Cloud-Edge-Client Continuum: Leveraging Browsers as Deployment Nodes with Virtual Pods

Mario Colosi
mcolosi@unime.it
University of Messina
Italy
Alma Digit SRL, Messina
Italy

Marco Garofalo
magarofalo@unime.it
University of Messina
Italy
marco.garofalo@phd.unipi.it
University of Pisa
Italy

Antonino Galletta
angalletta@unime.it
University of Messina
Italy

Maria Fazio
mfazio@unime.it
University of Messina
Italy

Antonio Celesti
acelesti@unime.it
University of Messina
Italy

Massimo Villari
mvillari@unime.it
University of Messina
Italy
Alma Digit SRL, Messina
Italy

ABSTRACT

Nowadays, thanks to the ever-increasing hardware capacity of Edge computing, the achievement of Ubiquitous Computing is no longer a utopia, even though it presents still several challenges. In this paper, we introduce the concept of the Cloud-Edge-Client Continuum, by extending the well-known Cloud-Edge Continuum paradigm with the addition of Clients as deployment nodes. Specifically, we propose both a system architecture and a piece of middleware that allows a web browser to be used seamlessly as a deployment Client node, introducing the concept of a Virtual Point of Deployment (VPod). Our solution allows to: a) leverage the computational capacity of a huge number of ready-to-use devices that do not require the installation of any dependencies; b) optimize the use of resources with clear benefits for end users, who can take advantage of their computing capacity to process sensitive data; c) reduce infrastructure costs. In addition, our proposal opens toward a multitude of scenarios, as the logical division that exists in the common client-server architecture is overcome, enabling the creation of a Cloud-Edge-Client Continuum environment.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**; • **Human-centered computing → Ubiquitous and mobile computing**; • **Computing methodologies → Distributed computing methodologies**.

BDCAT '23, December 4–7, 2023, Taormina (Messina), Italy
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0473-4/23/12…$15.00
https://doi.org/10.1145/3632366.3632395

KEYWORDS

Cloud-Edge Continuum, Pervasive Computing, Software Architecture, Volunteer Computing, Edge Computing, Virtual Pod, Browser, WebAssembly

ACM Reference Format:
Mario Colosi, Marco Garofalo, Antonino Galletta, Maria Fazio, Antonio Celesti, and Massimo Villari. 2023. Cloud-Edge-Client Continuum: Leveraging Browsers as Deployment Nodes with Virtual Pods. In *IEEE/ACM 10th International Conference on Big Data Computing, Applications and Technologies (BDCAT '23), December 4–7, 2023, Taormina (Messina), Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3632366.3632395

ABBREVIATIONS

WASM	WebAssembly
Pod	Point of deployment
VPod	Virtual Pod
VPod(C)	Client-side Virtual Pod
VPod(S)	Server-side Virtual Pod
HVPA	Horizontal Virtual Pod Autoscaler
S-Connector	Server Connector
C-Connector	Client Connector

1 INTRODUCTION

Recently, the Cloud-Edge Computing paradigm has already become a tangible practice for extending the Cloud capabilities to the Edge layer, thus leveraging resources close to the data sources and end-users.

For the past decade, the trend of exponential growth in the number and computational capacity of the Internet of Things (IoT), through smart devices capable of communicating and exchanging data over the Internet, has already become evident. In particular, the practice of managing the interoperability of such devices, and their integration within applications, has made use mainly of Web standards, eventually composing what is known as the Web of Things (WoT) [26].

Nevertheless, despite the ever-growing computing power at the Edge layer, the achievement of truly Ubiquitous Computing (or Pervasive Computing) appears to be forthcoming but still hindered by several limitations

[9], mainly due to heterogeneous hardware characteristics of devices, network constraints, and issues in decentralized application management. In addition, the ability to integrate Cloud and Edge efficiently, following the paradigm of Continuum Computing, is to date one of the hottest topics addressed in the literature. Indeed, there are several challenges still open including granular IoT management, serverless computing, federated resource allocation and management, optimization of energy consumption, and data locality [4].

In this context, we propose a new system architecture and a piece of middleware able to extend the Cloud-Edge Continuum (see Figure 1) by leveraging the web browser as a microservice deployment Client node. This opens towards a multitude of scenarios with reference to each point highlighted above, as the logical division existing in the common client-server architecture is overcome and a single **Cloud-Edge-Client Continuum** environment (see Figure 2), in which application components can communicate directly and transparently as if they were all in the same network, is created.

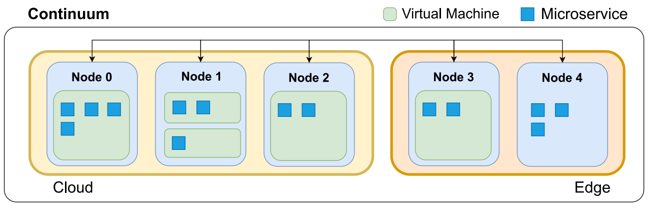

Figure 1: High-level representation of the Cloud-Edge Continuum composed of multiple nodes distributed between Cloud and Edge.

Using this novel approach, applications composed of distributed microservices can benefit from several advantages and possibilities:

- optimization of **resource usage**: client devices have an increasing computational capacity, which thus becomes possible to use dynamically without affecting the application design logic (computation offloading);
- **cost reduction**: the ability to use their own client resources can be advantageous both for the end user and for the infrastructure costs of the application;
- advantages in handling **sensitive data**: in some contexts, it may be essential to elaborate or pre-process sensitive data directly on the client, without it being transferred to the network;
- enable **resource sharing**: users can, through applications, make available the computational capacity of their devices by natively implementing Volunteer Computing logic;
- elimination of **logical separation** between client and server: as already pointed out, the design and implementation of an application can be carried out considering a single environment.

The aim of this paper is therefore to describe the system architecture and a piece of middleware allowing to achieve the objectives of Cloud-Edge-Client Continuum. The new concept of **Virtual Point of deployment** (VPod) is also introduced, through which transparent management of deployment and communication for client-side microservices is made possible. An application use case is also presented which is useful for analyzing the results obtained and highlighting strengths. The remainder of this paper is organized as follows: Section 2 describes the state of the art with reference to the Cloud-Edge Continuum and the computational possibilities inherent in the topic discussed. In Section 3, we present the main concepts behind the Cloud-Edge-Client Continuum, while Section 4 describes the implemented middleware architecture that enables the achievement of the set goals. In Section 5, we present experiments validating our solution. Conclusions and future research directions are discussed in Section 6.

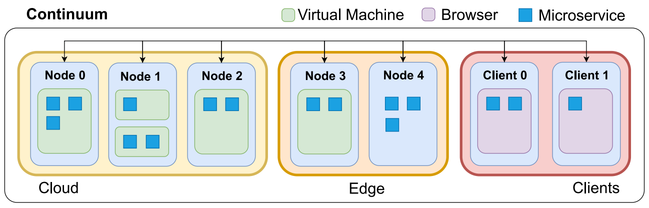

Figure 2: High-level representation of the Cloud-Edge-Client Continuum composed of multiple nodes distributed between Cloud, Edge, and Clients (leveraging web browsers). Comparing the representation with Figure 1, we observe the extension of the Edge with the addition of Clients as microservices deployment nodes

2 BACKGROUND AND RELATED WORKS

Cloud-Edge Continuum [22] refers to a computing environment that seamlessly integrates and spans a spectrum of computing resources and capabilities, from Edge devices and Fog Computing at the network's edge to centralized Cloud data centres. The compute Continuum has already demonstrated its effectiveness as a solution across various domains. For example, in [3], the impact of the Continuum was studied on a specific urban mobility use case. The study focused on using geotagged data to predict taxi passenger destinations, comparing Edge-Cloud Continuum, Edge-only, and Cloud-only architectures. Results from experiments showed that the Edge-Cloud Continuum with defined policies was superior to traditional architectures in terms of processing time and resource utilization. In the healthcare sector [12], the application of Cloud-Edge solutions is also proving to be a logical consequence of the rise of wearable IoT devices and sensor technologies. Furthermore, applications with strong security constraints can take advantage of decoupling Cloud and Edge [2]. For example, Federated Learning (FL), in which Edge clients perform training locally without exposing sensitive data to Cloud nodes, greatly benefits from running in the Continuum environment [10]. In addition, Homomorphic Encryption [6] finds a natural application in Continuum, as encryption operations can be performed in the Edge layer and transferred to the Cloud for storage and processing without decryption.

However, the compute Continuum presents a number of challenges that still need to be solved. The problem of offloading has been investigated by several proposals. For example, in [24], special attention is paid to determining when to perform offloading in Big Data analytic applications. The authors demonstrated that significant reductions in total task completion time can be realized through the strategic utilization of migration practices, shifting computational workloads from edge nodes to Cloud-based computing resources. Differently, the authors of [5], argue that a shift from the Cloud-Edge Computing Continuum paradigm to the Edge-Edge one would enhance the deployment of Artificial Intelligence (AI) at the edge of the network, by exploiting energy consumption forecasting. Especially in Mobile Edge Computing (MEC), the problem of offloading through the Continuum presents several issues [11]. Indeed, when Edge servers have limited resources, the problem of reducing the total task execution time is non-trivial due to the time dependency among different tasks. Moreover, in a heterogeneous computing environment, the selection of appropriate Edge servers for task assignment can become a complex problem, especially when dealing with tasks that have specific computational constraints, such as those of Augmented Reality (AR) or AI applications.

Another significant challenge is related to service portability, as the Continuum, by definition, is composed of heterogeneous systems. The Web of Things (WoT) [1] concept partially addresses this problem, as it aims to improve interoperability among Internet of Things (IoT) devices by leveraging Web protocols. However, it also introduces additional problems, particularly in the areas of security and smart object discovery. The new solutions are based on the use of WebAssembly (WASM) [15], an innovative technology that has transformed the Web development landscape. WASM is a binary instruction format tailored to facilitate high-speed execution of code within web browsers. It provides a versatile, secure, and resource-efficient mechanism for executing applications developed in languages such as C, C++, and Rust directly within web browsers, along with JavaScript. This technology serves as a unifying execution environment for Web applications, breaking down traditional divisions between various programming languages. A number of studies have scrutinized the performance disparities between WASM and JavaScript, revealing that WASM excels over JavaScript in multiple aspects and across most tasks [7, 25]. This research has transcended the realm of client-side applications, leading to innovative propositions of WASM as standalone runtimes. For instance, WasmEdge[1], a lightweight, high-performance WASM runtime tailored for Cloud-native, Edge, and decentralized applications, has fostered investigations into WASM's applicability as a containerization technology [13, 23]. This exploration even extends to well-known platforms like Docker[2], which introduced a technical preview founded on the WASM runtime[3]. The findings of these efforts are promising and show a significant reduction in cold-start times and a decrease in container image sizes.

Research on the compute Continuum is evolving rapidly, outlining possible strategies to solve current challenges. However, the most intricate challenge concerns the expansion of the Continuum itself into a new configuration that can effectively address the problems of heterogeneity and resource allocation, particularly in the context of data-intensive tasks. For example, in [18] various difficulties associated with the integration of non-Von Neumann architectures (e.g., quantum or neuromorphic computers) across the Continuum are outlined. The authors motivate this scenario by emphasizing the large computational capacity of non-von Neuman architectures. However, they also note that the inherent diversity among these systems could exacerbate the distribution of workloads throughout the Continuum, especially if a solution for ensuring portability is not in place.

In the current context, several solutions identify ad hoc Kubernetes[4] extensions to facilitate orchestration across the Continuum. An example of this is presented in [14], where FLEDGE is introduced as a low-level container orchestrator designed to seamlessly connect with Kubernetes clusters. This solution concerns the minimization of the overhead added by Kubelet on low-resource Edge devices. Specifically, the authors exploit the concept of Virtual Kubelet, a transparent implementation of Kubelet that acts as a proxy between Kubernetes and the FLEDGE agent, which handles typical operations of a container orchestrator, such as network management and cgroups. Conversely, the authors of [16] argue that the evolution of the Cloud-Edge Continuum is likely to be a paradigm shift toward *Liquid Computing*, a new paradigm for a transparent Continuum of computational resources over fragmented infrastructure, characterized by a decentralized, dynamic and intent-driven approach. They also present liqo[5], an open-source project that realizes this vision by abstracting dynamic Kubernetes multi-cluster topologies. Experimental results demonstrate the effectiveness of liqo with minimal overhead compared to vanilla Kubernetes and better performance than existing open-source solutions. Again, WASM represents a transitional technology at this stage of the Continuum progression. In fact, the authors in [21], envision an interoperable, scalable, and distributed Cloud-Edge Continuum that allows developers to focus on business value rather than infrastructure complexities by leveraging WebAssembly. As a practical demonstration, the authors demonstrate the feasibility of WASM on Continuum by running tests on different architectures, i.e. X86 and Arm, showing acceptable run-time overhead. WASM-based solutions, on the other hand, favor the development of runtime implementations aimed at strengthening the serverless paradigm, as seen in studies such as [17] and [20]. These efforts are driven by the desire to improve portability and performance in serverless computing environments.

Although the future directions of the Continuum are promising and include several aspects of future scenarios, only a few proposals have focused on the inclusion of the web browser via WASM within the Continuum as an additional computational resource. In [19], the Continuum is represented as the ideal environment for Browser-Based Volunteer Computing (BBVC), using WASM to solve the portability problem and MapReduce [8] for task distribution. The results show how the use of different browsers in distributed computing can greatly contribute to reducing the computation time of computationally intensive tasks.

3 CLOUD-EDGE-CLIENT CONTINUUM

In the modern concept of Edge Computing, it is common to include also devices directly used by the user, such as laptops and smartphones. However, with the term **Cloud-Edge-Client**, we want to emphasize that the device added to the classical Cloud-Edge computation environment is a mere web client, which becomes part of the Continuum and erases the logical separation of client-server applications, partly losing some of its characteristic of client and creating a unique distributed computational environment. To facilitate the description, therefore, with Cloud-Edge we will refer to the server side, while the client will also be referred to as a synonym of the web browser.

Adding a client device to the *Continuum* means, first of all, offering the possibility of enlarging the management of heterogeneity and the dynamism of computational resources on such devices, making sure that the developer of a distributed application does not have to be concerned about how and where the application components will be executed, neither from the point of view of their deployment nor of their communication. The execution location of a process within the Continuum must therefore be totally transparent to the application logic and managed in an infrastructure-aware and autonomous manner.

To achieve these goals, without installing architectural components within clients so that we can span a huge amount of *ready-to-use* devices, we argue that choosing to use a web browser as the execution environment is the optimal solution, ideally supplying an impressive additional computational capacity. Furthermore, since to date most of the applications deployed on the Continuum are managed through **Kubernetes** orchestration, our vision is to make use of its features and policies and also to enable the expansion of architectures that already use it.

3.1 Microservices Deployment Within Web Browsers

A microservice can be defined as a software component of a modular architecture that specializes in a domain-specific function of an application. Its main characteristics, in this context, include autonomy from other microservices, isolation from the surrounding environment, communication interface via API (e.g., RESTful), and scalability. To have these features, it is common to implement and deploy a microservice within a container, which is managed through orchestration tools such as Kubernetes.

To deploy a microservice inside a web browser, it therefore becomes essential to comply with these characteristics. This can be achieved simply by combining the features and standards that modern browsers provide.

[1]https://wasmedge.org/
[2]https://www.docker.com/
[3]https://www.docker.com/blog/announcing-dockerwasm-technical-preview-2/
[4]https://kubernetes.io/
[5]https://liqo.io/

3.1.1 *Web Workers.*

Isolation, autonomy, and a container-like environment within the web browser can be obtained with the **Web Worker** technology. A Web Worker, indeed, executes its code in a separate thread, thus performing tasks in parallel and in the background without affecting the responsiveness of the Web application user interface. They are also natively isolated, as they have their own separate execution context and do not share memory with the main thread or other Web Workers. This is a significant strength both in terms of performance since multiple instances of the same Web Worker can be launched, and in terms of security.

The execution code, requested as input when creating the Web Worker, is a simple Javascript code. Depending on the microservice we want to execute, we can use the appropriate launcher, which has several possibilities based on the language in which our microservice is implemented:

- **Javascript**, dynamically importing the code as a module;
- **WebAssembly**, natively supported by modern web browsers and obtainable by compiling different programming languages, such as C, C++, Rust, Golang, AssemblyScript, and others;
- **Python**, taking advantage of libraries such as Pyodide [6].

In particular, the use of WASM proves to be especially suitable in our case, not only because of its performance but also for the possibility of running the same code as the microservices executing on the server side, which, with proper care, can often be entirely or almost entirely reused. Of course, it is necessary to keep in mind what the restrictions of the browser environment are, for example in relation to direct system calls and filesystem access, as well as memory limitations. On these aspects, it is the responsibility of the application developer to find the most appropriate solution.

3.1.2 *Communication Via API.*

As pointed out in Section 3.1, one of the main aspects of microservices is their ability to communicate by exposing the Application Program Interfaces (APIs). Web Workers are an isolated execution environment that allows only asynchronous communication based on message exchange. At the same time, technologies such as WASM that run in sandboxes, add an additional layer of isolation that must be managed:

- for the **outgoing communication**, the Web Worker is free to use the APIs made available by other microservices, for example by making HTTP calls;
- for the **incoming communication**, on the other hand, the solution is to define a standard of communication, in which the running process of the microservice exposes methods to the Web Worker that are mapped based on the structure of the received message. Considering the structure of a RESTful call, by convention, we use the first segment of the HTTP request path as the name of the exposed function to be called; any other segments, query parameters, body and headers instead become input parameters of the function itself.

3.2 Virtual Pod

Having defined the methodology for deploying a microservice within the web browser, one of the main aspects, in order to comply with compute Continuum principles, is to allow the other microservices in the cluster to interact with it transparently, without any difference from the same microservice deployed server-side across the Cloud-Edge environment. In a Kubernetes cluster, for example, setting up a *service* allows pods to communicate with each other using just the service name, leaving Kubernetes to handle the microservice address resolution and abstracting its deployment location. To this end, we introduce one of the main components of our architecture, the **Virtual Pod** (VPod).

[6]https://pyodide.org

From the point of view of Kubernetes and the microservices that make up the cluster of a distributed application, a VPod is a common fully-fledged Pod, which can be contacted through the definition of a service exploiting the Kubernetes DNS and whose lifecycle can be managed entirely by Kubernetes.

A VPod is thus an object deployed on the server in the Cloud-Edge, whose role, within the cluster, is *to represent* the microservice deployed on the client, in our case inside the browser in the form of a Web Worker. Therefore, we can affirm that each microservice on the client uniquely corresponds to a VPod. It is important to underline that the same concept of VPod is not limited to just the browser, but can be extended to other types of clients or nodes that cannot be managed directly through Kubernetes.

With the definition of VPod, we used the verb *to represent* because, in reality, the VPod does not execute the code of the microservice with which it is associated, but rather it trivially acts as a reverse proxy, handling communications in both directions, from the cluster to the microservice in the client and vice versa. All the computation takes place within the client and the VPod only handles the communication, acting as a bridge. Consequently, as shown in Figure 3, this allows the following two flows to be handled appropriately:

- a microservice in the cluster that wants to interact with the microservice deployed on the client can simply interface with the VPod, from which it will also receive the response. In fact, it will be the latter, acting as a reverse proxy, that will turn the request to the microservice on the client, wait for the response, and deliver it to the requestor, thus making the integration flow totally transparent for the microservice in the cluster;
- the microservice on the client that wants to contact a microservice on the cluster, on the other hand, interfaces with its VPod, which again takes over the request, interacts with the microservice, and returns the response to the requester. Again, for the microservice in the cluster, the client layer is transparent, as its view is simply that of a request coming from a pod in the cluster, i.e., the representative VPod.

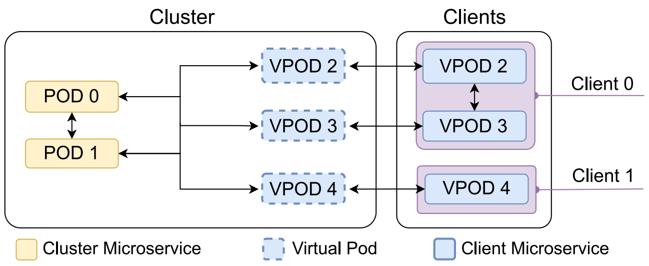

Figure 3: Example of a cluster composed of Pods and Virtual Pods, which represent the computational entities located in the clients and act as a bridge in managing communication.

Since from the cluster point of view, VPod and its corresponding microservice run within the client are the same entity, from now on, by the term VPod we will refer not only to the component inside the cluster but also to the associated microservice initiated in the web browser. When it is necessary to distinguish, we will instead use the terminology VPod(C) to refer to the microservice on the client, and VPod(S) for its server-side counterpart.

3.3 Client and Cluster Space Sessions

Considering the web browser used as a deployment location, one of the key aspects to consider is the need for the user to have the web page from which they access the application open so that they can make resources available and become part of the Continuum. This means that both the underlying architecture and the overlying application must be able to dynamically handle the presence and absence of a client node.

Furthermore, as the web browser is closely tied to the user, we can consequently introduce the concept of *session*, which also becomes important with regard to the intentions of the client user who makes his or her resources available and the possible contribution s/he wants to make within the cluster. We then define two types of client nodes, supported by the proposed architecture, that can be managed throughout the application:

- **user session space node**: the browser hosts only VPods that are related and useful to its session in relation to the application purposes;
- **cluster space node**: the browser becomes effectively a cluster node and hosts VPods also related to other users' sessions, based on policies defined at the application level, thus creating a form of Volunteer Computing.

3.4 Enabling Virtual Pods

As mentioned in Section 3.1.1, Web Workers are the solution for the deployment of microservices within the browser, and having overcome the obstacle of infrastructural transparency in communication through the introduction of VPods, it is now necessary to manage and make these components interoperable by introducing new architectural elements, which are also capable of automatically handling the session concept.

VPods(S), in fact, need to interact with their counterpart VPod(C) and must be started or removed automatically based on both the needs of the running application and the user's connection and disconnection from the session. In addition, it is desired to avoid publicly exposing their interface on the Internet, making it preferable to create a single communication channel between the client and the server.

For this reason, two connectors are introduced at the architectural level, representing the entry points to the bridge that connects the two faces of VPods, thus having the task of unifying the client-server environment and managing the different user sessions based on their own responsibilities.

Similarly, VPods' lifecycle management needs to be automated, and in this case, we leverage Kubernetes capabilities, with the aim of complying with application policies and keeping the state of the two VPod faces aligned. All these aspects will be described in detail in the next section.

4 MIDDLEWARE ARCHITECTURE

This Section describes the system architecture of the middleware implemented to enable the management of the Cloud-Edge-Client Continuum environment. The components that make up the basic architecture, through which generic distributed applications can be managed, are first shown and analyzed individually below, and then as a whole.

4.1 Server-side components

The Cloud-Edge environment consists of a cluster orchestrated by Kubernetes, composed of the middleware and the software application. The middleware components mainly have two tasks: to handle the user session and to manage the lifecycle and communication of VPods.

4.1.1 Kubelet.

Kubelet is described as the "node agent" in a Kubernetes cluster. Its role is to act as the bridge between the control plane, which includes the Kubernetes API server, and the worker nodes. For this reason, each worker node in the cluster runs a Kubelet instance.

At its core, Kubelet is responsible for ensuring that containers and pods are running correctly on its assigned node. It receives pod definitions from the Kubernetes API server, often referred to as PodSpecs. These manifest files define the desired state of pods, including which container images should run, their resource specifications, and other configurations. However, Kubelet does not handle containers not instantiated by Kubernetes, including our client-side pod representation as VPod(C). To tackle this problem, we designed and developed the ad hoc components described below.

4.1.2 HVPA.

The **Horizontal Virtual Pod Autoscaler** (HVPA) is a component, developed in Golang, that carries an associated Service Account which includes the Kubernetes role that authorizes it to use the Kubelet API, in order to create or delete VPods. The list of available VPods can be defined using a YAML configuration file, which contains each **VPod manifest** related to the application. To be more specific, the following aspects are required:

- information needed by Kubelet for the deployment of VPods;
- deployment policies: a VPod can be deployed at user session initialization or based on application logic or events;
- the reference to the microservice code for browser execution with the information about the respective technology (e.g. WASM-GO, WASM-Rust, Python, Javascript, etc.).

To distinguish VPods of different user sessions, a prefix containing the ID associated with the session itself is used. Moreover, in creating VPods, appropriate namespaces are defined to implement the *user session space* and *cluster space* logics.

4.1.3 S-Connector.

The **Server-Connector** (S-Connector) is one of the two components that connects the Client with the Cloud-Edge environment. The communication channel is instantiated using WebSockets, so that there is always an active two-way communication channel.

The first role of the S-Connector is to remain listening to establish connections with clients, thereby instantiating and maintaining the user sessions. A secondary WebSocket channel, on the other hand, is kept active with the HVPA, which is notified of the creation or termination of sessions and any application demand for VPod deployment. When HVPA deploys or removes a VPod(S), the S-Connector sends a message to the client containing all the information useful for starting the VPod(C) process within the browser.

Last but not least, the S-Connector is responsible for acting as a bridge to enable VPods' communications:

- an outgoing communication request from the VPod(C) is received as a message from the WebSocket channel, converted into an HTTP request and sent toward the cluster through VPod(S);
- for in-cluster Pod requests to a VPod(S) on the browser, S-Connector receives the request, converts it into a message, and forwards it into the WebSocket channel.

The S-Connector component is also developed in Golang and scales horizontally based on the number of sessions to be handled.

4.2 Browser components

On the browser, the middleware is activated simply by importing the C-Connector library, creating its class instance, and calling its *init* method, which takes as input the WebSocket address on which S-Connector is listening, as shown in Listing 1.

```
const c_Connector = new C_Connector();
c_Connector.init(serverURL);
```

Listing 1: Initialization of C-Connector component

4.2.1 C-Connector.

The Client-Connector (C-Connector) is the other side of the bridge connecting the Client with the Cloud-Edge environment. The component is developed in Typescript and has the following tasks:

- to start the session with the S-Connector, which returns to it a SessionID to be used for requests;

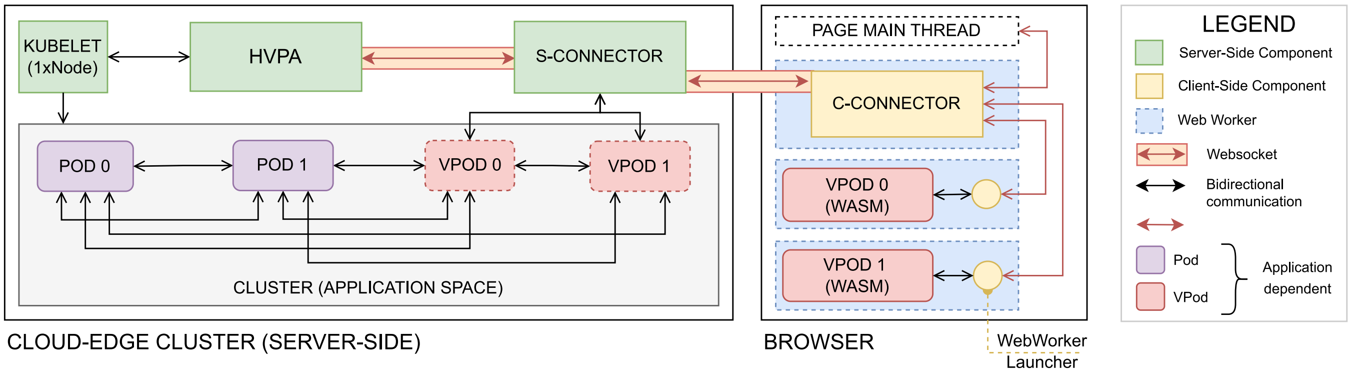

Figure 4: Overall middleware architecture with an example of a generic application consisting of two Pods and two VPods.

- to **manage the lifecycle** of VPods(C) based on the deployment messages sent by S-Connector. The VPod(C) is launched based on the technology used, using the correct WebWorker launcher;
- to **manage communications** in and out of VPods(C). In particular, a standard structure is used to encode requests, which is represented by a Javascript object having the structure shows shown below in Listing 2:

```
type DataMessage = {
  reqType: "httpIn" | "httpInResponse" | "httpOut"
      | "httpOutResponse";
  message: {
    sender: string;
    destination: string;
    path: string;
    sessionId: string;
    requestId: string;
    data: any;
    headers: any;
    params: any;
  };
};
```

Listing 2: Communication message structure

An additional consideration is needed for communication management. In fact, as pointed out in Section 3.1.2, while for incoming communication any functions exposed by the VPod(C) can be invoked by the C-Connector based on messages received, for outgoing communication, on the other hand, the VPod(C) is free to make requests to the Internet. It is easy to see, however, that the services associated with VPod(S) and Pod are not reachable outside the network managed by Kubernetes. To overcome this boundary, C-Connector takes care of automatically intercepting all fetches directed toward one of the Pods or VPods in the cluster, keeps the request on standby using a promise, and asynchronously forwards a message toward C-Connector to reach the destination Pod or VPod. A RequestID is also assigned to each request, so when the C-Connector receives the response, it resolves the promise by providing the response to the requesting VPod(C), closing the loop. In this way, the VPod(C) can make classic HTTP requests even internal to the cluster, since the subsequent handling of the request is totally up to the middleware. It is also noted that, in case the request is directed to another VPod started in the same session, a shortcut is used that does not involve server-side interactions.

4.3 Overall Architecture

Figure 4 shows the complete composition of the middleware providing an example of a generic application deployed, and highlights the interaction between the components.

4.3.1 VPod Deployment.
The deployment flow of a VPod, which is shown in Figure 5, can be summarized in the following steps:

(1) when the user accesses the Web page, S-Connector takes care of initializing a new user session;
(2) HPVA gets notified by S-Connector about the existence of the just newly created user session;
(3) HPVA reads the configuration of VPods and contacts Kubelet to deploy VPods(S);
(4) HPVA sends a notification to S-Connector for each VPod started;
(5) S-Connector forwards the message to C-Connector;
(6) C-Connector starts a new Web Worker for VPod(S).

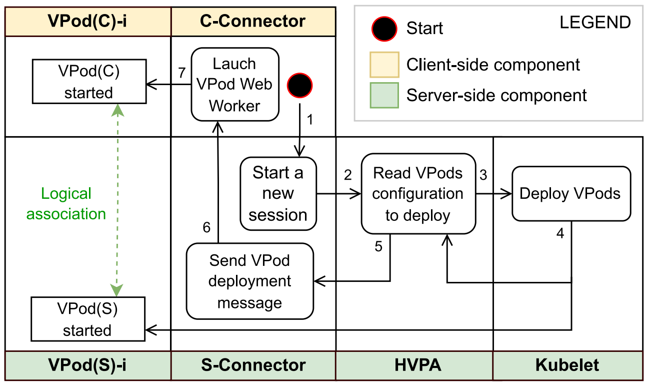

Figure 5: VPod deployment steps

4.3.2 Communications handling.
One of the key aspects of the proposed middleware is communication. The two flows related to incoming and outgoing requests with respect to VPod are described below:

- **VPod to Pod** (outgoing request, Figure 6):
  (1) VPod(C)-i makes an HTTP request to Pod-j;
  (2) C-Connector intercepts the request, creates a pending promise and associates it with a RequestID, encodes the request into a message, and forwards it to S-Connector through the WebSocket channel;
  (3) S-Connector receives the message and makes an HTTP call to VPod(S)-i;

(4) VPod(S)-i acts as a reverse proxy and turns the request back to the recipient, Pod-j;
(5) Pod-j processes the request, which from its perspective comes from VPod(S)-i, and replies;
(6) VPod(S)-i receives the response and forwards it to S-Connector;
(7) S-Connector encodes the response into a message and forwards it to C-Connector;
(8) C-Connector receives the message and resolves the promise associated with RequestID;
(9) VPod(C) receives the response to its HTTP call.

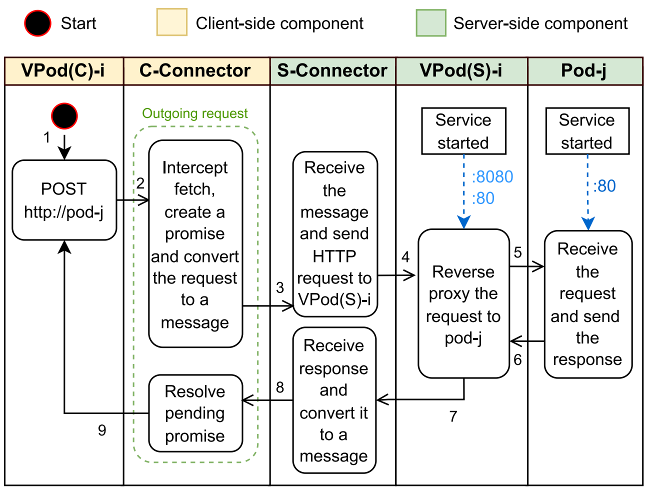
Start
Client-side component
Server-side component
VPod(C)-i
C-Connector
S-Connector
VPod(S)-i
Pod-j
Outgoing request
POST http://pod-j
Intercept fetch, create a promise and convert the request to a message
Receive the message and send HTTP request to VPod(S)-i
Service started
:8080
:80
Reverse proxy the request to pod-j
Receive the request and send the response
Receive response and convert it to a message
Resolve pending promise
1
2
3
4
5
6
7
8
9

Figure 6: Outgoing request diagram: from VPod to Pod

- **Pod to VPod** (incoming request, see Figure 7):

(1) Pod-j makes an HTTP request toward VPod(S)-i;
(2) VPod(S)-i acts as a reverse proxy and turns the request toward S-Connector, inserting the X-Sender and the X-Destination headers;
(3) S-Connector receives the request, assigns it a RequestID, converts it into a message addressed to X-Destination (VPod(C)-i), and sends it into the WebSocket channel associated with the SessionID, leaving the HTTP request momentarily unresolved;
(4) C-Connector reads the message and encodes it into a call to a function exposed by VPod(C)-i;
(5) VPod(C)-i processes the request and provides the response;
(6) C-Connector creates the response message and forwards it to S-Connector;
(7) S-Connector receives the response and resolves the request associated with the RequestID;
(8) VPod(S)-i gets the response and forwards it to Pod-j;
(9) Pod-j receives the response.

5 EXPERIMENTS AND RESULTS

In this Section, we validate the functionality and effectiveness of the implemented solution by conducting specific analyses and providing an example use case. We conducted several tests using the following hardware, over a LAN network:

- a virtual machine for server-side middleware deployment:
  - OS: Debian 12;
  - CPU: *Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz x4*;
  - RAM: *16GB*.
- a laptop as a client for browser usage:

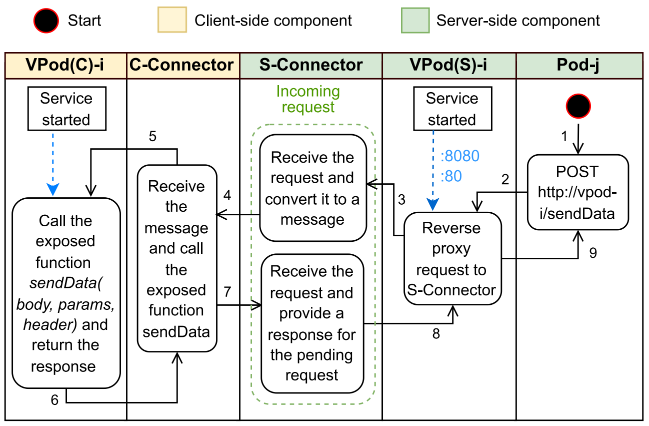
Start
Client-side component
Server-side component
VPod(C)-i
C-Connector
S-Connector
VPod(S)-i
Pod-j
Service started
Call the exposed function sendData(body, params, header) and return the response
Receive the message and call the exposed function sendData
Incoming request
Receive the request and convert it to a message
Receive the request and provide a response for the pending request
Service started
:8080
:80
Reverse proxy request to S-Connector
POST http://vpod-i/sendData
1
2
3
4
5
6
7
8
9

Figure 7: Incoming request diagram: from Pod to VPod

  - OS: Ubuntu 22.04.2 LTS or Windows 11 (dual boot);
  - CPU: *Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz x4*;
  - RAM: *16GB*.
- a smartphone as a client for browser usage:
  - Model: *Realme GT Master Edition (RMX3363)*;
  - Android version: *12*;
  - CPU: *Qualcomm Snapdragon 778G Octa-core 2.40GHz*;
  - RAM: *6GB*.

5.1 VPod deployment analysis

The first metric we analyze concerns the deployment time of a VPod. In this case, the test microservice was a classic *HelloWorld*, developed in Golang and compiled in WASM. The time measured is from the S-Connector request to deploy the VPod until the process starts.

As shown in Figure 8, the tests were carried out comparing the performance of different browsers in two different operating systems, Linux and Windows. In Figure 9, we can observe that most of the time is spent starting VPod(S), which we recall is a real Pod run by Kubernetes. In contrast, the overhead introduced by the Virtual Pod concept is around 25%, netting out any network latency which in this case, being in a local environment, is minimized. We also note that Firefox excels at handling WebWorkers and starting the WASM process.

5.2 Illustrative use case

The distributed application implemented as a use case is intended to highlight mainly two aspects:

- the possibility of offloading resources by exploiting the browser;
- the ability to process sensitive data directly client-side, without affecting the logic of the frontend interface.

The application consists of static analysis on a synthetic dataset representing people's income and consisting of 500,000 records. Specifically, the dataset is composed of the following fields: first name, last name, age, region, credit card, and income.

The computational task is based on an asynchronous workflow that follows a master-slave approach. The operation is managed through the following microservices:

- **anonymizer**: deployed on the client, it takes care of anonymizing sensitive data;
- **master**: gets the anonymized data as input and divides it among 4 workers. As the workers perform the analysis, it aggregates the results;

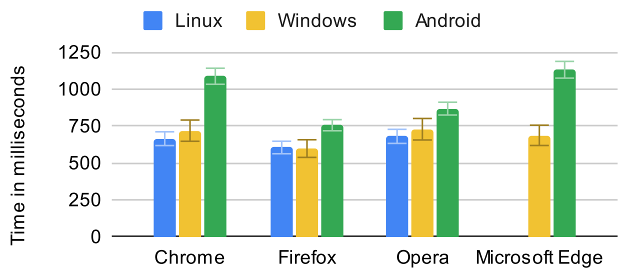

Figure 8: Analysis of deployment time of a VPod in different browsers considering heterogeneous devices

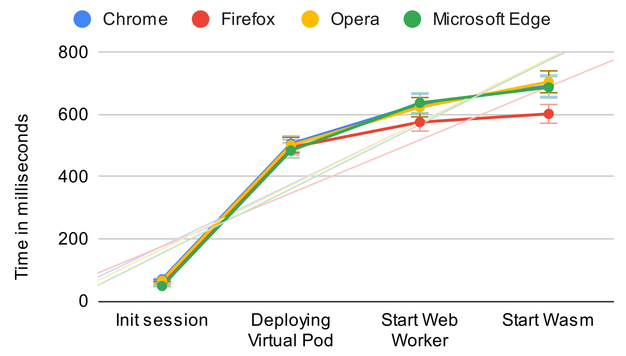

Figure 9: Detailed analysis of the individual steps of the deployment time of a VPod considering different browsers on a Windows laptop

- **worker**: four replicas, each of which receives a portion of data to process, splits it into chunks that it processes and sends individually to the master.

5.2.1 Application behaviour.
Considering the dataset data, we performed tests using the laptop computer with the Linux operating system as the client with different configurations and increased the portion of the dataset provided. Specifically, we kept the anonymizer and master microservices on the browser, while we evaluated the offloading possibilities of the workers in the following cases:

- 4 VPods: all the workers' computation is on the client side.
- 1 Pods and 3 VPod: 25% of workers computation is on the client side;
- 2 Pods and 2 VPods: computation is equally distributed between the client side and the server side;
- 3 Pods and 1 VPod: 25% of workers computation is on the server side;
- 4 Pods: all the workers' computation is on the server side.

It should be noted that the anonymizer and master microservices could also be moved to the server side, but were kept on the client by choice, thus highlighting the possibility of exclusively giving the user control over potentially sensitive data. We recall that on the client we are using resources that for the application infrastructure are nearly zero cost, especially when automating the required computational capacity. We also emphasize that leaving the computation on the client, avoiding data transmission over the network, avoids any network latency overhead that is minimized in this case.

As the results in Figure 10 show, although it appears that Pods perform slightly better than VPods, the final results are highly comparable in all

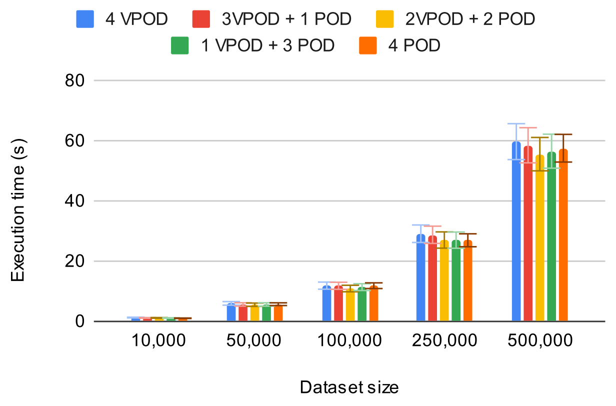

Figure 10: Analysis of the execution time of the use case on different workers' configurations varying the size of the given dataset.

configurations and the best results are obtained when the computation is equally distributed between Pods and VPods. From Figure 11 on the other hand, considering the processing of the larger dataset size, we can see that most of the time is spent on average in the anonymization phase. This can be justified by the fact that this operation was not parallelized and therefore the only VPod on the browser has to preprocess the entire dataset.

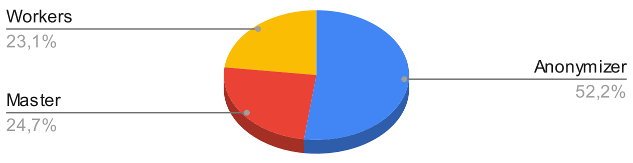

Figure 11: Analysis in percentages of the average division of labour in terms of time spent with a dataset of 500,000 records. The computation of the four workers is considered as a whole

In fact, analyzing the workflow performance in more detail in Figure 12, we note that in addition to the preprocessing phase spent by the anonymizer, much of the time is spent on the data transfer phase from the Web Worker *anonymizer* to the Web Worker *master*. This in fact involves the encoding and decoding of the entire provided dataset that has to be transferred between Web Workers up to the WASM process.

6 CONCLUSIONS AND FUTURE WORKS

The designed architecture and implemented middleware allow deploying a distributed application via Kubernetes on client nodes within the Continuum, leveraging the web browser, without adding any complexity to the release process and without affecting the application logic.

Implementation and testing highlighted the effectiveness of the solution, with prospects for promising future developments. The elimination of the logical separation between client and server represents a focal point for the possible design of future distributed applications, as it allows the developer not to have to worry about the diversity of the two entities. The introduced concept of Virtual Pods, moreover, can also be applied in other contexts in which a device is to be integrated within the Continuum.

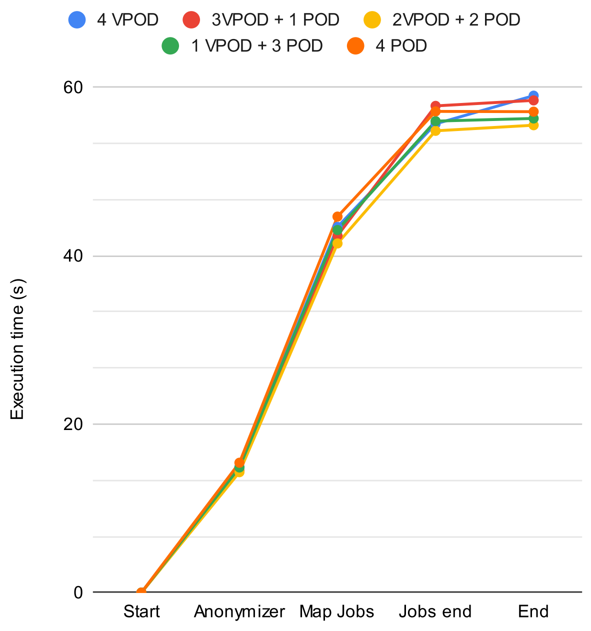

Figure 12: Detailed analysis on workflow's steps of execution time considering the dataset of 500,000 records.

There are plenty of future scenarios that we can analyze and that it will be our interest to investigate:

- the possibility of including mobile applications, in addition to web browsers, to integrate smartphones into pervasive experiences;
- enhancing security, a topic that was deliberately kept out of the scope of the paper, but which is vital for a real application of the solution;
- analyze system performance in more detail, comparing different devices and different programming languages for microservices deployed on the browser;
- apply the concept in real applications that effectively exploit the possibility of having microservices deployed on the client;
- exploit the scenarios opened up by the logical elimination between client and server.

In essence, the new *Cloud-Edge-Client Continuum* concept we proposed paves the way for future Pervasive Computing scenarios, giving users full control over the management of their data and resources.

ACKNOWLEDGMENTS

This work has been partially supported by the Italian Ministry of Health, Piano Operativo Salute (POS) trajectory 2 "eHealth, diagnostica avanzata, medical device e mini invasività" through the project "Rete eHealth: AI e strumenti ICT Innovativi orientati alla Diagnostica Digitale (RAIDD)"(CUP J43C22000380001) and the Horizon Europe TEMA project (Grant Agreement 101093003).

REFERENCES

[1] Anees, T., Habib, Q., Al-Shamayleh, A. S., Khalil, W., Obaidat, M. A., and Akhunzada, A. The integration of wot and edge computing: Issues and challenges. *Sustainability 15*, 7 (2023).

[2] Ayed, D., Dragan, P.-A., Félix, E., Mann, Z. A., Salant, E., Seidl, R., Sidiropoulos, A., Taylor, S., and Vitorino, R. Protecting sensitive data in the cloud-to-edge continuum: The fogprotect approach. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)* (2022), pp. 279–288.

[3] Belcastro, L., Marozzo, F., Orsino, A., Talia, D., and Trunfio, P. Using the compute continuum for data analysis: Edge-cloud integration for urban mobility. In *2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (2023), pp. 338–344.

[4] Bittencourt, L., Immich, R., Sakellariou, R., Fonseca, N., Madeira, E., Curado, M., Villas, L., DaSilva, L., Lee, C., and Rana, O. The internet of things, fog and cloud continuum: Integration and challenges. *Internet of Things 3-4* (oct 2018), 134–155.

[5] Carnevale, L., Ortis, A., Fortino, G., Battiato, S., and Villari, M. From cloud-edge to edge-edge continuum: the swarm-based edge computing systems. In *2022 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)* (2022), pp. 1–6.

[6] Catalfamo, A., Celesti, A., Fazio, M., and Villari, M. A homomorphic encryption service to secure data processing in a cloud/edge continuum context. In *2022 9th International Conference on Future Internet of Things and Cloud (FiCloud)* (2022), pp. 55–61.

[7] De Macedo, J., Abreu, R., Pereira, R., and Saraiva, J. On the runtime and energy performance of webassembly: Is webassembly superior to javascript yet? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)* (2021), pp. 255–262.

[8] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation* (San Francisco, CA, 2004), pp. 137–150.

[9] Dhyani, K., Bhachawat, S., Prabhu, J., and Kumar, M. S. A novel survey on ubiquitous computing. In *Data Intelligence and Cognitive Informatics* (Singapore, 2022), I. J. Jacob, S. Kolandapalayam Shanmugam, and R. Bestak, Eds., Springer Nature Singapore, pp. 109–123.

[10] Duan, J., Duan, J., Wan, X., and Li, Y. Efficient federated learning method for cloud-edge network communication. In *2023 5th International Conference on Communications, Information System and Computer Engineering (CISCE)* (2023), pp. 118–121.

[11] Feng, C., Han, P., Zhang, X., Yang, B., Liu, Y., and Guo, L. Computation offloading in mobile edge computing networks: A survey. *Journal of Network and Computer Applications 202* (2022), 103366.

[12] Firouzi, F., Farahani, B., Panahi, E., and Barzegari, M. Task offloading for edge-fog-cloud interplay in the healthcare internet of things (iot). In *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)* (2021), pp. 1–8.

[13] Gackstatter, P., Frangoudis, P. A., and Dustdar, S. Pushing serverless to the edge with webassembly runtimes. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)* (2022), pp. 140–149.

[14] Goethals, T., De Turck, F., and Volckaert, B. Extending kubernetes clusters to low-resource edge devices using virtual kubelets. *IEEE Transactions on Cloud Computing 10*, 4 (2022), 2623–2636.

[15] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. Bringing the web up to speed with webassembly. *SIGPLAN Not. 52*, 6 (jun 2017), 185–200.

[16] Iorio, M., Risso, F., Palesandro, A., Camiciotti, L., and Manzalini, A. Computing without borders: The way towards liquid computing. *IEEE Transactions on Cloud Computing 11*, 3 (2023), 2820–2838.

[17] Kakati, S., and Brorsson, M. Webassembly beyond the web: A review for the edge-cloud continuum. In *2023 3rd International Conference on Intelligent Technologies (CONIT)* (2023), pp. 1–8.

[18] Kimovski, D., Saurabh, N., Jansen, M., Aral, A., Al-Dulaimy, A., Bondi, A. B., Galletta, A., Papadopoulos, A. V., Iosup, A., and Prodan, R. Beyond von neumann in the computing continuum: Architectures, applications, and future directions. *IEEE Internet Computing* (2023), 1–11.

[19] Matsuo, H., Matsumoto, S., Higo, Y., and Kusumoto, S. Madoop: Improving browser-based volunteer computing based on modern web technologies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2019), pp. 634–638.

[20] Mendki, P. Evaluating webassembly enabled serverless approach for edge computing. In *2020 IEEE Cloud Summit* (2020), pp. 161–166.

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

[22] Moreschini, S., Pecorelli, F., Li, X., Naz, S., Hästbacka, D., and Taibi, D. Cloud continuum: The definition. *IEEE Access 10* (2022), 131876–131886.

[23] Pham, S., Oliveira, K., and Lung, C.-H. Webassembly modules as alternative to docker containers in iot application development. In *2023 IEEE 3rd International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB)*

(2023), pp. 519–524.

[24] Singh, R., Kovacs, J., and Kiss, T. To offload or not? an analysis of big data offloading strategies from edge to cloud. In *2022 IEEE World AI IoT Congress (AIIoT)* (2022), pp. 046–052.

[25] Tushar, and Mohan, B. R. Comparative analysis of javascript and webassembly in the browser environment. In *2022 IEEE 10th Region 10 Humanitarian Technology Conference (R10-HTC)* (2022), pp. 232–237.

[26] Zeng, D., Guo, S., and Cheng, Z. The web of things: A survey. *J. Commun. 6*, 6 (2011), 424–438.