3.0.3. https://swagger.io/specification/. Accessed: 2021-05-28.
[110] Tensorflow, 2020. Introducing the webassembly backend for tensorflow.js. https://blog.tensorflow.org/2020/03/introducing-webassembly-backend-for-tensorflow-js.html. Accessed: 2021-05-28.
[111] Twitch, 2019. Go memory ballast: How i learnt to stop worrying and love the heap. https://blog.twitch.tv/en/2019/04/10/go-memory-ballast-how-i-learnt-to-stop-worrying-and-love-the-heap-26c2462549a2/. Accessed: 2021-05-28.
[112] Verge, T., 2020a. Mozilla is laying off 250 people and planning a 'new focus' on making money. https://www.theverge.com/2020/8/11/21363424/mozilla-layoffs-quarter-staff-250-people-new-revenue-focus. Accessed: 2021-05-28.
[113] Verge, T., 2020b. Prolonged aws outage takes down a big chunk of the internet. https://www.theverge.com/2020/11/25/21719396/amazon-web-services-aws-outage-down-internet. Accessed: 2021-05-28.
[114] Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L., 1993. Efficient software-based fault isolation, in: Proceedings of the fourteenth ACM symposium on Operating systems principles, pp. 203–216.
[115] wasm3, 2021a. wasm3. https://github.com/wasm3/wasm3. Accessed: 2021-05-28.
[116] wasm3, 2021b. wasm3 performance. https://github.com/wasm3/wasm3/blob/main/docs/Performance.md. Accessed: 2021-05-28.
[117] WebAssembly, 2020. Reference types overview. https://github.com/WebAssembly/reference-types/blob/master/proposals/reference-types/Overview.md. Accessed: 2021-05-28.
[118] WebAssembly, 2021. Embedded devices: i8/i16 and memory pages less than 64kib. https://github.com/WebAssembly/spec/issues/899. Accessed: 2021-05-28.
[119] Whitmore, A., Agarwal, A., Da Xu, L., 2015. The internet of things—a survey of topics and trends. Information systems frontiers 17, 261–274.
[120] Yi, S., Hao, Z., Zhang, Q., Zhang, Q., Shi, W., Li, Q., 2017. Lavea: Latency-aware video analytics on edge computing platform, in: Proceedings of the Second ACM/IEEE Symposium on Edge Computing, pp. 1–13.
[121] Yousaf, F.Z., Bredel, M., Schaller, S., Schneider, F., 2017. Nfv and sdn—key technology enablers for 5g networks. IEEE Journal on Selected Areas in Communications 35, 2468–2478.
[122] Yu, W., Liang, F., He, X., Hatcher, W.G., Lu, C., Lin, J., Yang, X., 2017. A survey on the edge computing for the internet of things. IEEE access 6, 6900–6919.
[123] Yuki, T., 2014. Understanding polybench/c 3.2 kernels, in: International workshop on Polyhedral Compilation Techniques (IMPACT), pp. 1–5.
[124] Zdnet, 2020. Microsoft spots malicious npm package stealing data from unix systems. https://www.zdnet.com/article/microsoft-spots-malicious-npm-package-stealing-data-from-unix-systems/. Accessed: 2021-05-28.
[125] Zhang, Q., Cheng, L., Boutaba, R., 2010. Cloud computing: state-of-the-art and research challenges. Journal of internet services and applications 1, 7–18.