WebAssembly Orchestration in the Context of Serverless Computing

Vojdan Kjorveziroski[1*] and Sonja Filiposka[1†]

[1*]Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University, Rudzer Boshkovikj 16, Skopje, 1000, North Macedonia.

*Corresponding author(s). E-mail(s): vojdan.kjorveziroski@finki.ukim.mk;
Contributing authors: sonja.filiposka@finki.ukim.mk;
†These authors contributed equally to this work.

Abstract

Recent WebAssembly advancements including better programming language support and the introduction of both the WebAssembly System Interface, and the WebAssembly Component Model, have transformed it from primarily a client-side technology to a server-side one as well. The advantages associated with WebAssembly, such as cross platform portability, small software artifacts sizes, fast start up times, and per execution isolation make it a good fit for serverless scenarios. While there are existing initiatives for using WebAssembly in such serverless contexts, orchestration is still an open question. To overcome this issue, we present a way for extending Kubernetes, allowing it to orchestrate natively executed WebAssembly modules, in addition to containers. We describe an extension to an existing WebAssembly software shim for containerd and a new Kubernetes WebAssembly operator. Benchmarking results from the proposed solution obtained using 9 serverless functions packaged both as WebAssembly modules and OpenFaaS functions running in containers, show that WebAssembly has clear advantages for frequently executed serverless functions which require elasticity. WebAssembly functions enjoy two times faster deployment times and at least an order of magnitude smaller artifact sizes while still offering comparable execution performance. However, when it comes to sustained performance for long running serverless functions with processor intensive workloads, containers are the preferred choice, compensating for the increased cold start times with faster execution times.

Keywords: WebAssembly, Serverless Computing, Function as a Service, Kubernetes, Orchestration

1 Introduction

The serverless computing paradigm has experienced a steadfast growth in popularity and adoption since its inception [1]. Formally defined as a symbiosis between Function as a Service (FaaS) and Backend as a Service (BaaS) offerings [2], it provides developers with high levels of abstraction in terms of the underlying hardware where their software is executed, allowing them to focus on writing the programming logic itself, instead of managing the infrastructure. The distinct benefits of serverless computing compared to other "as a Service" products such as reduced development times, potentially lower infrastructure costs through granular billing policies, effortless vertical scaling, and an overall simplified development process [3], make it a feasible and attractive technology not only for web services running in the cloud, but for a plethora of other applications as well. Domains where serverless computing has already been successfully used include event-driven workloads commonly associated with Internet of Things (IoT) use-cases [4, 5], implementation of the Multi-Access Edge Computing (MEC) specification [6–8], High Performance Computing (HPC) [9], and the handling of Augmented Reality/Virtual Reality (AR/VR) workloads [10].

While the overall concept of serverless computing is versatile, when looking into the details of each use-case it becomes evident that there is no single blueprint which can be applied to every scenario. Different applications require different context specific optimizations and decisions, each with their own benefits and drawbacks. These choices can have a wide ranging impact on start-up times, security, performance and cost, both from the users' and providers' perspective. One such impactful aspect which is currently of great interest to the research community is the choice of the runtime environment for executing the serverless functions [11]. The most widely used options today, both by commercial and open-source providers, are containers and micro virtual machines. Containers, made popular by the advent of Docker, but defined long before that, allow a reproducible and to an extent isolated environment in which (un)trusted functions can be run in a multi-tenant fashion. The large number of container runtimes available today means that they can be used on various hardware platforms. Micro virtual machines, on the other hand, offer greater isolation, while also being more lightweight and faster to start than full-fledged, traditional, virtual machines [12–14]. Despite this, micro virtual machines are not as wide-spread as containers partly due to the lack of a comprehensive, user-friendly ecosystem of tools and applications.

Even though both containers and micro virtual machines are well established serverless runtime options today and already integrated by commercial and open-source serverless platforms alike, there are still a number of open

issues which prevent their wider deployment for event-triggered, time sensitive, workloads which require high levels of elasticity, small cold start delay, and per invocation isolation of functions [15]. A cold start delay or cold start time, in serverless terminology, is the time it takes to prepare a function instance for execution, once a request is received [16]. Existing runtime technologies, despite optimizations [17–19], still suffer from large cold start times, mainly because they were designed around the idea of long running workloads, which is contrary to the serverless paradigm. As a result of this, per invocation isolation of function executions is also very costly to implement, with serverless platforms instead resorting to reusing function instances across multiple executions, thus foregoing isolation between runs. The same issues apply to rapidly scaling a given function either up or down, since the cold start delay applies to each new instance [20]. This leads to serverless platforms keeping active instances "warm", leaving them running for a period of time in case they are required to serve requests again, which also leads to increased costs and unnecessary resource usage.

A potential solution to the previously described challenges affecting serverless computing is the introduction of WebAssembly (WASM) as a runtime environment where functions would be executed [21]. WebAssembly, at its core, is a binary instruction format first envisioned as a client-side technology [22], but which has recently seen greater adoption in server-side scenarios as well [23]. Server-side applications of WebAssembly have been made possible by the definition of the WebAssembly System Interface (WASI) [24], which allow it to leverage the underlying operating system's functionality for interfacing with the hardware and thus handle aspects such as file and network Input/Output Operations (I/O) [25, 26], in a similar fashion as to what is the POSIX interface. Some of the benefits that WebAssembly could introduce to serverless computing are drastically reduced cold start times, per invocation isolation of functions, and compatibility with existing and widely accepted programming languages by the developer community. While server-side WebAssembly runtimes already exist, the open issue of WebAssembly orchestration needs to be tackled before WebAssembly can be seen as a first-class serverless runtime, similar to what are containers and micro virtual machines today.

The goal of this paper is to propose and evaluate a concrete approach with which WebAssembly modules can be orchestrated at scale in serverless scenarios. We focus our effort on extending Kubernetes, the popular container orchestrator, allowing it to be used not only for container management, but also for management of WASM modules. This makes it possible to design hybrid serverless platforms which leverage multiple runtime environments, selecting the most optimal one for a given scenario. To this effect, the main contributions of this work are:

- Develop a Kubernetes operator which allows the scheduling and running of WebAssembly serverless functions in a Kubernetes environment;
- Extend a WebAssembly software shim for the containerd container runtime, making it possible to run both WebAssembly and more traditional,

containerized, serverless functions using the same uniform Command Line (CLI) and Application Programming (API) Interfaces;
- Benchmark the proposed solution using 9 serverless functions in 4 different scenarios, comparing its performance to the open-source OpenFaaS serverless platform which is using containers as the underlying runtime environment;
- Describe a way in which the Kubernetes orchestrator can be extended to support high resolution execution metrics, not limited to a resolution of one second, beneficial for future research dealing with this orchestrator.

All of the scripts, software, software modifications and results presented in this paper have been open-sourced under a permissive license and are freely available for reuse. The rest of the paper is organized as follows: in Section 2 we describe the state-of-the-art research related to WebAssembly as a runtime environment for execution of serverless functions. We then continue with Section 3 where we outline the used methodology for modifying the Kubernetes orchestrator to support WebAssembly modules, discuss the benchmarking functions, the motivation behind the execution scenarios and the execution strategies themselves. In Section 4 we present the obtained results, comparing the performance of the same set of serverless functions executed as WebAssembly modules and traditional container-based serverless functions in the same environment managed by Kubernetes. We finish the paper with Section 5, summarizing the obtained results, drawing conclusions, and outlining ideas for future research.

2 Related Work

With the introduction of the WebAssembly System Interface [27], the WebAssembly ecosystem has made its first steps towards becoming a viable technology for execution of server-side workloads. All of the WASI APIs are developed as community proposals [24], allowing anyone to asses the current implementation progress and contribute by themselves. WebAssembly's inherent advantages in terms of portability, fast instantiation and security, have allowed it to be identified as a potential runtime environment for execution of serverless functions [21, 23], despite its lack of feature parity with more mature alternatives. One such feature which can be beneficial in a serverless context, but is currently in the design phase is the support for multi-threading [28].

Proposals [29], along with actual serverless platform implementations [30] which leverage WebAssembly as their runtime environment have already been published, confirming the expected improvements to the cold start delays. Existing work done by Long et al., who have analyzed the cold start delays between containers and WASM modules shows that WebAssembly workloads offer up to 10 times faster start up times [31]. However, actual execution performance, measured after the environment has been set up is lower for WebAssembly modules compared to both native and containerized execution [32, 33]. These performance differences can be attributed partly to the

missing WebAssembly features, as well as to the introduction of additional intermediary layers between the hardware and the actual code being executed.

Despite the fact that WebAssembly's adoption for serverless scenarios brings numerous advantages which are not possible with current runtime environments [34], existing shortcomings primarily in the area of sustained execution performance, further underline the importance of choosing the right runtime environment depending on the nature of the serverless function being executed. Taking into account that WebAssembly workload orchestration is still an open question, this raises the possibility of reusing existing orchestration engines aimed at other runtime technologies and extending them to work with WASM modules as well. As a starting point for this effort, previous work focusing on optimizing orchestration engines for efficiently running serverless workloads even in resource constrained environments such as the edge can be used [35]. Goethals et al. [36] have also recently showcased a strategy for extending Kubernetes clusters to low-resource devices.

Sebrechts et al. [37], with the same aim of reducing the resource footprint of Kubernetes, have proposed a strategy for using Kubernetes operators executed as WebAssembly modules. A Kubernetes operator is a software component running inside the Kubernetes cluster which has direct interaction with the API server, and is responsible for maintaining the desired state of a certain aspect of the cluster [38]. However, to the best of our knowledge, there is no existing work which has extended Kubernetes as to allow it to orchestrate actual WebAssembly workloads in the same way that is currently possible for containers. Our aim is to fill this gap, building on top of previous work done in this area in terms of performance evaluation of Kubernetes distributions [35] and integration of WebAssembly with the CLI and API of container runtimes [39].

3 Methodology

To achieve the end-goal of a unified environment where containers and WebAssembly modules could be executed side-by-side and orchestrated by the same instance of the Kubernetes orchestrator, a number of steps were required, all of which are discussed in detail in this section. We first focus our attention on extending the existing containerd container runtime with support for WASM modules through the use of software shims, a process described in Subsection 3.1. Once containerd can be used as the underlying container runtime by Kubernetes, Subsction 3.2 describes the Kubernetes operator which we have developed that is capable of leveraging the modified version of containerd in order to orchestrate WASM modules. To test the performance of the proposed platform, we have extended an existing set of WebAssembly serveless benchmark functions and also adapted them for execution in a containerized environment. Details about these functions, along with an evaluation of the conversion process between a WebAssembly module and a containerized version of the same serverless function compatible with a popular serverless

platform is provided in Subsection 3.3. We conclude this section with Subsection 3.4 where we describe the execution strategy of the selected benchmarking functions across two different environments.

3.1 Extending Container Runtimes

The evolution of the complex ecosystem of container engines, container runtimes, and container orchestrators has resulted in the development of modular software which can be easily extended and made interoperable with third-party components. Containerd [40] is one such modular container runtime whose functionality can be extended through the use of software shims [41]. This feature makes it possible to reuse the existing CLI and API of containerd for managing other environments, not just containers. Recognizing this versatility, a number of containerd shims have been developed, including ones that provide the ability for containerd to interface with a WebAssembly runtime [42]. A detailed description and comparison between such shims is available in [39]. To realize the goal of Kubernetes orchestrated WebAssembly modules, we have chosen to base our work on the open-source containerd Spin shim [43].

Spin [44] is a WebAssembly runtime environment based on the popular Wasmtime runtime [29, 45], that also provides additional tooling, making the process of developing and running WebAssembly modules easier. It has support for a number of programming languages, including Rust and Go [46]. Apart from support for WASI, it also implements additional features which make it a viable solution for use in serverless scenarios. Spin allows remote invocation of WebAssembly modules via HTTP or Redis triggers, without requiring the necessary programming logic for doing so to be manually implemented by the developer into every function. This works in one of two ways, depending on the source programming language used for developing the WASM module. In the case of Rust, it leverages the WebAssembly component model, which allows WASM module reuse in a similar fashion to shared libraries [47]. For all other supported languages apart from Rust it uses an intermediary component called WebAssembly Gateway Interface (WAGI) [48] which acts as a Common Gateway Interface (CGI) proxy. It listens for incoming requests, and passes them serialized as standard input to the WASM module. Everything that is printed on the standard output during the module's execution is serialized and sent as a response to the invoking party. Apart from allowing faster and simpler development of serverless function, this integrated invocation strategy also has the potential to even further reduce the file size of WASM modules, thus easing their distribution.

Since containerd is used as an interface for the Spin runtime, the distribution medium for the WebAssembly modules, in this case, are Open Container Initiative (OCI) images. Such OCI images containing WASM binaries are created from a "scratch" layer, and contain only the WASM module itself along with a structured Spin file describing the runtime behavior, for example the invocation URL. This ensures that the resulting OCI image is as small as possible, not bundling any additional dependencies, omitting the need to

use more complex base images, as is usually the case with serverless platforms which leverage containers as their runtime environment. Additionally, by reusing the OCI format, it is also possible to make use of all existing container image registries for a simplified and uniform distribution of WASM serverless functions.

It is widely accepted that to exploit the benefits of serverless functions to their full extent, integration with external systems needs to be possible. This behavior is even explicitly specified in the formal definition of serverless computing, as being comprised of FaaS and BaaS. Even though WebAssembly has limited support for network sockets, Spin allows outbound communication from the WebAssembly modules either via HTTP requests or a Redis queue. With this feature, serverless functions instantiated as WebAssembly modules have a feature parity with their container based counterparts, allowing them to make use of the same BaaS products for persistence, reporting, authentication, and authorization.

Outbound network communication while undoubtedly beneficial, and required for real-world use-case scenarios, complicates the integration with Kubernetes. In a Kubernetes cluster, the container networking is handled by a Container Network Interface (CNI) plugin [49]. To allow outbound communication while running in a Kubernetes cluster, bypassing the CNI plugin, the WebAssembly module can be run in the host's network namespace. Since Spin allows outbound network communication only towards endpoints which have been explicitly whitelisted during the WebAssembly's module deployment, this does not pose an increased security risk. Sharing the host's network namespace also does not impact the invocation mechanism for the given WASM module, since it can be exposed using a standard Kubernetes service object. The purpose of such a service object is to declare the listening network port of the respective Spin shim responsible for serving the module, allowing other Kubernetes components to interact with it. More details regarding the shim's extension and the resulting interface for communicating between Shim and the container orchestrator are provided in the next subsection, where we also discuss the overall architecture of the Kubernetes operator.

In conclusion, Wasmtime's proven performance [39], together with the bundled functionality for invoking and communicating with external resources were the deciding factors in selecting Spin as the underlying environment for executing serverless function.

3.2 WebAssembly Support in Kubernetes

To allow the seamless deployment of WASM modules in a Kubernetes cluster, and to mimic the instantiation process that is already in place for container based workloads, we have created a dedicated Kubernetes WASM operator. A Kubernetes operator is a third-party component which has access to the Kubernetes API server, can monitor the occurrence of certain events which are of interest, and can react based on those events, reconciling the current state of the cluster with the desired state as defined in the API server.

The Kubernetes operator defines a new Custom Resource Definition (CRD), WasmApp, which is used for describing WebAssembly workloads. This introduces an additional abstraction layer to the deployment of WASM modules in a Kubernetes environment, consolidating information usually scattered around multiple resources (Deployment, Ingress, Service) into a single one, omitting boilerplate statements, allowing easier reusability.

The use of Custom Resources (CR), including WasmApp in this case, is compatible with existing Kubernetes tools, and can easily be integrated even into existing platforms running on top of Kubernetes. For example, WasmApps can be deployed using the Helm Kubernetes application manager [50], independently or as part of a larger Helm release, even one which is comprised of additional containerized applications as well.

The Kubernetes operator depends on the presence of the modified version of the Spin containerd shim. The underlying operations executed during a WasmApp deployment are visualized in Fig. 1, and can be summarized as follows:

- A new WasmApp Kubernetes custom resource is created in the cluster.
- An event is generated, triggering the Kubernetes operator's reconciliation function. The operator validates the passed parameters in the WasmApp manifest, and if successful, instantiates the necessarsy built-in Kubernetes resources. Each WasmApp, at a minimum, is backed by the following resources: a Deployment, specifying the OCI image, a custom Kubernetes RuntimeClass, and exposed ports; a Kubernetes Service; an Ingress resource which configures an existing ingress controller assumed to be already available in the cluster.
- Once the Deployment is created, it triggers the instantiation of a Pod which is scheduled on a selected node by the Kubernetes scheduler. The local Kubelet running at the node, based on the associated RuntimeClass, invokes the modified version of the Spin shim. It binds the HTTP listener on a randomly selected port which is shared with the Kubernetes operator via a Redis instance.
- Once the Pod representing the WASM module enters a ready state, the Kubernetes operator queries the Redis instance, acquires the value for the random port, and makes the necessary corrections to the Service and Ingress objects, so that traffic can pass towards it.

A similar reconciliation process is done for each change and deletion of a WasmApp resource, adjusting the behavior accordingly. It should be noted that a Kubernetes version equal or greater than 1.20 is required for running the operator, since the required RuntimeClass resource which selects what containerd shim to use for a given Pod is available starting from that version [51].

It is worth pointing out that when executing WASM modules, there is per execution isolation, since the runtime environment is not reused across multiple invocations. After a response is returned, the environment is torn down

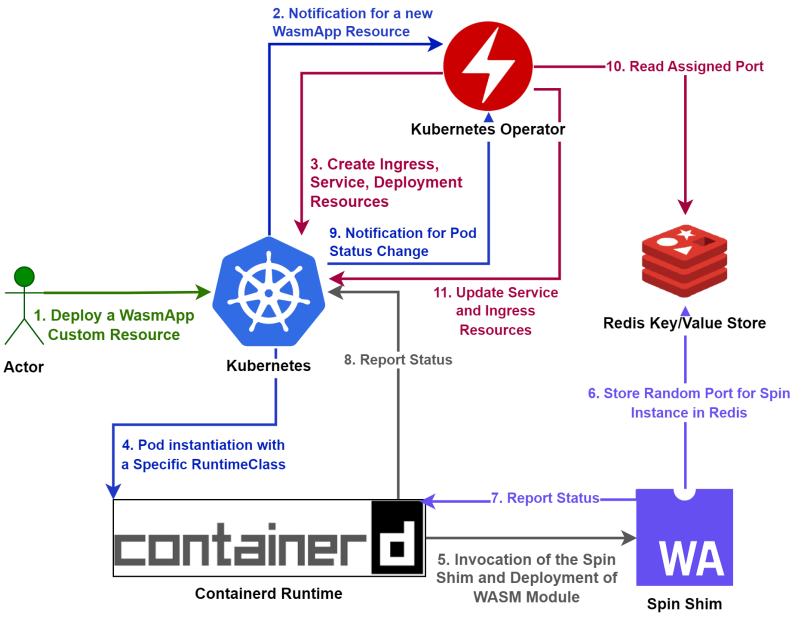

Fig. 1 Deployment Process for a new WASM Serverless Function

and instantiated from scratch for the next request. This behaviour not only increases the overall security and isolation between functions, but also reduces their resource footprint, offering true scale to zero behavior when the functions are not actively being invoked. The described practice can also potentially be useful in edge computing scenarios, allowing overprovisioning of functions in case they do not need to be executed concurrently, thus reducing hardware costs and energy usage.

3.3 Benchmarking Functions and Serverless Platform Selection

To evaluate the performance of the proposed solution and to compare it with an existing container oriented serverless platform, we have extended an existing set of serverless benchmarking functions [39, 52]. During the development of those functions, attention was paid to only use libraries which could be compiled as WebAssembly modules. This is an important point to make since, as discussed previously, WebAssembly currently does not have feature parity with traditional execution environments, and lacks support for some commonly used functionalities, such as multi-threading. However, a number of proposals are underway in various stages of implementation and adoption [53] and at this point in time, the core functionality required for developing and compiling serverless functions to WASM is present. Taking into account the modular architecture of the proposed solution, the WebAssembly runtimes can

be replaced with new, more feature rich versions once they are available, preserving the compatibility with the rest of the tools, including the container orchestrator.

Table 1 describes the serverless functions which have been used for benchmarking in more details.

Table 1 Description of the serverless benchmarking suite

Name	P. Lang.[1]	Description	Type[2]
aes	Go	Encrypt a given text using AES symmetrical encryption multiple times	R
float-operation	Go	Calculate sin, cos and square root of an arbitrary number	M
fuzzysearch	Rust	Find all occurrences of a given phrase in an online text	R
linear-equations	Go	Solve a system of linear equations	M
matmul	Go	Square matrix multiplication	M
n-body	Rust	Model the orbits of Jovian planets	M
prime-numbers	Rust	Search for prime numbers among the first n numbers	M
whatlang	Rust	Determine the natural language of a given string	R
user-manager	Rust	Interact with a BaaS database over a network	R

[1]Source programming language of the given benchmarking function

[2]M – Microbenchmark; R – Real-world workload

The benchmarking suite is comprised of 5 microbenchmarks and 4 real-world scenarios for a total of 9 functions. The microbenchmarks are targeted at evaluating the raw execution performance of the tested platform, while real-world scenarios are focused on modeling a plausible real-world use-case for which a serverless platform might be leveraged. The user-manager function has been added to the benchmarking suite with the aim of evaluating the HTTP outbound capabilities of Spin, and to test whether indeed it is possible to interact with external BaaS services from WebAssembly modules. Upon each execution, the function sends an HTTP POST request to a database as a service, thus persisting a new record in an SQL database.

All 9 functions were compiled both as WebAssembly modules using the default optimization options of the bundled WASM compiler in the Spin toolchain, as well as native binaries using the default options of the Rust and Go compilers, respectively. No changes were made to the source code of the functions to accommodate for the different targets, apart from replacing the Spin HTTP library with its respective native implementation when targeting

the x86 platform. It should be noted though that this operation was only limited to altering the dependencies list, since the Spin libraries follow the same API as their native implementations.

Once binaries were obtained for the two target platforms, OCI images were created. In the case of WebAssembly and Spin, the container image was derived from a "scratch" layer onto which only two files were copied, the WASM binary and the manifest file required for execution of any Spin module. For execution in a containerized environment, the official Rust and Go function templates were used, as recommended by the OpenFaaS project [54, 55].

We have selected the open-source version of the OpenFaaS [56] as a reference implementation of a container-based serverless platform. Even though there are other alternatives in this space as well [57–59], OpenFaaS is one of the most popular ones, is undergoing an active development, has an easy deployment process on an existing Kubernetes cluster without requiring an extensive set of dependencies, and offers function templates for our chosen programming languages. It should be noted that our goal was not to evaluate the performance of the OpenFaaS or any other established serverless platform, but instead to simply use it to obtain a baseline in terms of cold start and execution times for container based execution of serverless functions.

3.4 Benchmark Execution Strategy

A total of 4 different execution scenarios were defined for each of the two evaluated environments:

- Serial execution – each function instance is invoked continuously in a time frame of 5 minutes using only a single thread.
- Function instantiation time and first execution delay – each function is redeployed 100 times, measuring the time for it to become ready to serve new requests after it has been removed. Once ready, the function is invoked, measuring the response time of its first invocation.
- Concurrent execution – each function instance is continuously invoked for 1 minute by 5 threads, each limited to a maximum of 5 requests per second. In the most optimal scenario, this translates to a maximum of 25 requests per second for 1 minute, or a total of 1500 requests in the given time frame.
- Image size evaluation – comparison of the OCI image sizes hosting WASM modules and native binaries.

All of the tests were performed in the same environment, using a set of 4 identical bare-metal machines. Each machine had a dedicated role in the experiments, as follows: 1 Kubernetes master node; 2 Kubernetes worker nodes; 1 helper node not part of the Kubernetes cluster from which the benchmarks were executed. The hey [60] benchmarking tool was used for conducting the experiments. The machines were all part of the same local area network (LAN), interconnected by a 1Gbit switch. The OCI images for the 18 functions in total (9 WASM and 9 container-based) were all preloaded on the worker nodes, thus eliminating any unexpected impacts on the obtained results due to network

issues and congestion which could not be easily reproducible. The hardware characteristics of the bare-metal nodes, as well as the release versions of the major software components which were utilized are provided in Table 2.

Table 2 Details about the execution environment

Component	Description
CPU	Intel Xeon x5647, 2.93GHz, 4 cores
Storage	320GB mechanical hard drive
Memory	8GB DDR3
Operating System	Ubuntu 22.04.1 LTS
Containerd Version	v1.5.16
Kubernetes version	1.22.17
K3s version	v1.22.17+k3s1
Spin version	0.6

We have opted to use K3s [61] as the Kubernetes distribution of choice due to its lightweight design, easy installation, and easy customizability. K3s bundles all of the necessary Kubernetes components including the kubelet, kube-controller-manager, kube-scheduler, and the Kubernetes API in a single binary. In order to support more precise time-related measurements, modifications to the source code of the K3s distribution were required. By default, Kubernetes exposes information about timestamps related to container scheduling, container initialization, and container start up times up to a resolution of a second. This was seen as a limiting factor during the execution of the benchmarks and we have mitigated it by modifying the logging logic of the K3s binary. At each event which was of interest, such as container schedule time, container initialization time, container start time, container ready time, we have logged the exact timestamp, up to a microsecond precision. By cross-referencing these logs with the unique Pod names obtained from the Kubernetes API, we were able to get much more precise measurements regarding the pod lifecycle events both in the case of running WASM workloads using the Spin shim, as well as when executing native containers in the case of OpenFaaS.

To limit external network traffic, required services such as object storage and database as a service were implemented in the same LAN as Docker containers running on a dedicated host not part of the described Kubernetes cluster. The object storage service required for the execution of the whatlang function was mocked using an nginx web server configured to serve the required set of files, while the database as a service was implemented using a PostgreSQL instance that had a REST API exposed using the pRest open-source project [62].

4 Results

The obtained results from each of the 4 different testing strategies are discussed in details in the subsections that follow. We start by comparing the serial execution times between WebAssembly and native containers created by the OpenFaaS serverless platform. We then focus on the function instantiation times, before moving forward with evaluating the performance characteristics during multiple concurrent executions of the same function. We finish this section with a comparative analysis of the OCI image sizes for each test between the two execution scenarios.

4.1 Serial Execution

To evaluate the serial execution performance of WebAssembly modules deployed within a Kubernetes cluster and to compare it with the results obtained from the OpenFaaS platform, all 9 functions were executed for 5 minutes using a single thread, thus limiting the number of concurrent requests to 1. Reuse of TCP connections among requests was explicitly disabled, as to not skew the results. It should be noted that this execution scenario is less than optimal for WASM and it is expected that containerized execution will offer better results, since it has better sustained performance. In this test, the main drawback of using containerization, which is the large cold-start delay, is eliminated because the container is already running when the request is issued. On the other hand, in the case of Spin and WebAssembly, the function instance is started once the request is received, the result is computed, and the instance is brought down, offering true per invocation isolation. Despite this, with the inclusion of this test we hope to provide additional clarification in terms of the performance differences between a warm container instance and WebAssembly, which together with the rest of the tests should paint a clearer picture whether the potential benefits obtained by using WebAssembly in terms of reduced cold start times are offset by the increased execution times. Figure 2 presents the mean response times per function for both Spin and OpenFaaS.

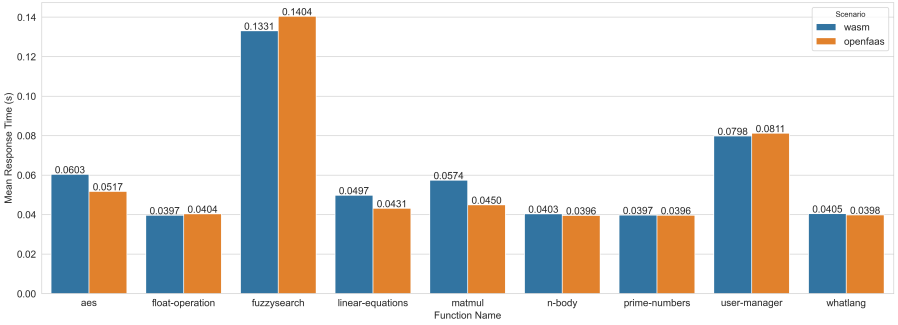

Fig. 2 Comparison of Function Mean Response Times in Seconds between WASM (Spin) and OpenFaaS

Judging by the mean response times, OpenFaaS offers faster execution in 6 out of 9 tests. WebAssembly modules are faster in the float-operation, fuzzysearch, and user-manager tests. Interestingly, the two functions with outbound HTTP capability, fuzzysearch and user-manager, have a better performance as WebAssembly modules than native containers. Even though in the majority of the cases the results are comparable, the highest difference in the mean response times is exhibited for the aes and matmul functions. OpenFaaS offers 16.63% faster execution in the case of the aes benchmark and 27.56% in the case of matmul. This result is expected, taking into account that they are both processor intensive workloads.

Figure 3 plots all obtained response times across all executions with the aim of evaluating the performance consistency of the two scenarios.

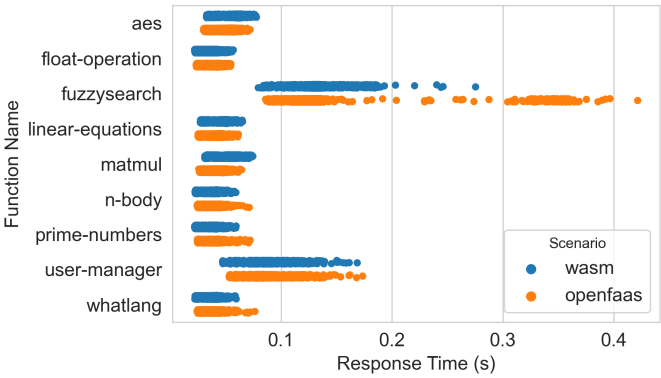

Fig. 3 Response Times in Seconds Across Different Serial Execution Runs

OpenFaaS exhibits less consistent execution performance, characterized by higher unpredictability. Table 3 summarizes the standard deviation and coeficient of variation for the serial execution response times across all functions.

As shown by the table above, across all 9 tests, WASM exhibits lower standard deviation and coefficient of variation for the function response times. In light of these results, it is worth mentioning that the OpenFaaS serverless platform stack is more complex than the WebAssembly stack, since it requires multiple additional components [63], including an intermediary gateway through which all requests are proxied, as well as monitoring agents running on each node, helping with the scaling. The execution of these components introduces additional nondeterministic load which might account for the more variable execution times. In the case of WebAssembly, the only required component is the Spin shim.

To verify the statistical significance of the execution results for WASM and OpenFaaS, we used a nonparametric Mann-Whitney U test, with the following two hypotheses and an alpha value of 0.05:

Table 3 Comparison of Standard Deviation and Coefficient of Variation for Serial Executions

Function	SD[1] WASM	SD[1] OF[2]	CV[3] WASM	CV[3] OF[2]
aes	0.0055	0.0068	0.0907	0.1312
float-operation	0.0045	0.0052	0.1124	0.1286
fuzzysearch	0.0137	0.0606	0.1029	0.4317
linear-equations	0.0047	0.0054	0.0935	0.1253
matmul	0.0051	0.0057	0.0885	0.126
n-body	0.0044	0.0057	0.1093	0.1451
prime-numbers	0.0044	0.0058	0.1099	0.1463
user-manager	0.0121	0.0131	0.1522	0.1608
whatlang	0.0044	0.0059	0.1083	0.1469

[1]Standard deviation

[2]OpenFaaS

[3]Coefficient of variation

- H0: the two populations are equal
- H1: the two populations are not equal

In all 9 cases evaluating the results from the different function executions, the obtained p-value was smaller than 0.05, leading to the rejection of the null hypothesis, and thus concluding the statistical significance of the results.

4.2 Function Instantiation

The time to instantiate a new instance of a given function is a very important metric in serverless scenarios, especially in use-cases where a function needs to be regularly migrated to different nodes (e.g., multi access edge computing) or needs to scale up and down to meet a varying rate of invocations. To better evaluate the total function instantiation time, we performed 100 deployments of each function both for WASM and OpenFaaS, measuring the time between when the Kubernetes API reported the pod as successfully scheduled and the time it was started on the selected node. Fig. 4 shows the mean instantiation times for all 9 functions across the two evaluated scenarios.

Fast deployments are one of the main advantages of WebAssembly, and this is confirmed with the obtained results as well. WASM modules consistently have around 2 times faster instantiation times than their container counterparts in the OpenFaaS case. This can be attributed to the more lightweight software artifacts when it comes to WASM, discussed in more details in subsection 4.4 below. The smaller number of OCI image layers for WASM modules not only conserves storage capacity, but also allows faster instantiation due to the reduced time required for extracting the image archive. It should be noted that the mean times presented in Fig. 4 do not include the response time of the first invocation. To evaluate whether the first invocation after instantiation offers worse results compared to sustained serial execution, we executed once each function after deployment, before deleting it and repeating the instantiation process. Fig. 5 shows the mean response times of the 9 functions being

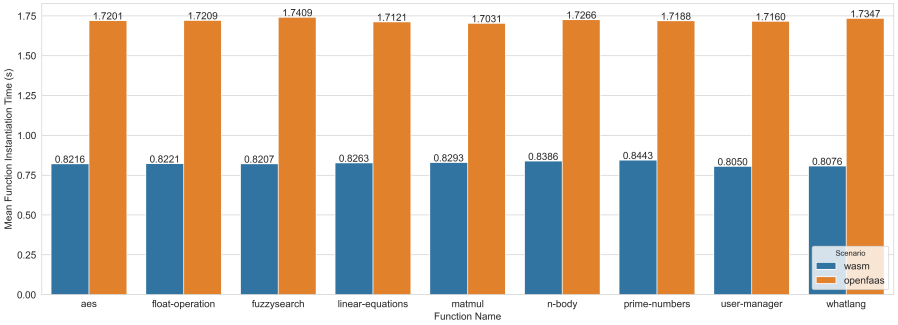

Fig. 4 Comparison of Function Mean Instantiation Times in Seconds between WASM (Spin) and OpenFaaS

invoked immediately after their instantiation, once the pod has entered a ready state.

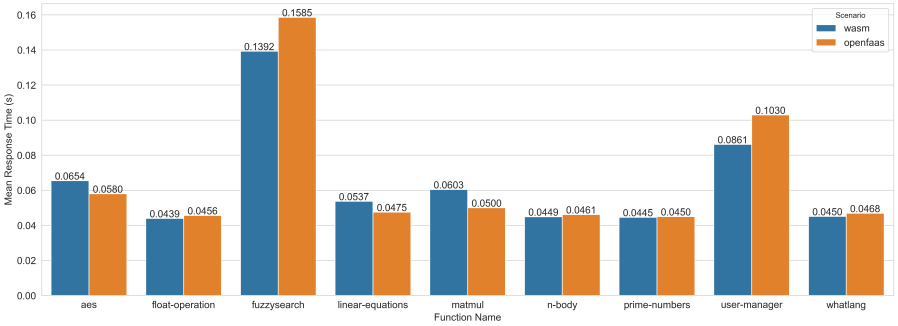

Fig. 5 Comparison of Function Mean Response Times for First Invocation after Instantiation

The response times during a first invocation are higher, albeit to a small extent, compared to those obtained during sustained serial execution, which were discussed in details in Subsection 4.1. This is somewhat expected with one potential reason being the caching behavior of the underlying operating system. Interestingly, the time differences for the two functions with outbound HTTP functionality, fuzzysearch and user-manager, has increased more substantially between WASM sustained serial execution, with WASM still being faster. Additionally, WASM exhibits better performance in two additional tests, n-body and prime-numbers, which was not the case previously. Both of these functions were invoked with the same input parameters across all executions, allowing the long-lived OpenFaaS container to have more efficiently cached intermediary results across serial executions, previously. Since in this scenario a new container is created from scratch for each invocation, this advantage is no longer present.

To test the statistical significance of the obtained results, we repeated the same nonparametric Mann-Whitney U test described in the subsection evaluating the serial execution peformance, reusing the hypotheses with the same alpha value of 0.05:

- H0: the two populations are equal
- H1: the two populations are not equal

In all cases the obtained p-value was significantly smaller than alpha, leading to the rejection of the null hypothesis.

In conclusion, WASM instantiation times are two times faster across the board for all functions, while offering comparable execution performance, especially for simpler workloads which are not CPU bound.

4.3 Concurrent Execution

To test the performance of the two execution environments under a sustained concurrent load, each function was executed for a duration of 1 minute using 5 threads, each limited to 5 requests per second. In the most optimal case, when the rate of incoming requests is less than the response time of a given function (and thus no queuing is required), this translates to 1500 requests in the time frame of 60 seconds. No automatic scaling strategies were applied for OpenFaaS, and the default behavior of the Spin shim was utilized. As discussed previously, Spin instantiates the WASM module upon each request and shuts it down once processing is complete. Translating this behavior to the described scenario, the 1500 requests per function are handled by 1500 different instances of the same WASM module, offering per request isolation. In the case of OpenFaaS, a single long running container with an embedded HTTP server is handling all the incoming requests.

Fig. 6 shows the results obtained during concurrent execution of the 9 functions for each of the evaluated scenarios.

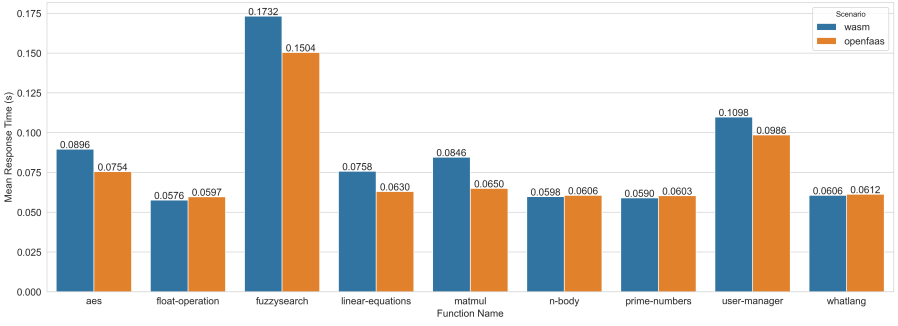

Fig. 6 Comparison of Mean Response Time during a Concurrent Execution between WASM (Spin) and OpenFaaS

Analyzing the figure, WASM shows lower mean response time for 4 functions: float-operation, n-body, prime-numbers, and whatlang. In the two functions which include outbound HTTP logic, fuzzysearch and user-manager, OpenFaaS shows better performance, which was not the case during serial execution of the same functions. One possible reason for this is the keep-alive connection polling mechanism of the underlying HTTP library [64], which is possible when multiple outbound requests are made from the same long-running container in a short period of time, as is the case in this scenario.

The results from some functions such as n-body, prime-numbers, and whatlang are very similar between both WASM and OpenFaaS. To test whether a statistically significant difference exists, we repeated the nonparametric Mann-Whitney U test with an alpha value of 0.05 and the hypotheses:

- H0: the two populations are equal
- H1: the two populations are not equal

Insufficient evidence to reject the null hypothesis was present in two cases: n-body (p=0.23) and whatlang (p=0.25). In the remaining cases the obtained p-value was smaller than alpha, leading to the rejection of the null hypothesis.

Fig. 7 compares the performance consistency between WASM and OpenFaaS, visualizing all of the obtained response times.

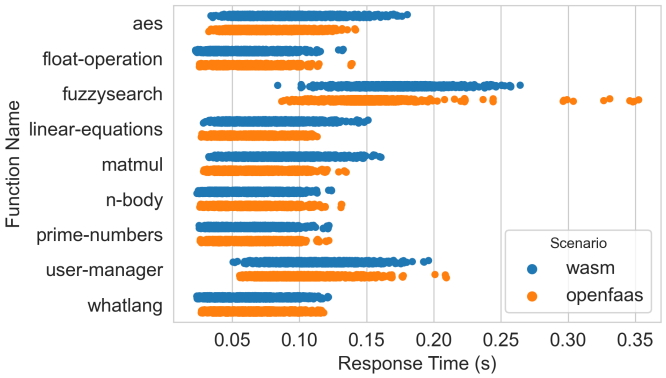

Fig. 7 Response Times in Seconds Across Different Concurrent Execution Runs

Similarly to the serial execution case, fuzzysearch exhibits a more inconsistent performance profile in the case of OpenFaaS than WASM. Focusing solely on the the standard deviation and coefficient of variation across all concurrently executed functions, we see results that are very similar between each other, with minor differences in terms of the computed values, as shown in Table 4.

In conclusion, OpenFaaS offers better concurrent execution performance compared to WASM when using an already warm container instance. However, WASM offers true scale to zero behavior without incurring an additional cold

Table 4 Comparison of Standard Deviation and Coefficient of Variation for Concurrent Executions

Function	SD[1] WASM	SD[1] OF[2]	CV[3] WASM	CV[3] OF[2]
aes	0.0253	0.0205	0.2828	0.2711
float-operation	0.0201	0.02	0.3481	0.3356
fuzzysearch	0.0231	0.0227	0.1336	0.1506
linear-equations	0.0221	0.0195	0.2923	0.3097
matmul	0.0235	0.0195	0.2777	0.3004
n-body	0.02	0.0199	0.3347	0.3277
prime-numbers	0.0193	0.0187	0.3279	0.3102
user-manager	0.0233	0.023	0.212	0.2334
whatlang	0.0203	0.0193	0.3353	0.3159

[1]Standard deviation

[2]OpenFaaS

[3]Coefficient of variation

start delay, easier scalability, and comparable concurrent performance when executing workloads which are not processor intensive.

4.4 Image Size Evaluation

The size of the software artifacts representing the functions to be executed is an important parameter when discussing serverless platforms and runtimes. Taking into account that serverless platforms, in production environments, can span across hundreds of nodes, and that functions need to be instantiated on multiple nodes either due to scaling or migration, large artifacts can have a significant impact on network traffic, storage requirements, and even processing power, in case they are transferred in an archive format (for example OCI images). Both the Spin shim and OpenFaaS work with the OCI image format, allowing for a comprehensive comparison between the image sizes for the functions. Whereas Spin only requires the WASM binary and a manifest file to be present in the image, the recommended OpenFaaS templates include additional supporting tools, and are based on slimmed down versions of OCI operating system images (alpine or debian-slim). Table 5 shows the size difference in kilobytes between the WASM and OpenFaaS OCI images.

A notable size increase is present in the case of Rust based functions since despite optimizations to the Dockerfile of the OpenFaaS template and introduction of two-staged builds, we still had to use "debian:bullseye-slim" as a base image. In the case of the Go functions, a more lightweight "alpine" base image is used. Nevertheless, as a result of using such base images, the OpenFaaS artifacts are at least an order of magnitude larger than their WASM counterparts. This can directly translate to higher costs for storage, increased utilization of network links, larger cold start times, and higher CPU utilization along with an increase in energy consumption during the unpacking process of the images.

Table 5 Comparison of OCI image sizes between WASM and OpenFaaS in kilobytes

Function	WASM Size (KB)	OF[1] Size (KB)	Increase[2]
aes	132	8780	x66.52
float-operation	122	8790	x72.05
fuzzysearch	712	52620	x73.9
linear-equations	161	8920	x55.4
matmul	134	8850	x66.04
n-body	562	50970	x90.69
prime-numbers	675	50950	x75.48
user-manager	733	52500	x71.62
whatlang	753	51040	x67.78

[1]OpenFaaS

[2]Size increase of the OpenFaaS OCI image compared to WASM

5 Conclusion

The lack of a comprehensive orchestration framework for WebAssembly modules is one of the last remaining hurdles that needs to be overcome before WebAssembly can be used as a serverless runtime technology for deployment of functions at scale, on par with other alternatives. To this effect, we extended the popular container orchestrator Kubernetes, allowing it to be used for managing WebAssembly modules, in addition to containers. Using a software shim which allows the containerd container runtime used by Kubernetes to interact with the Spin WebAssembly runtime, along with a new and dedicated Kubernetes operator for managing WASM workloads, we presented a solution that allows WASM serverless functions to be deployed using Kubernetes. The presented modifications do not require any changes to the underlying Kubernetes API, thus keeping compatibility with existing tools and application managers, allowing them to be reused in this new context.

The fact that at this point WebAssembly does not offer complete feature parity with more mature execution environments is expected to be resolved in the future, as more WebAssembly proposals are fully implemented and adopted by the WASM runtimes. The lack of certain specific features does not prevent the use of WASM for serverless functions, and the core functionality is already available. We feel that it is important to tackle the orchestration problem as early as possible, which might also lead to increased interest in this technology and thus grater adoption. This is also supported by the fact that the proposed solution for orchestrating WASM modules using Kubernetes is modular, allowing newer and more advanced WASM runtimes to be used once they become available, integrating them in the same way, through the use of software shims.

Results from the performance benchmarks conducted using a set of 9 benchmarking functions deployed both as WebAssembly modules using the Kubernetes WASM operator, as well as standard containers deployed using the OpenFaaS serverless framework confirm that WASM can indeed be seen

as a viable serverless runtime. Taking into account that each request is served by a dedicated instance of the WebAssembly module, the presented solution offers comparable execution performance to container based alternatives for most workloads, while offering per invocation isolation, and drastically smaller storage footprint. The software artifacts for WASM modules are at least an order of magnitude smaller than the traditional OCI images for containerized serverless functions, since they do not require any underlying layers, and can be constructed from a "scratch" one. This leads to drastically reduced storage requirements, as well as faster deployment times in case the artifact is not yet present on the given compute node. Experiments show that WASM modules have at least 2 times faster instantiation times compared to containers, even when the software artifacts are already preloaded on the nodes, and no external network connectivity is required.

In conclusion, WebAssembly has a great potential to be used as a runtime environment for multi-tenant serverless functions which are frequently invoked, require elasticity, and are not computationally intensive. The drastically reduced cold start times, coupled with smaller artifact sizes, native scale to zero capability, and per invocation isolation make WebAssembly the preferred choice for such scenarios. For serverless functions with processor intensive workloads, containers offer better performance, and for more complex workloads the time savings as a result of the faster instantiation time of WASM modules can be offset by the longer execution times. This supports the conclusion that at least for the time being, WebAssembly should be used in tandem with existing serverless runtime technologies, and not as a complete replacement for them. As a result of these findings, we plan to design a high-level, multi-tenant, serverless platform in the future, which could be directly accessed by multiple groups of users, allowing them to choose the most optimal runtime environment for their function during the function deployment process.

6 Threats to Validity

We strove to mitigate to the extend possible any potential threats to validity to the presented results during the design and execution phases of the research. To this effect, all tests were executed on completely uniform set of bare-metal nodes, with the exact hardware configuration. All of the OCI images required for instantiating the serverless functions were preloaded beforehand on the nodes, as to eliminate any outside influence on the function deployment times which would not be controllable. The functions which required outbound network communication to connect to BaaS services such as object storage and external databases reachable via a REST API, did so using local instances of those services, hosted in the same local area network. We recognize that the benchmarking functions might not be optimized to the full extend possible in terms of execution performance, but since the same version was executed across all nodes and both scenarios (both WASM and OpenFaaS) this makes

the obtained results relevant when compared with each other. Finally, the presented results are representative of the software component versions available during the time when the research was conducted. It is expected that additional performance improvements will be available in the future, in newly released versions of the software.

Declarations

6.1 Ethics approval and consent to participate

Not applicable.

6.2 Consent for publication

Not applicable.

6.3 Availability of data and material

The generated raw data, software, and outputs from the data analysis are publicly available under a permissive license on https://github.com/wasm-orchestration.

6.4 Competing interests

The authors have no relevant financial or non-financial interests to disclose.

6.5 Funding

This study was funded by the Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University, Skopje, North Macedonia under the "NSA" project.

6.6 Authors' contributions

Conceptualization: S.F. and V.K.; Investigation: V.K.; Methodology: S.F. and V.K.; Software: V.K.; Validation: S.F. and V.K.; Formal analysis: S.F. and V.K. Writing (original draft preparation): V.K.; Writing (review and editing): S.F and V.K. All authors have reviewed the manuscript.

6.7 Acknowledgements

Not applicable.

References

[1] Datadog: The State of Serverless. https://www.datadoghq.com/state-of-serverless/ Accessed 2023-02-26

[2] Kratzke, N.: A Brief History of Cloud Application Architectures. Applied Sciences **8**(8), 1368 (2018). https://doi.org/10.3390/app8081368

[3] Hassan, H.B., Barakat, S.A., Sarhan, Q.I.: Survey on serverless computing. Journal of Cloud Computing **10**(1), 39 (2021). https://doi.org/10.1186/s13677-021-00253-7

[4] Pfandzelter, T., Bermbach, D.: IoT Data Processing in the Fog: Functions, Streams, or Batch Processing? In: 2019 IEEE International Conference on Fog Computing (ICFC), pp. 201–206. IEEE, Prague, Czech Republic (2019). https://doi.org/10.1109/ICFC.2019.00033

[5] Varghese, B., Buyya, R.: Next generation cloud computing: New trends and research directions. Future Generation Computer Systems **79**, 849–861 (2018). https://doi.org/10.1016/j.future.2017.09.020

[6] Multi-Access Edge Computing (MEC); Framework and Reference Architecture. ETSI MEC ISG. https://www.etsi.org/deliver/etsi_gs/MEC/001_099/003/02.02.01_60/gs_MEC003v020201p.pdf Accessed 2023-02-27

[7] Yang, S., Xu, K., Cui, L., Ming, Z., Chen, Z., Ming, Z.: EBI-PAI: Towards An Efficient Edge-Based IoT Platform for Artificial Intelligence. IEEE Internet of Things Journal, 1–1 (2020). https://doi.org/10.1109/JIOT.2020.3019008

[8] Baresi, L., Filgueira Mendonça, D., Garriga, M.: Empowering Low-Latency Applications Through a Serverless Edge Computing Architecture. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) Service-Oriented and Cloud Computing. Lecture Notes in Computer Science, pp. 196–210. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-67262-5_15

[9] Gusev, M., Ristov, S., Amza, A., Hohenegger, A., Prodan, R., Mileski, D., Gushev, P., Temelkov, G.: CardioHPC: Serverless Approaches for Real-Time Heart Monitoring of Thousands of Patients. In: 2022 IEEE/ACM Workshop on Workflows in Support of Large-Scale Science (WORKS), pp. 76–83 (2022). https://doi.org/10.1109/WORKS56498.2022.00015

[10] Salehe, M., Hu, Z., Mortazavi, S.H., Mohomed, I., Capes, T.: VideoPipe: Building Video Stream Processing Pipelines at the Edge. In: Proceedings of the 20th International Middleware Conference Industrial Track, pp. 43–49. ACM, Davis CA USA (2019). https://doi.org/10.1145/3366626.3368131

[11] Kjorveziroski, V., Filiposka, S., Trajkovik, V.: IoT Serverless Computing at the Edge: A Systematic Mapping Review. Computers **10**(10), 130 (2021). https://doi.org/10.3390/computers10100130

[12] Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., Yasukata, K., Raiciu, C., Huici, F.: My VM is Lighter (and Safer) than your Container. In: Proceedings of the 26th Symposium on Operating Systems Principles. SOSP '17, pp. 218–233. Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3132747.3132763

[13] Randazzo, A., Tinnirello, I.: Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way. In: 2019 Sixth International Conference on Internet of Things: Systems, Management And Security (IOTSMS), pp. 209–214 (2019). https://doi.org/10.1109/IOTSMS48152.2019.8939164

[14] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., Popa, D.-M.: Firecracker: Lightweight Virtualization for Serverless Applications. In: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pp. 419–434 (2020)

[15] Hellerstein, J.M., Faleiro, J., Gonzalez, J.E., Schleier-Smith, J., Sreekanti, V., Tumanov, A., Wu, C.: Serverless Computing: One Step Forward, Two Steps Back. arXiv:1812.03651 [cs] (2018) arXiv:1812.03651 [cs]

[16] Kjorveziroski, V., Canto, C.B., Roig, P.J., Gilly, K., Mishev, A., Trajkovik, V., Filiposka, S.: IoT Serverless Computing at the Edge: Open Issues and Research Direction. Transactions on Networks and Communications **9**(4), 1–33 (2021). https://doi.org/10.14738/tnc.94.11231

[17] Agarwal, S., Rodriguez, M.A., Buyya, R.: A Reinforcement Learning Approach to Reduce Serverless Function Cold Start Frequency. In: 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 797–803 (2021). https://doi.org/10.1109/CCGrid51090.2021.00097

[18] Ling, W., Ma, L., Tian, C., Hu, Z.: Pigeon: A Dynamic and Efficient Serverless and FaaS Framework for Private Cloud. In: 2019 International Conference on Computational Science and Computational Intelligence (CSCI), pp. 1416–1421. IEEE, Las Vegas, NV, USA (2019). https://doi.org/10.1109/CSCI49370.2019.00265

[19] Wang, B., Ali-Eldin, A., Shenoy, P.: LaSS: Running Latency Sensitive Serverless Computations at the Edge. In: Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing, pp. 239–251. Association for Computing Machinery, New York, NY, USA (2020)

[20] Gorlatova, M., Inaltekin, H., Chiang, M.: Characterizing task completion latencies in multi-point multi-quality fog computing systems. Computer

Networks **181**, 107526 (2020). https://doi.org/10.1016/j.comnet.2020.107526

[21] Hall, A., Ramachandran, U.: An execution model for serverless functions at the edge. In: Proceedings of the International Conference on Internet of Things Design and Implementation. IoTDI '19, pp. 225–236. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3302505.3310084

[22] W3C WebAssembly Working Group. https://www.w3.org/wasm/. https://www.w3.org/wasm/ Accessed 2023-02-26

[23] Wang, Z., Wang, J., Wang, Z., Hu, Y.: Characterization and Implication of Edge WebAssembly Runtimes. In: 2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys), pp. 71–80 (2021). https://doi.org/10.1109/HPCC-DSS-SmartCity-DependSys53884.2021.00037

[24] WebAssembly System Interface – Proposals. WebAssembly. https://github.com/WebAssembly/WASI/blob/bac366c8aeb69cacfea6c4c04a503191bf1cede1/Proposals.md Accessed 2022-02-27

[25] WASI Filesystem. WebAssembly. https://github.com/WebAssembly/wasi-filesystem Accessed 2022-02-27

[26] WASI Sockets. WebAssembly. https://github.com/WebAssembly/wasi-sockets Accessed 2022-02-27

[27] The WebAssembly System Interface. https://wasi.dev/. https://wasi.dev/ Accessed 2022-02-27

[28] Wasi-Threads. WebAssembly (2022). https://github.com/WebAssembly/wasi-threads Accessed 2022-02-27

[29] Kjorveziroski, V., Filiposka, S., Mishev, A.: Evaluating WebAssembly for Orchestrated Deployment of Serverless Functions. In: 2022 30th Telecommunications Forum (TELFOR), pp. 1–4 (2022). https://doi.org/10.1109/TELFOR56187.2022.9983733

[30] Gadepalli, P.K., McBride, S., Peach, G., Cherkasova, L., Parmer, G.: Sledge: A Serverless-first, Light-weight Wasm Runtime for the Edge. In: Proceedings of the 21st International Middleware Conference. Middleware '20, pp. 265–279. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3423211.3425680

[31] Long, J., Tai, H.-Y., Hsieh, S.-T., Yuan, M.J.: A lightweight design for serverless Function-as-a-Service. IEEE Software **38**(1), 75–80 (2021) arXiv:2010.07115 [cs]. https://doi.org/10.1109/MS.2020.3028991

[32] Hockley, D., Williamson, C.: Benchmarking Runtime Scripting Performance in Wasmer. In: Companion of the 2022 ACM/SPEC International Conference on Performance Engineering. ICPE '22, pp. 97–104. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3491204.3527477

[33] Jangda, A., Powers, B., Berger, E., Guha, A.: Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code (2019). https://doi.org/10.5555/3358807.3358817

[34] Ménétrey, J., Pasin, M., Felber, P., 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, pp. 3–8 (2022). https://doi.org/10.1145/3526059.3533618

[35] Kjorveziroski, V., Filiposka, S.: Kubernetes distributions for the edge: Serverless performance evaluation. The Journal of Supercomputing **78**(11), 13728–13755 (2022). https://doi.org/10.1007/s11227-022-04430-6

[36] Goethals, T., De Turck, F., Volckaert, B.: Extending Kubernetes Clusters to Low-Resource Edge Devices Using Virtual Kubelets. IEEE Transactions on Cloud Computing **10**(4), 2623–2636 (2022). https://doi.org/10.1109/TCC.2020.3033807

[37] Sebrechts, M., Ramlot, T., Borny, S., Goethals, T., Volckaert, B., De Turck, F.: Adapting Kubernetes Controllers to the Edge: On-Demand Control Planes Using Wasm and WASI. arXiv (2022). https://doi.org/10.48550/arXiv.2209.01077

[38] Kubernetes Operator Pattern. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/ Accessed 2022-02-27

[39] Kjorveziroski, V., Filiposka, S.: WebAssembly as an Enabler for Next Generation Serverless Computing. Submitted

[40] Containerd – An Industry-Standard Container Runtime with an Emphasis on Simplicity, Robustness and Portability. https://containerd.io/. https://containerd.io/ Accessed 2022-02-27

[41] Definition of Shim - Gartner Information Technology Glossary. https://www.gartner.com/en/information-technology/glossary/shim Accessed 2022-02-27

[42] Containerd - WasmEdge Runtime. https://wasmedge.org/book/en/use_cases/kubernetes/cri/containerd.html Accessed 2022-02-27

[43] Containerd-Shim-Spin-v1 – Containerd Shims for Running WebAssembly Workloads in Kubernetes. https://github.com/deislabs/containerd-wasm-shims/tree/main/containerd-shim-spin-v1 Accessed 2022-02-27

[44] Introducing Spin. https://spin.fermyon.dev Accessed 2022-02-27

[45] Wasmtime. https://wasmtime.dev/ Accessed 2022-02-27

[46] Building Spin Components in Other Languages. https://developer.fermyon.com Accessed 2022-02-27

[47] Component Model Design and Specification. https://github.com/WebAssembly/component-model Accessed 2022-02-27

[48] Wasm, WASI, Wagi: What Are They? https://www.fermyon.com/blog/wasm-wasi-wagi Accessed 2022-02-27

[49] Kumar, R., Trivedi, M.C.: Networking Analysis and Performance Comparison of Kubernetes CNI Plugins. In: Bhatia, S.K., Tiwari, S., Ruidan, S., Trivedi, M.C., Mishra, K.K. (eds.) Advances in Computer, Communication and Computational Sciences. Advances in Intelligent Systems and Computing, pp. 99–109. Springer, Singapore (2021). https://doi.org/10.1007/978-981-15-4409-5_9

[50] Helm - The Package Manager for Kubernetes. https://helm.sh/ Accessed 2022-02-27

[51] Runtime Class. https://kubernetes.io/docs/concepts/containers/runtime-class/ Accessed 2022-02-27

[52] Kim, J., Lee, K.: FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 502–504 (2019). https://doi.org/10.1109/CLOUD.2019.00091

[53] WebAssembly Proposals. https://github.com/WebAssembly/proposals Accessed 2023-06-13

[54] OpenFaaS Golang HTTP Templates. https://github.com/openfaas/golang-http-template Accessed 2022-02-27

[55] OpenFaaS Rust HTTP Template. https://github.com/openfaas-incubator/rust-http-template Accessed 2022-02-27

[56] OpenFaaS - Serverless Functions Made Simple. https://www.openfaas.com/ Accessed 2022-02-27

[57] Apache OpenWhisk Is a Serverless, Open Source Cloud Platform. https://openwhisk.apache.org/ Accessed 2022-02-27

[58] Knative. https://knative.dev/ Accessed 2022-02-27

[59] Kubeless. https://kubeless.io/ Accessed 2022-02-27

[60] Dogan, J.: Rakyll/Hey. https://github.com/rakyll/hey Accessed 2022-02-27

[61] K3s: Lightweight Kubernetes. https://k3s.io/ Accessed 2022-02-27

[62] pREST —Instant RESTFul APIs"" for Your Data Join Data across Databases, REST Services to Build Powerful Modern Applications. https://prestd.com/ Accessed 2022-02-27

[63] Stack - OpenFaaS. https://docs.openfaas.com/architecture/stack/ Accessed 2022-03-08

[64] Client in Reqwest::Blocking - Rust. https://docs.rs/reqwest/latest/reqwest/blocking/struct.Client.html Accessed 2022-03-04