The Journal of Supercomputing (2022) 78:13728–13755
https://doi.org/10.1007/s11227-022-04430-6

Kubernetes distributions for the edge: serverless performance evaluation

Vojdan Kjorveziroski[1] · **Sonja Filiposka**[1]

Accepted: 3 March 2022 / Published online: 24 March 2022

Abstract

Serverless computing, especially when deployed at the edge of the network, is seen as an enabling technology for the future development of more complex Internet of Things systems. However, special care must be taken when deploying new edge infrastructures for serverless workloads in terms of resource usage and network connectivity. Inefficient utilization of the available computing resources might easily cancel out the benefits acquired by moving the equipment closer to the edge, namely the reduced communication latency. Containers, together with the Kubernetes container orchestrator, are used by many serverless platforms today. We evaluate the performance of three different Kubernetes distributions—full-fledged Kubernetes, K3s, and MicroK8s when deployed in a resource constrained environment at the edge. We use the OpenFaaS serverless platform and employ 14 different benchmarks divided into three separate categories to evaluate various aspects of the execution performance of the distributions. Four different test types are performed focusing on cold start latency, serial execution performance, parallel execution using a single replica, and parallel execution utilizing different autoscaling strategies. Our results show that the edge-oriented K3s and MicroK8s distributions offer better performance in the majority of the tests, while a full-fledged deployment exhibits noticeable advantages for sustained loads such as parallel function invocation using a single replica.

Keywords Serverless computing · Internet of Things · Function as a service · Kubernetes · Performance evaluation

Vojdan Kjorveziroski and Sonja Filiposka equally contributed to this work.

✉ Vojdan Kjorveziroski
vojdan.kjorveziroski@finki.ukim.mk

Sonja Filiposka
sonja.filiposka@finki.ukim.mk

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

1 Introduction

The emergence of the cloud computing paradigm has drastically altered the computational landscape and has had a significant influence on both existing and future application architectures [1]. Incentivized by pay-as-you go pricing, virtually unlimited capacity, and fast time to market, developers have even devised new programming trends and application designs, better leveraging the features of the cloud infrastructure [2]. These trends include the rising popularity of the microservice architecture, the use of containers as runtime environments where the applications are executed, as well as the use of various *as a service* managed products [3], further easing the development and deployment processes. Perhaps one of the most impactful of these new offerings is serverless computing [4], with its combination of function as a service and backend as a service [5], both made possible and inspired by the scalability of the cloud.

Serverless computing completely abstracts the underlying infrastructure, instead focusing solely on the logic that needs to be performed to solve a given task. Despite its name, servers are still utilized, but the responsibility for their setup, management, and maintenance has simply shifted further up the chain of developer, platform provider, infrastructure provider. Developers submit finished functions written in their programming language of choice, usually utilizing additional backend as a service offering by their platform provider for user management, application persistence, or engagement analysis [6].

Even though cloud computing has revolutionized application development and infrastructure management since its inception, a number of open issues still remain, especially in terms of communication delays between remote locations and the data centers where the infrastructure itself is hosted [7]. These issues become more prevalent with the continuing rise of Internet of Things (IoT) devices, which have become ubiquitous in everyday homes, while also being seen as enablers for new industry technologies [8]. To deal with the sheer data volume generated by these low-powered and resource constrained devices, researchers have recommended the adoption of edge computing. By setting up compute infrastructure closer to the data source capable of preprocessing the incoming data, numerous benefits can be achieved such as faster response times for latency sensitive applications, increased user privacy by sending only filtered data upstream to the cloud, and reduced network congestion as a result of the initial preprocessing done at the edge [9].

However, the question of how to deploy these edge infrastructures still remains today. Simply copying the existing cloud architectures' design has been shown to be inefficient for the resource constrained edge due to the potentially limited compute capacity [10]. Overcoming these problems, serverless computing has been identified as a suitable fit for edge infrastructures because of the benefits that it offers. Recently, many commercial serverless providers have adapted their portfolios to offer computing options at the edge which can either run on leased or privately owned infrastructure [11–13]. At the same time, open-source solutions have been published, attracting significant developer interest, and establishing

thriving communities [14]. Despite this, open issues still remain when it comes to adapting the serverless computing to the edge of the network [15], with perhaps the most pressing one being the selection of the appropriate runtime environment where the functions are executed [5, 16].

Considering that containers are the most used execution environment for the increasingly popular edge serverless platforms, a number of container orchestrators have been adapted to the resource constrained edge. Kubernetes, as the most widely used container orchestrator today [17], has been chosen as an underlying component for most of the open source serverless platforms currently available. This has attracted a noticeable interest from both academia and industry to better optimize the orchestrator and make it more suitable for deployment at the edge of the network.

The goal of this paper is to analyze three different Kubernetes distributions together with their deployment methods and evaluate the performance impact that they have on serverless function execution performance on constrained edge infrastructures. The main contributions of this work are given as follows:

- Description of a varied benchmarking strategy encompassing different modes of serverless function execution, including serial and parallel invocation with and without scaling mechanisms.
- Adaptation of the FunctionBench serverless benchmarking suite to the previously unsupported OpenFaaS platform and open sourcing the resulting implementation.
- Evaluation of the performance characteristics of three different Kubernetes distributions, along with an assessment of how they adapt to the edge of the network, elaborating their advantages in different scenarios.
- Analysis of the supported scaling strategies of the OpenFaaS serverless platform using different workload types.

The rest of this paper is organized as follows: in section two, we provide additional background information on lightweight Kubernetes distributions and discuss related work in this field. We then proceed with section three where we explain the testing methodology that we have devised, the infrastructure where the tests have been performed, as well as the motivation behind the different benchmarking scenarios. In section four, we present the obtained results before moving to the fifth and final section with which we conclude the paper, summarizing the findings and discussing how the various optimizations of the tested Kubernetes distributions have impacted their performance.

2 Background and related work

Serverless edge computing has become a popular topic in recent years [18], attracting a significant research interest from various research groups, some of which have also focused on the performance aspects and how they can be measured in a reproducible manner using cross-platform benchmarking suites. In this section, we first present the motivation for the existence of the various Kubernetes distributions

today, before continuing with an overview of the latest literature concerning serverless platform benchmarks and the underlying infrastructures on top of which they are deployed.

2.1 The relationship between Kubernetes and serverless

Since its initial release in 2014, Kubernetes has seen an enormous rise in its popularity. On one hand, many organizations have chosen to reorganize their computing infrastructure to incorporate this container orchestrator, hoping to reap the benefits from easier management of applications at scale. On the other hand, academia has identified it as an interesting research topic, proposing yet more areas and usage scenarios where Kubernetes can be applied [19].

As a result of its popularity and the diverse use-cases in which it can be used, alternative Kubernetes distributions have been developed which are specifically optimized for a given scenario [20]. The need for such custom distributions has arisen because of the increasingly large number of components which have been incorporated in Kubernetes itself, allowing it to be deployed on diverse infrastructures, ranging from multiple distributed cloud providers with thousands of nodes, to development machines for testing purposes. The inclusion of these addons increases the size, resource requirements and the deployment complexity, even though many of the addons are not utilized in smaller or resource constrained clusters, such as those deployed at the edge of the network. To better describe the nature of these additional components, they include features such as the option of seamlessly integrating with the existing service offerings of the cloud provider where the cluster is deployed on persistent datastores that are optimized for supporting large number of nodes, instead of minimizing the resource footprint. It is evident that in edge-based scenarios, such cloud focused components are not only unnecessary, but their omission could potentially lead to increased runtime performance as well. In the wider context of serverless computing, where one of the primary focuses is function instantiation speed as a prerequisite for efficient scale-to-zero behavior, this added complexity could also potentially affect the initial start up times of containers, impacting end-user experience.

To alleviate these shortcomings, a number of open-source initiatives such as K3s [21] and MicroK8s [22] have emerged, shedding unnecessary Kubernetes components or merging existing ones, with the aim of reducing their footprint. Focused on enabling easy deployments on bare-metal infrastructure, they are appropriate options for edge computing and can be utilized for hosting serverless edge platforms or enabling multi access edge computing (MEC) scenarios. These improvements have allowed the resource requirements to be reduced, the deployment process to be greatly sped up, and at the same time increasing the elasticity, using simpler scale up and scale down operations. Nonetheless, all these Kubernetes distributions are still compatible with the original software and can run and utilize the same configuration and applications [23].

The differences between the various Kubernetes distributions in terms of the included components raise the question whether they can indeed offer better

performance compared to a full-fledged Kubernetes cluster, and what are the compromises which allow them to do so.

2.2 Related work

The main issue when it comes to any performance evaluation is reproducibility and the introduction of relevant benchmarks which can accurately depict the performance specifics of the various platforms undergoing the tests. To solve these problems, researchers have developed comprehensive test suites for serverless functions, including both microbenchmarks and real-world workloads [24, 25]. While microbenchmarks focus exclusively on a specific component such as processor speed, network performance, or the number of input/output (I/O) operations in a given time frame, real-world workload benchmarks aim to evaluate the performance characteristics of commonly executed tasks. In most cases, the performance of these real-world tasks is not determined solely by a single component, thus allowing these tests to determine the overall quality of the infrastructure. Examples of such workloads that are representative of real use cases include the training of machine learning models, image processing, big-data analysis, video encoding and decoding [26, 27].

Unfortunately, the lack of cross-compatibility between the serverless solutions [28] forces researchers to only target a specific set of platforms, relying on the open-source community or other interested researchers to adapt them to other, initially unsupported environments. Maissen et al. have developed [29] and open sourced FaaSdom [30], a set of serverless functions whose aim is to characterize the performance of popular cloud computing serverless platforms such as AWS Lambda, Azure Functions, Google Cloud Functions, and IBM Cloud Functions. In a similar manner, the authors of [26] provide an overview of popular commercial serverless platforms, providing a taxonomy covering 20 characteristics and devising benchmarks aimed at describing their performance. Eismann et al. [31] approach the issue of benchmarking public serverless platforms in a different manner. Instead of evaluating their performance simply at a given point in time, they test the performance variability across time and how the presumed overall load of the public cloud infrastructure might impact execution of individual functions.

This research interest is not limited only to commercial serverless platforms but also extends to open source and self-hosted ones. Li et al. [14] analyze four different solutions running on top of the Kubernetes container orchestrator, evaluating their performance while providing architecture details and how they might have impacted the acquired results. The authors of [32] compare the performance of different platforms, including OpenWhisk and Lean-OpenWhisk, a more lightweight alternative, suitable for deployment on edge devices. Evaluation of serverless edge platforms has also been performed in [33], where two commercial and one open-source serverless platform have been included with the aim of testing single node serverless edge infrastructures.

The optimization of existing open-source serverless platforms to better run at the edge of the network where computing capacity is limited has not been restricted only to OpenWhisk and its Lean-OpenWhisk counterpart. Authors of [34] improve

the scheduling of Knative, another popular serverless framework built on top of Kubernetes by implementing linear regression models, optimizing the incurred cold start latency of new function invocations. Similar work has also been done for Kubeless [35] and OpenWhisk [36].

While it is important to optimize existing serverless solutions for the network edge, the underlying computing infrastructure on top of which they are running must be taken into account as well. Kayal et al. [20] evaluate the overall feasibility of running Kubernetes on edge infrastructures and discuss a number of lightweight Kubernetes distributions which make it possible to do so. Eiermann et al. [37] further confirm the need for lighter Kubernetes solutions, showing that the idle resource usage of Kubernetes is greater than other similar container orchestrators albeit with a more limited feature set.

It is clear that a number of studies have been conducted to evaluate the performance characteristics of various serverless platforms on one hand, and container orchestrators, on another. However, to the best of our knowledge, there is no comprehensive analysis of how the different distributions of the most popular container orchestrator today, Kubernetes, impact the performance of serverless functions. It must be noted that some of these new Kubernetes distributions have been explicitly tailored for the resource constrained edge, so a comprehensive evaluation is required. In our opinion, the performance of serverless platforms running on top of the Kubernetes orchestrator greatly depends on the chosen Kubernetes distribution, and these two factors should be analyzed together.

3 Methodology

In order to evaluate whether a given Kubernetes distribution plays a significant role in the exhibited performance of a serverless platform when deployed on edge infrastructure with modest performance, the following aspects should be taken into account:

- Appropriate benchmarks capable of capturing different performance aspects;
- A serverless platform, preferably one which is widely used, serving as the benchmarking functions' execution environment;
- A criterion for choosing which Kubernetes distributions to include;
- A benchmark execution strategy, with the aim of acquiring as relevant data as possible, reminiscent of real-world execution scenarios.

In the following subsections, we elaborate the choices that we have made for each of these points.

3.1 Benchmarks selection

Recently, a number of serverless function benchmarking suites have been published [13, 25, 38], aimed at different platforms and execution scenarios. It is common for

these benchmarks to focus exclusively on either commercial or open source serverless platforms due to the lack of a unified application programming interface (API) and no common set of supported features between them.

For our tests, we have chosen to use a mix of microbenchmarks and real-world workloads applicable to edge computing scenarios, with the end goal of independently testing different performance aspects of the underlying Kubernetes platforms. As to not fragment this field even further by devising a custom set of tests, and at the same time to ensure higher reproducibility, we have adapted the FunctionBench serverless benchmarking suite presented by Kim et al. [38] which has been fully open sourced [39]. We have utilized 14 different tests in total, divided into three distinct categories:

- CPU & Memory—The central processing unit (CPU) performance is usually the most important metric to be evaluated, due to its relevancy for different usage scenarios. We have decided to include in this category memory tests as well, since many serverless platforms offer only a limited number of execution environment flavors, tying the allocation of CPU shares in terms of the configured memory or vice versa [27]. 8 of the 14 conducted tests can be associated with this category:
  - Float operation—performance of common mathematical operations;
  - Image processing—time taken to apply different filters to a given image;
  - Linear equation performance—solving linear equations of an arbitrary size;
  - Matrix multiplication—multiplication of square matrices;
  - Model training—training a machine learning (ML) model on a given dataset;
  - Advanced Encryption Standard (AES) encryption and decryption—decryption and encryption of arbitrary messages using the AES block-cipher algorithm;
  - XML manipulation—rendering performance of a large XML document;
  - Video processing—video manipulations performed on sample video files.
- Disk Input/Output (I/O) operations—the speed of the storage devices has a significant role in the overall perceived performance of the serverless functions and directly impacts all function types, not only those with heavy I/O workloads. Non-optimal I/O performance leads to longer start up times and reduced performance when loading large programming libraries for the first time into memory. 4 of the 14 conducted tests are related to measuring disk performance:
  - Disk throughput using the dd command—sequential disk throughput using the UNIX dd tool;
  - Gzip compression—Gzip compression of arbitrary files;
  - Random disk I/O—testing random disk I/O performance using Python libraries;
  - Sequential disk I/O—testing sequential disk I/O performance using Python libraries.
- Network performance—While network performance is important for any modern system that uses network connectivity for information exchange, it is also of

paramount importance for edge based serverless functions, playing a major role not only in the total throughput, but also in the incurred network latency. 2 of the 14 tests focus on evaluating network performance:

- Large file download—download of an arbitrary large file from an object storage service;
- Small dataset download—download of a modestly sized JSON dataset which is then deserialized and serialized again.

The described benchmarks are a mix of pure microbenchmarks testing a single hardware aspect on one hand, and real-world scenarios relevant for the cases of edge computing, on the other. With the inclusion of the model training, video processing, and image processing tests, we strove to show the performance characteristics of the tested distributions in terms of these increasingly popular and realistic edge and serverless workloads [19]. Full-fledged application scenarios were omitted on purpose, since the results obtained by testing them would not be as relevant outside of their specific use-case, would be heavily dependant on their implementation, and thus would not be representative of the distributions' end-to-end performance differences.

3.2 Serverless platform selection

The majority of serverless platforms today are using containerization as the chosen runtime technology. Even though it offers reduced performance in comparison to more efficient runtime environments [40], this choice increases adoption levels among diverse groups of administrators and developers since it relies on well-established and well-tested technology, with which many are already familiar. Using containers as the runtime environment also does not impose any restrictions in terms of which functions can be deployed, allowing virtually any programming language to be used, and at the same time easing the migration process to serverless for existing code bases.

The serverless platforms that are based on containerization require a container orchestrator capable of distributing the instantiated functions across different computing nodes, each of them joined to the same computing cluster.

We have opted to use OpenFaaS [41] as the serverless framework with which to benchmark the performance of the different Kubernetes distributions. It is the most popular serverless framework among those that natively support Kubernetes [42–44], based on the number of GitHub stars that the project has received [45]. Furthermore, OpenFaaS supports two different function scaling modes [46]: native scaling based on internal metrics which can be customized by the administrator (e.g., requests per second, response codes, or resource usage); scaling using the default Kubernetes Horizontal Pod Autoscaler (HPA) [47] mechanism, which is based on current CPU and memory usage. This allows us to determine the behavior of these two approaches across different underlying Kubernetes distributions.

OpenFaaS has many built-in function templates for popular programming languages which have been optimized for performance. However, as with other similar

frameworks, it also allows developers to specify their own container template to be used. These approaches, coupled with dedicated client-side command line tooling, allow easy adaptation of existing serverless functions, including the FunctionBench benchmarks, to the OpenFaaS serverless platform.

3.3 Kubernetes distribution selection

Even though Kubernetes is a rather complex system involving multiple different components, throughout the years, the codebase has matured, and advancements have been made which allow simple and fast deployment of new production and testing clusters. These advancements have been focused mainly on modularizing the architecture and publishing well-defined APIs for interaction with external systems, such as storage [48] or networking [49]. The increased modularity on one hand and the ever-growing number of components on the other have incentivized the development of lightweight Kubernetes distributions that ship only with the necessary components for a given use-case. Nevertheless, these are fully conformant Kubernetes platforms, capable of running the same software and undergoing the same configuration.

For our tests, we have selected three Kubernetes deployment methods: Kubespray [50], K3s [21], and MicroK8s [22]. Kubespray is an open-source project which aims to simplify the deployment of full-fledged Kubernetes clusters on different infrastructures, ranging from on-premise bare-metal servers to multi-cloud setups. Contrary to this, both K3s and MicroK8s are lightweight Kubernetes distributions specifically aimed at the network edge. Both are characterized with fast and simple setup procedures and support multi-node architectures as well.

To aid the deployment process, all three selected Kubernetes distributions come with extensive documentation. Kubespray is comprised of a set of Ansible playbooks (specialized automation scripts), capable of deploying the Kubernetes cluster once all parameters have been specified. K3s is packaged as a single binary and the whole deployment process can be completed by simply downloading and executing the official shell script hosted on the project's website. Finally, MicroK8s is packaged as a snap application, capable of being deployed on any GNU/Linux distribution where the snapd daemon is running.

Apart from the packaging and distribution choices, other aspects can play a significant role in potential performance differences between the distributions. K3s has opted to bundle all Kubernetes components into a single binary which runs as a single process, thus co-locating both the K3s server and the agent [51]. To reduce load on the underlying storage devices and offer better I/O speed, the etcd key–value store traditionally used for persisting the Kubernetes cluster state has been replaced by SQLite, using an adapter mechanism. The development of such an adapter also facilitates the use of different database backends. Due to these changes and the removal of the cloud specific integrations, K3s has lower minimum hardware requirements compared to a traditional Kubernetes deployment. It requires 512MB of memory for master nodes and 256MB for worker nodes [52], compared to more than 2GB per node for a traditional deployment with Kubeadm [53], also used by

Table 1 Execution environment specification

Parameter	Value
Operating system	Ubuntu 20.04
Number of nodes	6 (1 master, 5 workers)
CPU	Intel Xeon X5647
Memory	8 GB
Disk space	320 GB
LAN connection	1 Gbps
Kubernetes version	1.20.7[a], 1.23.0[b]
CNI plugin	Calico 3.21.2
OpenFaaS version	0.21.1

[a] Kubespray and K3s

[b] MicroK8s

Kubespray. Differences also occur in terms of processor cores, with K3s requiring at least 1, while Kubernetes 2. In terms of MicroK8s, apart from using the snap packaging format, it has also replaced the etcd database with a new implementation called Dqlite [54], which works similarly to SQLite, albeit with the possibility of distributed operation. The minimum hardware requirements for MicroK8s are higher than those for K3s, requiring at least 540MB of memory [55], with no official recommendation for the number of processor cores.

More information regarding the different component versions, setup procedure and execution strategy is available in the next subsection, Execution Strategy.

3.4 Execution strategy

Table 1 provides an overview of the execution environment used for the benchmarks, listing the versions of the various software components, as well as the hardware specification of the bare-metal machines. In summary, a set of 6 physical machines was used for the deployment of the three selected Kubernetes distributions. Each cluster was comprised of 1 master and 5 worker nodes. The master node did not host any function instances, thus reserving its resources for the Kubernetes control plane. Once all of the tests for the given distribution were finished, the machines were reinstalled from scratch and the next Kubernetes distribution was deployed. Kubernetes version 1.20.7 was used for both the Kubespray and K3s deployments and Kubernetes 1.23.0 for MicroK8s. The difference in Kubernetes versions is due to the fact that MicroK8s supports additional worker nodes without enabling high-availability by default only since version 1.23.0 [56]. Even though node addition is supported in previous MicroK8s versions, the first two additional nodes (up to a total of three) are automatically made part of the control plane which in our case would have negatively impacted the acquired results. Control plane nodes take an active role in maintaining the cluster state and require the database to be replicated between them, resulting in increased

Table 2 Execution parameters for each test

Test category	Test name	Parameters	Purpose	Values
CPU & Memory	Float-operation	*n*	Number	*n*: 10,000,000
	Image-processing	URL	File to download	–
	Linpack	*n*	Matrix size	*n*: 5000
	Matmul	*n*	Matrix size	*n*: 5000
	Model-training	URL	Source data	–
	Pyaes	*n*, *m*	Length, iterations	*n*: 1000 *m*: 100
	Chameleon	*n*, *m*	Rows x cols	*n*, *m*: 2000
	Video-processing	URL	File to download	–
Disk	Gzip-compression	File_size	Size (MB)	File_size: 50
	dd	bs, count	Block size, num. files	bs: 100M count: 1
	Random-disk-io	File_size, byte_size	File and block size	File_size: 100 byte_size: 1024
	Sequent0ial-disk-io	File_size, byte_size	File and block size	File_size: 100 byte_size:1024
Network	s3-object-storage	Input_bucket, output_bucket, object_key	Connection parameters	100 MB binary file download
	Json-dumps-loads	URL	File to download	–

load and disk activity. As a result of this, we have chosen to dedicate only a single master node to be responsible for the control plane, in all three deployments.

All 6 machines have the exact same hardware specification providing a uniform execution environment and avoiding any performance differences between the nodes due to mismatched hardware. Each bare-metal host is equipped with an Intel Xeon X5647 octa-core CPU, 8GB of random-access memory (RAM), and a 320GB hard drive.

The same Kubernetes container networking interface (CNI) plugin was used across all distributions—Calico. Even though some of the distributions install Calico automatically during the deployment process, we opted for a manual installation, in order to ensure that the exact configuration was used in all cases. OpenFaaS was deployed using the official Helm chart [57], and the Longhorn storage plugin was used for providing persistent volumes to the containers which requested them.

The FunctionBench serverless functions used for benchmarking require additional parameters to be passed upon each invocation of a particular test function instance. A short description of the parameters required by each test is given below in Table 2.

A total of 5 test runs were executed for each Kubernetes distribution, testing different performance aspects:

- Cold start performance—each function is executed 100 times to test the cold start delay. After every execution, the number of instances for the function is

scaled down to 0. Upon the next invocation, a new container instance needs to be created before a response is returned.
- Serial execution performance—each function is continuously invoked for a period of 5 min using a single thread. Once a response is received, a new request is immediately sent. Auto-scaling is manually disabled as to not skew the results using multiple replicas.
- Parallel execution performance using a single replica—each function is invoked exactly 20 times using 20 parallel threads. This simulates a bursty workload where the number of requests per second increases dramatically in a short amount of time. Auto-scaling is manually disabled as to not skew the results using multiple replicas.
- Parallel execution using native OpenFaaS auto scaling—each function is invoked for a fixed amount of time using varying concurrency to determine the performance of the auto-scaling behavior.
- Parallel execution using Kubernetes Horizontal Pod Autoscaler—each function is invoked for a fixed amount of time using varying concurrency to determine the performance of the auto-scaling behavior.

Additional details and a discussion about the obtained results are available in Sect. 4, Results.

To execute the requests and to obtain accurate measurements in terms of response-time and number of executions in the given time frame, we have utilized the hey benchmarking tool [58]. All tests were invoked from a standalone machine present in the same local network as the Kubernetes cluster, but without having a role in the cluster. The network segment was dedicated to the testing infrastructure, avoiding any impact on latency or throughput due to external factors. The nodes which hosted the Kubernetes cluster itself were also dedicated to this task and no additional workload was run on them in order to avoid skewing of the results [31]. External resources required for some of the tests, such as datasets for model training, video files, or large binary files have been served from a dedicated local object storage server present in the same network, but on a different host, not part of the Kubernetes cluster, eliminating any unforeseen performance degradation due to network congestion.

3.5 Reproducibility and extensibility

Reproducibility aspects are very important for all experimental works which try to objectively measure a given metric, such as performance. However, in the case of modern technologies which undergo significant advancements regularly, and where the rate of feature change is great, the question of long-term relevancy and future extensibility needs to be addressed as well.

Recognizing these challenges, we have decided to employ a set of existing serverless benchmarks for all of the performed tests. The selected FunctionBench benchmarking suite has already been reused in various other works, targeting both commercial and open source platforms, with authors adding support for new computing

infrastructures previously unsupported by the original implementation [59–63]. Its permissive Apache 2.0 open source license also makes it possible to integrate completely new testing scenarios, either as additional microbenchmarks or real-world workloads.

Due to the lack of cross-platform compatibility between serverless solutions, the FunctionBench benchmarks which we decided to use had to be adapted for execution on the OpenFaaS platform. This required refactoring of the original code so that it could take full advantage of OpenFaaS, as well as rebuilding of the associated container images. All functions used the recommended OpenFaaS of-watchdog template [64] for increased performance, while keeping the same programming language—Python 3.7. We have published all of our modifications to the original serverless functions, along with the source code for executing the benchmark runs on GitHub [65], thus further contributing to the extensibility of the original implementation. Open sourcing this work allows the wider research community to:

- Eextend the existing benchmarking suite with new functions either in the form of microbenchmarks or real-world workloads;
- Reuse the benchmarking suite as is, leveraging the prepared Docker containers for execution of the same scenario on new Kubernetes distributions in the future, comparing their performance to those already tested;
- Reuse the benchmarking suite as is, testing different commercial and private infrastructures without focusing exclusively on Kubernetes as the underlying orchestration platform. Even though OpenFaaS can already be deployed both on Kubernetes clusters hosted on either private or public clouds, faasd [66] is a slimmed down version of the same serverless platform, which does not require a container orchestrator to function and has full compatibility with the original implementation.

In terms of the reproducibility of the acquired results, we have also published the full raw metrics acquired during the different testing runs in the same GitHub repository. These results can either be reused as reference points in future works or explored interactively using the accompanying Jupyter notebook, which serves a dual purpose. Apart from being used for data exploration, it also allows for the existing raw data to be replaced with testing results from different platforms, allowing for a head-to-head comparison, while keeping the same analysis methodology and visualizations.

4 Results

All three selected distributions (Kubespray, K3s and MicroK8s) support the complete lifecycle management of the deployed clusters, including node addition and node removal, but the complexity of executing these operations varies. Kubespray takes the longest to both provision a new cluster and upgrade an existing one, owing to the fact that the Kubernetes cluster is created in a conventional manner, with all components included. On the other hand, both K3s and MicroK8s take significantly

less time to execute the same operations. Another differentiating factor between these deployment methods is the number of supported plugins which can be automatically installed by the distribution. To ease the administrative burden, all three support installation and configuration of additional plugins for networking, storage, or monitoring. Kubespray does this by using the official Kubernetes manifests for each plugin, introducing templating variables where needed. Once the templates have been rendered, the generated manifests can be used on any Kubernetes cluster, not limited to those deployed using Kubespray. Contrary to this, K3s supports enabling of additional plugins by passing parameters to its installation script, abstracting away the rest of the process, while MicroK8s supports enabling and disabling of plugins at any time using its command line interface. This is made possible by custom shell scripts for each plugin that needs to be deployed. These shell scripts are applicable only to MicroK8s environments.

In the following subsections, we focus on these results obtained from each of the five test runs executed on the different Kubernetes distributions, as explained in 3.4 Execution Strategy.

4.1 The problem with the cold start delay

One of the main advantages of serverless computing compared to other paradigms is the option of scaling down to zero replicas functions which are not currently executed. This allows resources to be released when functions are idle, leading to more efficient resource usage and reducing billing costs. However, a drawback of this approach is the increased latency that is incurred upon the next invocation of a function that has been scaled down, since the runtime environment will need to be set up from scratch, loading both the code and all associated libraries from disk in the process. This cold start delay can be significantly larger than consecutive executions of an already active function.

The platform that we are using for running the tests, OpenFaaS, uses containers as the only supported runtime environment for executing the deployed serverless functions. Taking into account the fact that all of the tests are ran on multiple nodes with the same exact hardware, any cold start delay difference between the three Kubernetes distributions will be as a result of their underlying architecture and complexity.

Figure 1 shows the average container cold start delay across all different functions for each platform. Every single function has been executed 100 times, leading to 1400 measurements for all 14 functions on each Kubernetes distribution, or 4200 measurements in total, across all three distributions. It can be seen that Kubespray, which deploys a full-fledged Kubernetes cluster, exhibits a 15% increase in the cold start delay compared to both K3s and MicroK8s. The two lightweight edge distributions show very similar results when it comes to bringing up new container instances.

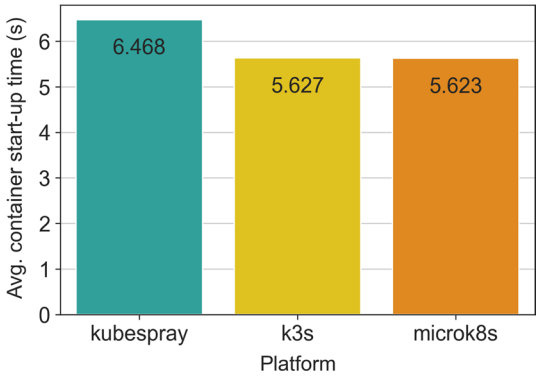

Fig. 1 Average container cold start delay for each of the three platforms

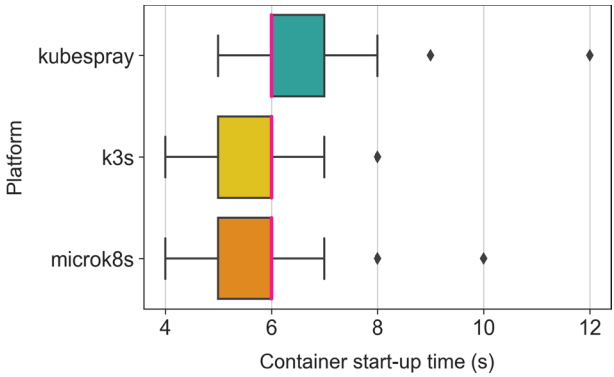

Fig. 2 Box plot of the cold start delays for each of the three platforms

To better visualize the difference in cold start performance, Fig. 2 shows a box plot outlining the median, minimum, maximum and the outliers[1] for the execution times incurred by the different Kubernetes distributions across all executions. Even though the median values are consistent, the lower performance of Kubespray is visible in this case as well.

Continuing the cold start delay analysis, Fig. 3 shows a more granular representation of the average delay incurred when creating a new container instance for each of the 14 functions that were used during the benchmarks. As was the case with the previously discussed results, a noticeable difference is present between the results of Kubespray on one hand and the more lightweight Kubernetes distributions on the other.

The small difference in performance between K3s and MicroK8s raises the question whether it is statistically significant or not. To test this, we performed the

[1] The extreme outliers for the K3s platforms (larger than 17s) have been cut off to aid the visibility of the figure. However, they have been taken into account in all other analyses and calculations.

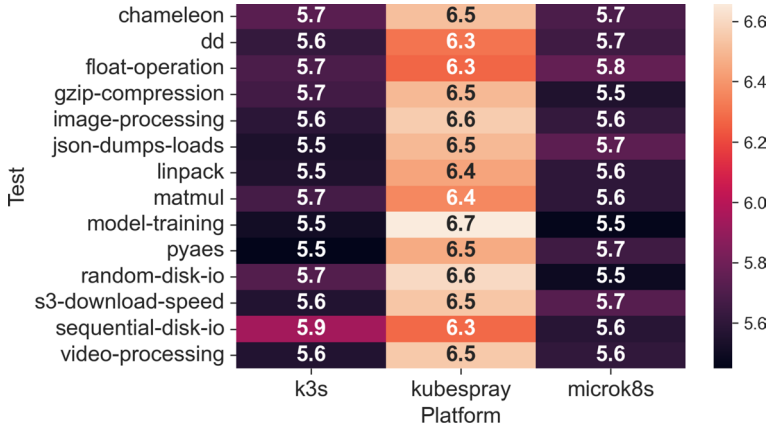

Fig. 3 Average cold start delay (in seconds) per function

nonparametric Mann–Whitney U test, with the following two hypotheses and an alpha value of 0.05:

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

After conducting the test, the p value is 0.202, leading to a failure to reject the null hypothesis. The acquired results do not show that there is a statistically significant difference between the cold start delays incurred by K3s and MicroK8s.

4.2 Serial execution performance

Most serverless platforms, especially those relying on containers as their runtimes, utilize methods to reduce the cold start delay. These methods can be implemented in a variety of ways, ranging from simple to more complex. A number of platforms simply leave a function running for a specific period of time after it has been executed, expecting further invocations as a result of temporal centrality. However, there are also examples of advanced monitoring infrastructure being utilized for performance data gathering which is then used for training machine learning models [67, 68].

In the case of OpenFaaS, while it does support automatic scale down to zero, this feature requires a paid license. However, manual scale down to zero via a simple API call to the OpenFaaS gateway is possible, allowing developers to design their own scale-up or scale-down implementations.

To test the raw performance of each platform, without taking into account the cold start delay when a new container instance is created, we have executed each function for 5 min, continuously sending requests after each received response from a single worker. All scaling features have also been disabled, to prevent an undesired

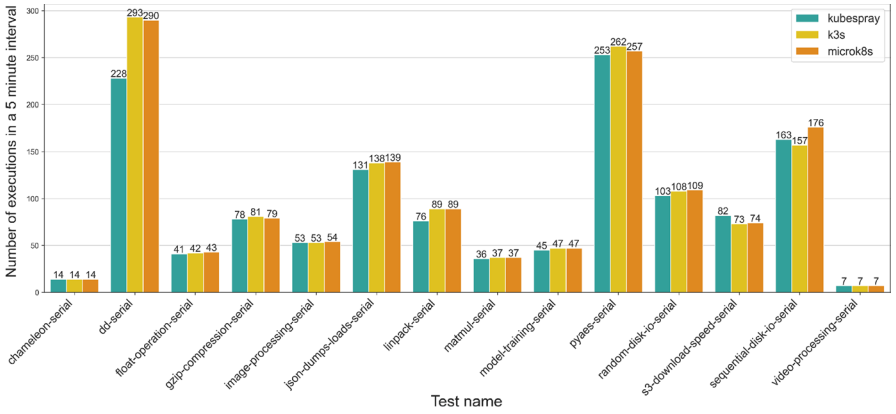

Fig. 4 Total number of serial executions for each function in a 5-min period

increase in the number of replicas for a given function, thus skewing the results. All the tests were performed one-by-one instead of at the same time for all functions.

Figure 4 shows the total number of executions for each function instance across all tested Kubernetes distributions. As was the case with the cold start benchmarks, K3s and MicroK8s exhibit very similar performance, being tied or differing by a single execution in 10 of the 14 tests. Kubespray has the most executions in the given time window in only a single test, the large file download from an object storage (*s3-download-speed*), while lagging in some of the more CPU intensive tests such as AES encryption/decryption and linear equation solving.

To offer another perspective on the obtained results, Fig. 5 shows the average response times for each function instance during the 5-min serial execution. As expected, the results between the lightweight K3s and MicroK8s distributions are comparable.

Several functions exhibited very similar performance not only between K3s and MicroK8s, but across all three different deployment methods, Kubespray included. To test the statistical significance of the results, we conducted the nonparametric Kruskal-Wallis test, testing the three groups. The tested hypothesis, using an alpha value of 0.05, in this case were given as follows:

- H0: the population medians are equal
- H1: the population medians are not equal

Statistically significant results were obtained for 13 of the 14 functions. The null hypothesis failed to be rejected for the video-processing test.

Unfortunately, the previous test is not able to specify where the difference occurs, among which groups. From the figures above, it is evident that the results between K3s and MicroK8s are much closer together compared to those of Kubespray in most cases. To formally test this, we performed the Mann-Whitney U test between the populations for each function for these two distributions, keeping the same hypothesis and alpha value of 0.05, as previously in Sect. 4.1. This has led to

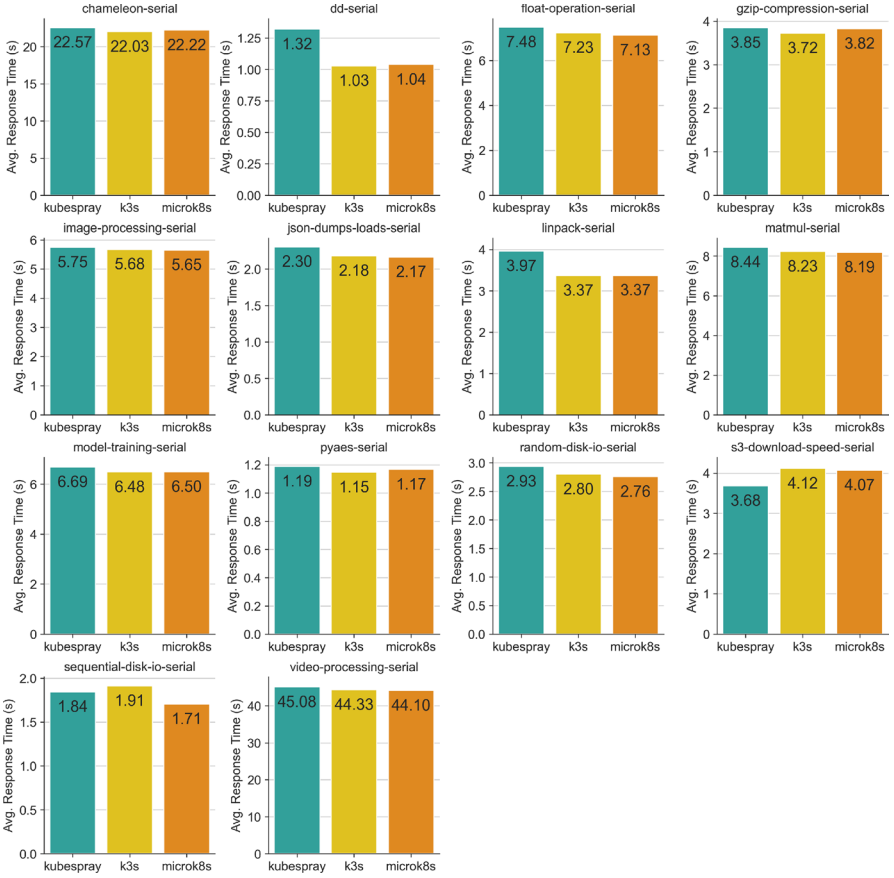

Fig. 5 Average response time for each function during a serial execution

statistically significant results and thus the rejection of the null hypothesis in 10 of the 14 tests: *chameleon*, *dd*, *float-operation*, *gzip-compression*, *json-dumps-loads*, *matmul*, *model-training*, *pyaes*, *random-disk-io*, *sequential-disk-io*. The null hypothesis failed to be rejected for the remaining 4: *image-processing*, *linpack*, *s3-download-speed*, *video-processing*. Three of these tests for which the null hypothesis was not rejected are CPU bound benchmarks, while *s3-download-speed* is a network intensive one.

4.3 Parallel execution without automatic replica scaling

A common practice whose aim is to offer better function execution performance is to reuse the same runtime environment for consecutive executions, instead of tearing it down and recreating it. While this does conserve computational resources, it opens new possibilities for attacks and data leakage, as a result of the reduced isolation.

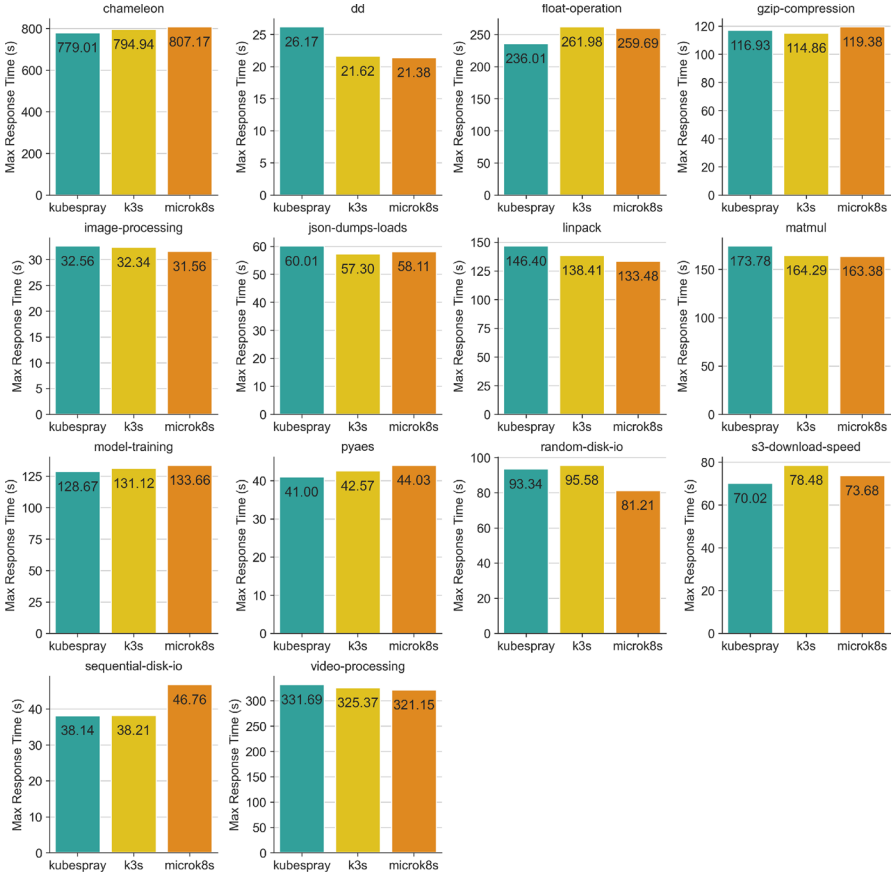

Fig. 6 Total time required for completing 20 requests using a single function replica

In the case of OpenFaaS, it is evident that a given container can be reused for multiple consecutive executions, since the containers are not immediately terminated once a response has been sent. To test how many requests can each function instance serve, we have executed 20 parallel requests at the same time, measuring the time taken to complete them. Apart from allowing us to analyze the concurrency rating per container instance, this test also shows how well the different Kubernetes distributions cope with bursty workloads. As previously, all scaling mechanisms were disabled during this test, forcing each function to run only with a single instance.

Figure 6 shows the total time required for each distribution to complete the 20 requests. Contrary to previous results, Kubespray exhibits better performance than both K3s and MicroK8s in 6 of the 14 tests, namely *chameleon*, *float-operation*, *model-training*, *pyaes*, *s3-download-speed*, and *sequential-disk-io*. K3s takes the lead in 2 tests: *gzip-compression* and *json-dumps-loads*, while MicroK8s in 6 (same as Kubespray): *dd*, *image-processing*, *linpack*, *matmul*, *random-disk-io*, *video-processing*.

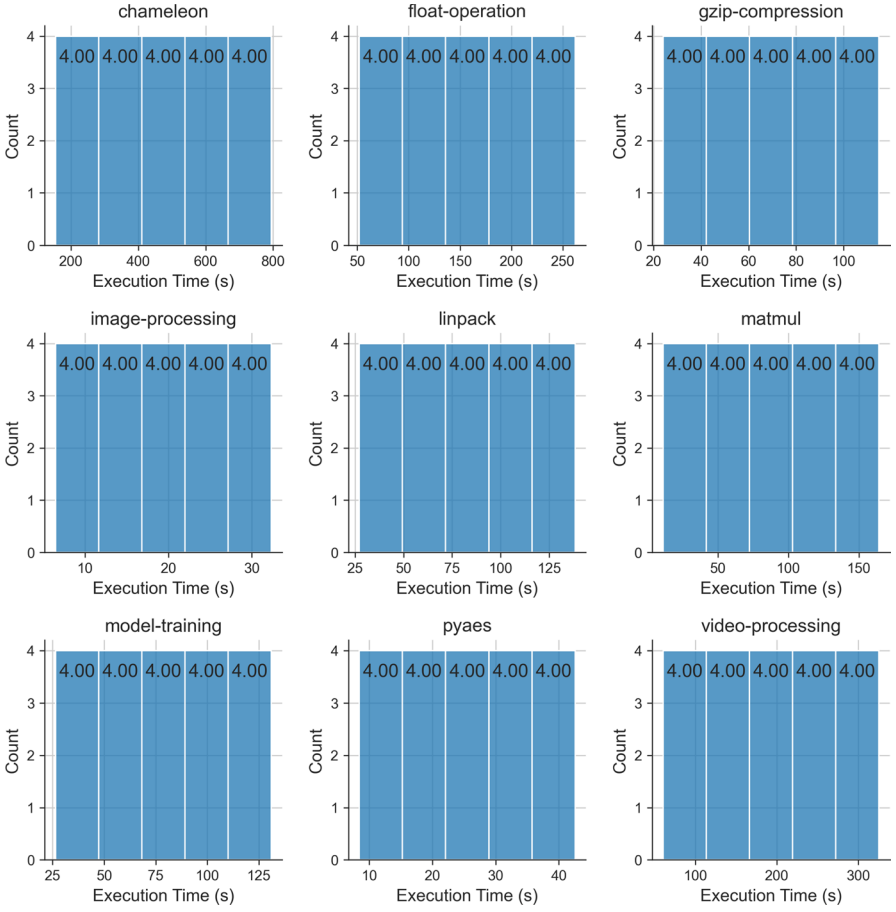

Fig. 7 Number of concurrent executions per container instance

To test the number of parallel executions per container instance for single threaded workloads at any given point in time, we have plotted the response times on a histogram, presented in Fig. 7. It is clearly visible that in all cases, OpenFaaS forks up to 4 different function processes in the same container. These results are consistent with the ones obtained in [33] where the lightweight version of OpenFaaS, FaaSd has been analyzed. It should be noted, however, that this behavior is configurable, but we decided to evaluate the default configuration across all different tests.

4.4 Parallel execution with automatic replica scaling

The real benefit of the serverless paradigms is the option to seamlessly scale the number of function instances depending on the current load. One of the reasons why OpenFaaS was selected as the platform for evaluating the three different Kubernetes

distributions is the fact that it can leverage two diverse scaling mechanisms. This allowed us to test the scaling behavior and performance when different parameters were used for driving the scaling decisions.

In the following subsections, we outline how these two strategies work and describe the results that we have obtained during the testing.

4.4.1 OpenFaaS native automatic replica scaling

The default scaling strategy which comes preconfigured on each new OpenFaaS install is based on function metrics which are gathered by the popular Prometheus monitoring tool [69] and then evaluated using Alertmanager [70]. Once the configured threshold is reached, an alert fires which in turn executes the required API call for increasing the number of function replicas.

Even though this approach is extremely flexible, allowing users to write scaling rules based on the various metrics that Prometheus scrapes related to each function, the rule which is enabled by default on new installations is somewhat binary in nature. The default rule tests the number of successful invocations per second for the last 10 seconds, and if this number is larger than 5, it scales up the number of function instances up to a preconfigured maximum. This means that even when executing requests at a constant rate, say 6 per second, it is possible to obtain the maximum number of allowed replicas, same as when testing with a much larger number of requests per second, for example 30 or 40. We have tested this behavior by executing 1 request per second from 6 concurrent workers for more than 200 seconds, successfully reaching the defined threshold for the maximum number of replicas.

In summary, this behavior of not taking into account the current number of deployed replicas, instead solely relying on the number of requests per second, leads to suboptimal scaling decisions. Under a consistent load, it either scales to the maximum number of configured replicas or it does not scale at all. However, as a result of using well-known third-party applications for the actual monitoring and rule definitions, administrators should be able to change the default behavior with relative ease, thus better reflecting their needs.

4.4.2 Leveraging the Kubernetes horizontal pod autoscaler

The alternative method for automatic scaling of function instances supported by OpenFaaS is by using the Kubernetes Horizontal Pod Autoscaler (HPA). HPA is the native Kubernetes scaling mechanism and by default relies on metrics acquired via the metrics server component, namely CPU and memory utilization. Different autoscaling profiles can be associated with different functions, catering to their runtime specifics. In our tests, we have configured a custom HPA profile which fired whenever the *float-operation* function has used more than 350 CPU shares (0.35 of a core). With testing, it was determined that when executing the *float-operation* function using 100,000 as the input parameter, and not 10,000,000 as was the case with previous benchmark executions, 350 CPU shares are utilized. These 350 shares accounted for four parallel executions per container instance, the default value as discussed previously in Sect. 4.3.

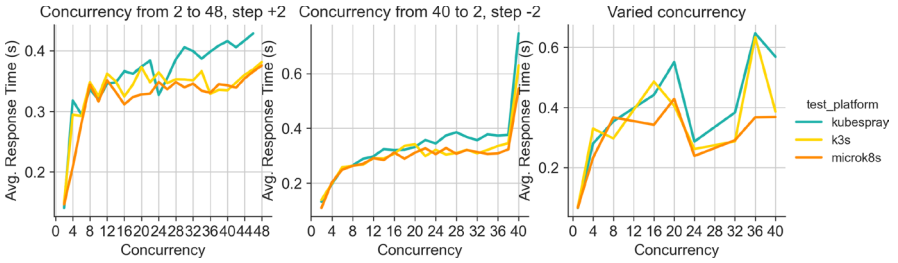

Fig. 8 Average response time for float-operation using various concurrency levels across the different distributions

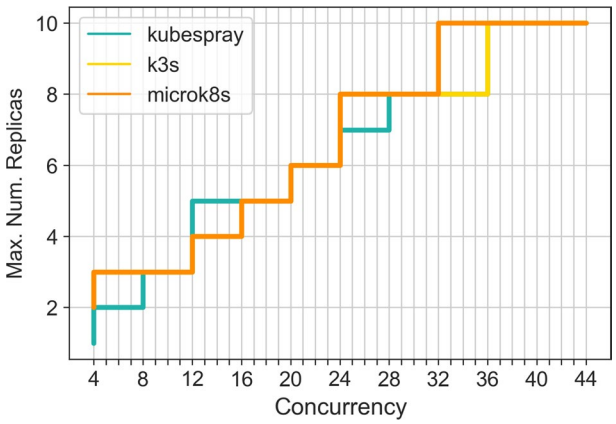

Fig. 9 Change in number of replicas based on requests per second

To evaluate the behavior of the HPA, we have executed the *float-operation* function multiple times with varying concurrency levels on each Kubernetes distribution. Figure 8 shows the results for three different execution strategies:

1. Start from 1 function replica, execute 2 concurrent requests per second, increasing the concurrency rate by 2 every 5 min, until 48 requests per second are achieved.
2. Start from 1 function replica, execute 40 concurrent requests per second, decreasing the concurrency rate by 2 every 5 min, until 2 requests per second are achieved.
3. Start from 1 function replica and vary the number of concurrent requests every 5 min using the following strategy: 8, 1, 20, 4, 40, 24, 1, 4, 16, 1, 36, 32.

The idea behind the three different execution scenarios defined above is to test how the HPA behaves under different circumstances. Both (1) and (2) gradually increase and decrease the load respectively, while (3) significantly varies it every 5 min, forcing more dramatic changes in the number of active function instances.

Kubespray exhibits higher response times across the three presented tests in Fig. 8, while the results obtained from K3s and MicroK8s are similar, albeit with MicroK8s having a noticeable advantage in terms of the varied workload.

To better gauge the auto-scaling behavior, Fig. 9 presents the concurrency rate versus the number of replicas during that period. These results are obtained from a test where the initial number of replicas for the function was 1 and the execution started with 4 concurrent requests per second, increased by 4 every 5 min. This strategy was chosen because theoretically it should lead to a single replica increase every 5 min, or at every concurrency increase, as a result of the auto scaling policy elaborated previously.

As with all previous tests, the auto-scaling behavior of K3s and MicroK8s is closely matched, showing a difference in only a single case. MicroK8s increases the number of replicas slightly faster when faced with 32 concurrent requests per second, while K3s follows with the same increase when 36 requests per second are executed. Kubespray lags behind the other two platforms in two cases, while pulling ahead in one.

5 Discussion and conclusion

Using a set of existing serverless benchmarks executed on the OpenFaaS serverless platform, we have tested the performance characteristics of three different Kubernetes distributions: Kubespray which deploys a full-fledged Kubernetes cluster, as well as K3s and MicroK8s, both aimed at resource constrained devices placed on the network edge.

Our results show that the more lightweight Kubernetes distributions offer better performance on bare-metal devices where advanced integrations with third-party cloud systems are not required. By simply removing the unnecessary components and altering the default persistent store used by regular Kubernetes clusters, both K3s and MicroK8s have not only managed to reduce the deployment time and complexity but have also improved performance. This is consistent across all four different test types that were conducted.

When it comes to the cold start delay, the full-fledged Kubernetes cluster deployed using Kubespray takes longer to instantiate new function instances. This leads to larger delays during the initial execution of a function which has been scaled down to zero replicas. The results obtained from K3s and MicroK8s are comparable, and there is no evidence to suggest that they differ significantly from one another.

The results from the serial execution, where each function has been serially executed for a period of 5 min also paint a similar picture. There is no evidence to suggest a significant performance difference in only one of the 14 relevant tests between the three Kubernetes distributions. However, when comparing only the K3s and MicroK8s platforms among themselves, which have shown more similar results compared to Kubespray, the null hypothesis failed to be rejected in only 4 of the 14 relevant cases. Of the remaining 10, in one half of those K3s exhibits better performance, while in the other MicroK8s.

Kubespray manages to perform better than the more lightweight alternatives in 6 of the 14 tests when under extreme load from 20 simultaneous requests being executed against a single function replica. This same test has allowed us to determine that in the default OpenFaaS configuration for our architecture, 4 concurrent requests can be served by a single function container instance.

Finally, we have also tested two modes of automatic scaling supported by OpenFaaS and their performance on different Kubernetes distributions. The first scaling strategy which is enabled by default and relies on external monitoring components offers a binary scaling mechanism which is not suitable for cases where there is a consistent load for longer periods of time. On the other hand, using the native Kubernetes Horizontal Pod Autoscaler, MicroK8s exhibits lower response times.

In conclusion, Kubernetes distributions that are explicitly optimized for resource constrained devices show better performance when it comes to cold start latency and certain families of disk throughput tests. One such example are the sequential read and write tests conducted using the dd tool where a traditional Kubespray deployment shows 22% decrease in the number of total executions during a 5 min window, and a 28% increase in the average response time. The evident volatility of the Kubespray results compared to the other distributions can be attributed to a number of factors. Firstly, both K3s and MicroK8s have omitted non-essential components used for interacting with external systems and have opted to use more lightweight database backends based on SQLite instead of etcd, impacting both disk performance and idle resource usage. Further optimizations such as the colocation of multiple different components into a single process, as is the case with K3s, have also positively affected performance, and have led to a reduction of their overall resource footprint. These improvements are evident in certain CPU bound benchmarks as well. The traditional Kubernetes deployment experienced higher average response times in both serial and parallel executions for matrix multiplication and linear equations solving. However, there are also workloads where Kubespray offers comparative performance, such as video processing during parallel and serial executions, where no statistically significant difference was detected.

Even though the optimizations made by both K3s and MicroK8s offer better performance for the majority of serverless edge workloads, the compromises of doing so must be taken into account, such as the reduced integration capabilities or limited scalability when it comes to large numbers of nodes. Careful consideration is needed when deciding which database to use for the control plane in cases where a different Kubernetes distribution is used, especially when it applies to large clusters.

Great progress was achieved in the last few years in regards to simplifying Kubernetes deployments and optimizing the container orchestrator for different use-cases. We expect this to continue in the future as well, combined with additional techniques to reduce the significant cold start latency of new containers, typical for this runtime technology.

6 Threats to validity

We have strived to eliminate as many potential threats to validity as possible, using dedicated bare-metal infrastructure with exact hardware characteristics between the different nodes across all tests. Furthermore, all network communication during the tests was done on an isolated network segment and did not connect to external destinations, eliminating external delays influencing the results. OpenFaaS also published guidelines on how to properly configure the platform for maximum performance [71], which we have followed. However, it must be pointed out that different strategies in terms of routing the requests to the function instance may produce different results. In our case, we have routed all requests first to the function gateway instead of directly to the function itself, in order to ensure a fairer load-balancing between multiple replicas [72]. It should be recognized though that this method might have introduced additional latency in comparison to direct invocation. Nevertheless, the same strategy was used for all different Kubernetes distributions, thus largely canceling out any potential impacts when comparing the results between themselves.

Acknowledgements The work presented in this paper has received funding from the Faculty of Computer Science and Engineering under the "SCAP" project.

References

1. Mell P, Grance T (2011) The NIST definition of cloud computing. Technical report NIST special publication (SP) 800-145, National institute of standards and technology. https://doi.org/10.6028/NIST.SP.800-145
2. Armbrust M, Fox A, Griffith R, Joseph A, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zaharia M (2009) Above the clouds, A Berkeley view of cloud computing. http://bnrg.eecs.berkeley.edu/randy/Papers/RHKPubs07-11/A. Accessed 26 Dec 2021
3. Duan Y, Fu G, Zhou N, Sun X, Narendra NC, Hu B (2015) Everything as a service (XaaS) on the cloud, origins current and future trends. In: 2015 IEEE 8th International Conference on Cloud Computing, pp 621–628. https://doi.org/10.1109/CLOUD.2015.88
4. Jonas E, Schleier-Smith J, Sreekanti V, Tsai C.-C, Khandelwal A, Pu Q, Shankar V, Carreira J, Krauth K, Yadwadkar N, Gonzalez J.E, Popa R.A, Stoica I, Patterson DA (2019) Cloud programming simplified, A Berkeley view on serverless computing. arXiv:1902.03383 [cs]
5. Kratzke N (2018) A brief history of cloud application architectures. Appl Sci 8(8):1368. https://doi.org/10.3390/app8081368
6. Eismann S, Scheuner J, van Eyk E, Schwinger M, Grohmann J, Herbst N, Abad CL, Iosup A (2021) Serverless applications, why when, and how? IEEE Softw 38(1):32–39. https://doi.org/10.1109/MS.2020.3023302
7. Bittencourt L, Immich R, Sakellariou R, Fonseca N, Madeira E, Curado M, Villas L, DaSilva L, Lee C, Rana O (2018) The Internet of Things fog and cloud continuum: integration and challenges. Internet of Things 3–4:134–155. https://doi.org/10.1016/j.iot.2018.09.005
8. Varghese B, Buyya R (2018) Next generation cloud computing: new trends and research directions. Future Gener Comput Syst 79:849–861. https://doi.org/10.1016/j.future.2017.09.020
9. Pfandzelter T, Bermbach D (2019)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. https://doi.org/10.1109/ICFC.2019.00033
10. Carvalho G, Cabral B, Pereira V, Bernardino J (2021) Edge computing: current trends, research challenges and future directions. Computing 103(5):993–1023. https://doi.org/10.1007/s00607-020-00896-5

11. AWS IoT Greengrass: Amazon Web Services. https://aws.amazon.com/greengrass/ Accessed 26 April 2021
12. IoT Hub | Microsoft Azure. https://azure.microsoft.com/en-us/services/iot-hub/ Accessed 26 April 2021
13. Das A, Patterson S, Wittie M (2018) EdgeBench, benchmarking edge computing platforms. In: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), pp 175–180. IEEE Zurich. https://doi.org/10.1109/UCC-Companion.2018.00053
14. Li J, Kulkarni S.G, Ramakrishnan KK, Li D (2021) Analyzing open-source serverless platforms, characteristics and performance, pp 15–20. https://doi.org/10.18293/SEKE2021-129
15. Kjorveziroski V, Canto CB, Roig PJ, Gilly K, Mishev A, Trajkovik V, Filiposka S (2021) IoT serverless computing at the edge, open issues and research direction. Trans Netw Commun 9(4):1–33. https://doi.org/10.14738/tnc.94.11231
16. Gadepalli P.K, McBride S, Peach G, Cherkasova L, Parmer G (2020) Sledge: a Serverless-first, Light-weight Wasm Runtime for the Edge. In: Proceedings of the 21st International Middleware Conference. Middleware '20, pp 265–279. Association for Computing Machinery New York. https://doi.org/10.1145/3423211.3425680
17. 2021 Kubernetes Adoption Survey. https://www.purestorage.com/content/dam/pdf/en/analyst-reports/ar-portworx-pure-storage-2021-kubernetes-adoption-survey.pdf Accessed 26 Dec 2021
18. Kjorveziroski V, Filiposka S, Trajkovik V (2021) IoT serverless computing at the edge. A systematic mapping review. Computers 10(10):130. https://doi.org/10.3390/computers10100130
19. Risco S, Moltó G, Naranjo DM, Blanquer I (2021) Serverless workflows for containerised applications in the cloud continuum. J Grid Comput 19(3):30. https://doi.org/10.1007/s10723-021-09570-2
20. Kayal P (2020) Kubernetes in fog computing, feasibility demonstration limitations and improvement scope: invited paper. In: 2020 IEEE 6th World Forum on Internet of Things (WF-IoT), pp 1–6. https://doi.org/10.1109/WF-IoT48130.2020.9221340
21. K3s: Lightweight Kubernetes. https://k3s.io/ Accessed 05 Sept 2021
22. MicroK8s-Zero-ops Kubernetes for developers, edge and IoT | MicroK8s. http://microk8s.io Accessed 05 Sept 2021
23. Software conformance(Certified Kubernetes). https://www.cncf.io/certification/software-conformance/ Accessed 26 Dec 2021
24. Martins H, Araujo F, da Cunha PR (2020) Benchmarking serverless computing platforms. J Grid Comput 18(4):691–709. https://doi.org/10.1007/s10723-020-09523-1
25. Gan Y, Zhang Y, Cheng D, Shetty A, Rathi P, Katarki N, Bruno A, Hu J, Ritchken B, Jackson B, Hu K, Pancholi M, He Y, Clancy B, Colen C, Wen F, Leung C, Wang S, Zaruvinsky L, Espinosa M, Lin R, Liu Z, Padilla J, Delimitrou C (2019) An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '19, pp 3–18. Association for Computing Machinery New York. https://doi.org/10.1145/3297858.3304013
26. Wen J, Liu Y, Chen Z, Chen J, Ma Y (2021) Characterizing commodity serverless computing platforms. J Softw: Evol Process. https://doi.org/10.1002/smr.2394
27. Scheuner J, Leitner P (2020) Function-as-a-service performance evaluation: a multivocal literature review. J Syst Softw. https://doi.org/10.1016/j.jss.2020.110708
28. Hellerstein JM, Faleiro J, Gonzalez JE, Schleier-Smith J, Sreekanti V, Tumanov A, Wu C (2018) Serverless computing, one step forward two steps back. In: CIDR 20019 Monterey, CA. http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf Accessed 09 Jan 2022
29. Maissen P, Felber P, Kropf P, Schiavoni V (2020) FaaSdom, a benchmark suite for serverless computing. In: Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems, pp 73–84. https://doi.org/10.1145/3401025.3401738
30. Bschitter: benchmark-suite-serverless-computing (2021). https://github.com/Bschitter/benchmark-suite-serverless-computing Accessed 24 Dec 2021
31. Eismann S, Costa DE, Liao L, Bezemer C-P, Shang W, van Hoorn A, Kounev S (2021) A case study on the stability of performance tests for serverless applications. arXiv:2107.13320 [cs]
32. Tzenetopoulos A, Apostolakis E, Tzomaka A, Papakostopoulos C, Stavrakakis K, Katsaragakis M, Oroutzoglou I, Masouros D, Xydis S, Soudris D (2021) FaaS and curious, performance implications of serverless functions on edge computing platforms. In: Jagode H, Anzt H, Ltaief H, Luszczek P (eds) High performance computing. Lecture notes in computer science. Springer Cham, pp 428–438. https://doi.org/10.1007/978-3-030-90539-2_29

33. Kjorveziroski V, Filiposka S, Trajkovik V (2021) Serverless platforms performance evaluation at the network edge. In: 13th ICT Innovations Conference 2021 Skopje, North Macedonia
34. Wang I, Liri E, Ramakrishnan KK (2020) Supporting IoT applications with serverless edge clouds. In: 2020 IEEE 9th International Conference on Cloud Networking (CloudNet), pp 1–4.https://doi.org/10.1109/CloudNet51028.2020.9335805
35. Agarwal S, Rodriguez MA, Buyya R (2021) 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. https://doi.org/10.1109/CCGrid51090.2021.00097
36. Wang B, Ali-Eldin A, Shenoy P (2020) 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. https://doi.org/10.1145/3431379.3460646
37. Eiermann A, Renner M, Großmann M, Krieger UR (2017) On a fog computing platform built on ARM architectures by docker container technology. In: Eichler G, Erfurth C, Fahrnberger G (eds) Innovations for community services. Communications in computer and information science. Springer, Cham, pp 71–86. https://doi.org/10.1007/978-3-319-60447-3_6
38. Kim J, Lee K (2019) FunctionBench, a suite of workloads for serverless cloud function service. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp 502–504. IEEE Milan, Italy. https://doi.org/10.1109/CLOUD.2019.00091
39. kmu-bigdata/serverless-faas-workbench. BigData Lab. in KMU (2021). https://github.com/kmu-bigdata/serverless-faas-workbench Accessed 09 May 2021
40. Aslanpour MS, Toosi AN, Cicconetti C, Javadi B, Sbarski P, Taibi D, Assuncao M, Gill SS, Gaire R, Dustdar S (2021) Serverless edge computing, vision and challenges. In: 2021 Australasian Computer Science Week Multiconference. ACSW '21, pp 1–10. Association for Computing Machinery New York, NY, USA. https://doi.org/10.1145/3437378.3444367
41. OpenFaaS—Serverless Functions Made Simple with Kubernetes. https://www.openfaas.com/ Accessed 26 April 2021
42. Knative. https://knative.dev/ Accessed 10 April 2021
43. Nuclio. https://nuclio.io/ Accessed 27 April 2021
44. Kubeless. https://kubeless.io/ Accessed 26 April 2021
45. openfaas/faas. OpenFaaS (2021). https://github.com/openfaas/faas Accessed 26 Dec 2021
46. Autoscaling - OpenFaaS. https://docs.openfaas.com/architecture/autoscaling/ Accessed 26 Dec 2021
47. Nguyen T-T, Yeom Y-J, Kim T, Park D-H, Kim S (2020) Horizontal Pod autoscaling in kubernetes for elastic container orchestration. Sensors 20(16):4621. https://doi.org/10.3390/s20164621
48. Container Storage Interface (CSI) for Kubernetes GA (2019). https://kubernetes.io/blog/2019/01/15/container-storage-interface-ga/ Accessed 26 Dec 2021
49. Kumar R, Trivedi MC (2021) Networking analysis and performance comparison of kubernetes CNI plugins. In: Bhatia SK, Tiwari S, Ruidan S, Trivedi MC, Mishra KK (eds) Advances in computer communication and computational sciences. Advances in intelligent systems and computing. Springer, Singapore, pp 99–109. https://doi.org/10.1007/978-981-15-4409-5_9
50. Kubespray—Deploy a Production Ready Kubernetes Cluster. https://kubespray.io/#/ Accessed 26 Dec 2021
51. Galal H, Introduction to K3s. https://www.suse.com/c/rancher_blog/introduction-to-k3s/ Accessed 18 Feb 2022
52. K3s System Requirements. https://rancher.com/docs/k3s/latest/en/installation/installation-requirements/ Accessed 17 Feb 2022
53. Kubeadm System Requirements. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/ Accessed 17 Feb 2022
54. High availability (HA) | MicroK8s. http://microk8s.io Accessed 18 Feb 2022
55. MicroK8s System Requirements. http://microk8s.io Accessed 17 Feb 2022
56. MicroK8s 1.23 Release Notes. https://github.com/ubuntu/microk8s/releases Accessed 27 Dec 2021
57. OpenFaaS Helm Chart for Kubernetes. https://github.com/openfaas/faas-netes Accessed 26 Dec 2021
58. Dogan J (2021) rakyll/hey. https://github.com/rakyll/hey Accessed 26 Dec 2021
59. Park J, Kim H, Lee K. (2020) Evaluating concurrent executions of multiple function-as-a-service runtimes with MicroVM. In: 2020 IEEE 13th International Conference on Cloud Computing

(CLOUD), pp 532–536. IEEE Beijing, China. https://doi.org/10.1109/CLOUD49709.2020.00080. https://ieeexplore.ieee.org/document/9284320/ Accessed 17 Feb 2022
60. Ustiugov D, Petrov P, Kogias M, Bugnion E, Grot B. (2021) Benchmarking, analysis, and optimization of serverless function snapshots. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp 559–572. ACM Virtual USA. https://doi.org/10.1145/3445814.3446714. Accessed 17 Feb 2022
61. Chadha M, Jindal A, Gerndt M (2021) Architecture-specific performance optimization of compute-intensive FaaS functions. arXiv: 2107.10008. Accessed 17 Feb 2022
62. Choi J, Lee K (2020) Evaluation of network file system as a shared data storage in serverless computing. In: Proceedings of the 2020 Sixth International Workshop on Serverless Computing, pp 25–30. ACM Delft Netherlands. https://doi.org/10.1145/3429880.3430096. Accessed 17 Feb 2022
63. Zhao L, Yang Y, Li Y, Zhou X, Li K (2021) Understanding, predicting and scheduling serverless workloads under partial interference. In: Proceedings of the International Conference for High Performance Computing Networking Storage and Analysis. SC '21, pp 1–15. Association for Computing Machinery New York, NY, USA. https://doi.org/10.1145/3458817.3476215. Accessed 16 Feb 2022
64. of-watchdog. OpenFaaS (2021). https://github.com/openfaas/of-watchdog Accessed 26 Dec 2021
65. korvoj/k8s-distributions-iot-edge: Kubernetes distributions for the edge, serverless performance evaluation. https://github.com/korvoj/k8s-distributions-iot-edge Accessed 18 Feb 2022
66. faasd - OpenFaaS. https://docs.openfaas.com/deployment/faasd/ Accessed 28 Feb 2022
67. Patman J, Chemodanov D, Calyam P, Palaniappan K, Sterle C, Boccia M (2020) Predictive cyber foraging for visual cloud computing in large-scale IoT systems. IEEE Trans Netw Serv Manag 17(4):2380–2395. https://doi.org/10.1109/TNSM.2020.3010497
68. Cho C, Shin S, Jeon H (2020) QoS-aware workload distribution in hierarchical edge clouds. A reinforcement learning approach. IEEE Access 8. https://doi.org/10.1109/ACCESS.2020.3033421
69. Prometheus: Prometheus—Monitoring system & time series database. https://prometheus.io/ Accessed 26 Dec 2021
70. Prometheus: Alertmanager | Prometheus. https://prometheus.io/docs/alerting/latest/alertmanager/ Accessed 26 Dec 2021
71. Performance—OpenFaaS. https://docs.openfaas.com/architecture/performance/ Accessed 26 Dec 2021
72. OpenFaaS Endpoint Load-Balancing. https://github.com/openfaas/faas-netes Accessed 26 Dec 2021