algorithms

MDPI

Article

Performance Evaluation of Open-Source Serverless Platforms for Kubernetes

Jonathan Decker [1,*], Piotr Kasprzak [2] and Julian Martin Kunkel [1,2]

[1] Institute for Computer Science, Universität Göttingen, Goldschmidtstraße 7, 37077 Göttingen, Germany; julian.kunkel@gwdg.de
[2] GWDG, Burckhardtweg 4, 37077 Göttingen, Germany; piotr.kasprzak@gwdg.de
* Correspondence: jonathan.decker@uni-goettingen.de

Abstract: Serverless computing has grown massively in popularity over the last few years, and has provided developers with a way to deploy function-sized code units without having to take care of the actual servers or deal with logging, monitoring, and scaling of their code. High-performance computing (HPC) clusters can profit from improved serverless resource sharing capabilities compared to reservation-based systems such as Slurm. However, before running self-hosted serverless platforms in HPC becomes a viable option, serverless platforms must be able to deliver a decent level of performance. Other researchers have already pointed out that there is a distinct lack of studies in the area of comparative benchmarks on serverless platforms, especially for open-source self-hosted platforms. This study takes a step towards filling this gap by systematically benchmarking two promising self-hosted Kubernetes-based serverless platforms in comparison. While the resulting benchmarks signal potential, they demonstrate that many opportunities for performance improvements in serverless computing are being left on the table.

Keywords: serverless; open source; Kubernetes; benchmark; performance; self-hosted; HPC

Citation: Decker, J.; Kasprzak, P.; Kunkel, J.M. Performance Evaluation of Open-Source Serverless Platforms for Kubernetes. *Algorithms* **2022**, *15*, 234. https://doi.org/10.3390/a15070234

Academic Editors: Yunquan Zhang and Liang Yuan

Received: 27 May 2022
Accepted: 28 June 2022
Published: 2 July 2022

1. Introduction

The market for serverless computing (often simply called 'serverless') has exploded in popularity over the past few years [1]. The main drivers behind this growth are offerings by large cloud providers (e.g., Amazon Web Services (AWS), Google, Microsoft) that have been adopted for many use cases, such as glue code, websites, Application Programming Interfaces (APIs), and many more. These offerings are attractive as they remove the need for developers to set up infrastructure to handle logging, scaling, and monitoring for their code. Instead, the cloud providers provide these functionalities 'out of the box'. Another selling point of serverless is its pricing model. It is commonly cheaper for users to only pay for resources that they have actually consumed, compared to paying for reserved resources [2].

Serverless is of interest in other areas as well, such as scientific and high-performance computing (HPC), as it can potentially provide scalable generic computing instances to solve scientific computing tasks [3]. However, serverless offerings by cloud providers, called Function-as-a-Service (FaaS), are often very restrictive in terms of execution time, memory, and configurability. Notwithstanding this, there exist many open-source serverless platforms (e.g., Knative: https://knative.dev/, Nuclio: https://nuclio.io/, OpenFaaS: https://openfaas.com/, Fission: https://fission.io/, OpenWhisk: https://openwhisk.apache.org/ accessed on 27 May 2022) that can be deployed on-premise and allow for more configuration and customization compared to the public cloud. Administrators of HPC clusters may find serverless to be advantageous, compared to reservation-based workload managers such as Slurm (https://slurm.schedmd.com/ accessed on 27 May 2022) due to the more fine-grained resource sharing capabilities of serverless. In serverless, the workloads of multiple users can commonly run safely next to each other on the same

CPU (or GPU) due to isolation technologies such as container isolation. Thus, computing resources can be shared dynamically by any number of users, instead of being reserved by one user at a time.

Nevertheless, in practice it is not clear whether open-source serverless platforms can live up to the performance requirements of HPC workloads such that they are a valid alternative. Furthermore, serverless employs a different execution model compared to common distributed computing clusters, which usually depend on the Message Passing Interface (MPI) (https://www.mpi-forum.org/ accessed on 27 May 2022) to coordinate computing efforts, necessitating either the adoption of applications and programs that are designed to operate on serverless platforms or the adaptation of existing applications and programs to the serverless model. Thus, switching to serverless could incur additional costs and work requirements beyond setting up a serverless platform. This makes it necessary to ensure that the chosen open-source serverless platform is able to deliver the desired improvements in resource utilization by benchmarking these platforms against each other.

Despite this, there is currently a distinct lack of comparative studies on open-source serverless platforms. Most studies of serverless platforms focus on the most popular instance, AWS, which is present in 88% of serverless studies, followed by Microsoft (26%) and Google (23%), with open-source platforms combined amounting to only 14% according to Scheuner et al. [4]. Even fewer studies have made an effort to compare the performance of multiple open-source platforms. Moreover, it is difficult to produce comparable benchmarks for serverless platforms, as there are no standardized benchmarks for them; due to the many variations in the architectures of serverless platforms, it is difficult to design universal benchmarks [5]. This paper focuses on two open-source platforms running on top of Kubernetes (https://kubernetes.io/ accessed on 27 May 2022), namely, OpenFaaS (https://openfaas.com/ accessed on 27 May 2022) and Nuclio (https://nuclio.io/ accessed on 27 May 2022).

In this paper, Kubernetes is used as a basis for running the serverless platforms under study for two reasons. First, Kubernetes is either supported by or even depended on by almost all open-source serverless platforms. Second, Beltre et al. [6] investigated whether Kubernetes is a valid base for HPC workloads and found that, for the majority of use cases, it can keep up with bare metal in terms of performance. The goal of this paper is to identify performance differences between OpenFaaS and Nuclio when running sample workloads and to investigate what aspects or features of these platforms cause these differences. The focus is on data processing workloads with either a high data throughput or high computational cost; while these might not reflect the most common use cases for serverless, they are, nevertheless, interesting for optimizing the platforms for HPC.

1.1. Background

Serverless is described in detail by [1,2]; in summary, it is an execution model for cloud applications that abstracts away concrete servers. This means that developers do not know exactly which machine their code will be run on. Furthermore, serverless applications are composed of serverless functions that usually handle a single task and become active when invoked through a certain type of event. Serverless functions are commonly embedded into function environments to form function images (commonly container images). These images can then be used to create function instances, which are usually just container instances. Depending on the current amount of incoming load for a serverless function, the serverless platform can automatically scale the number of active function instances by starting new instances or terminating existing ones and load balancing the incoming requests among all active instances.

Kubernetes fits well with serverless, as it provides all the necessary technologies and could itself even be considered as accidentally serverless. Installing a serverless platform, such as OpenFaaS or Nuclio, on top of Kubernetes adds workflows for creating serverless functions, uploading them to the platform and adding endpoints that invoke them. Moreover, the serverless platforms provide templates for function environments that

must be used to build function images. As Python and Javascript are the most popular programming languages for building serverless functions (See DataDog State of Serverless 2021: https://www.datadoghq.com/state-of-serverless/ accessed on 27 May 2022), most serverless platforms offer well-developed templates for these languages. C/C++ is far less popular for serverless computing, and only few serverless platforms offer dedicated support for it, meaning that users must often fall back on general-purpose templates, if any are available.

1.2. Related Work

In a literature assessment of serverless performance studies by Kuhlenkamp et al. [7], the authors show that studies of serverless platforms do exist; however, serverless as a scientific field is developing rapidly, and scientific finding and publishing cannot keep up with the speed of serverless innovation. Therefore, more studies are required to understand the effects of features and design choices on serverless platforms.

Van Eyk et al. [8] analyzed many serverless platforms and formulated a reference architecture based on the similarities they found. This reference architecture makes it easier to discuss differences in the feature sets of serverless platforms; nevertheless, the authors argue that closed-source platforms (e.g., AWS, Microsoft, Google) limit understanding of the internal workings of these platforms.

Scheuner et al. [4], Kuhlemkamp et al. [7], and Eismann et al. [9] have all pointed out that studies of serverless platforms largely focus on public cloud platforms, especially AWS, and neglect open-source self-hosted alternatives. Scheuner et al. [4] further argue that studies of self-hosted serverless platforms are more complicated, as researchers have full control over the platforms under study and must set up and configure them on their own. Moreover, researchers must document all of these steps in order to ensure the comparability and reproducibility of their studies, while public cloud offerings work out of the box.

Not only are serverless offerings in the public cloud partially a black box, they are constantly changing. Public cloud vendors are continually developing and improving their platforms in order to better satisfy their customers, which in turn makes it difficult or even impossibly to fully replicate studies on them. Furthermore, it is not possible to test a study against a specific version of a public cloud platform. A comparative study of a public serverless platform, such as that carried out by Jain et al. [10], who analyzed multiple serverless frameworks for AWS, loses relevance as the platform it was tested on irreversibly changes away from the version that was studied.

Spillner et al. [3] argued that serverless has potential for HPC. A study by Mallah et al. [11] further confirms this point, as they tested the viability of using serverless public cloud offerings instead of virtual machines (VMs) on their HPC cluster. They found that while this approach is very much valid, it performs best when running embarrassingly parallel tasks. Furthermore, it removes the complexity of having to set up VMs and application runtimes by packaging the applications and runtimes as containers. This last point has been confirmed by Zhou et al. [12], who investigated using Kubernetes and containers for HPC and found that it significantly improved portability and eased the installation of dependencies.

1.3. Contributions

This study makes the following contributions:

- A comparative assessment of two open-source Kubernetes-based serverless platforms. Initially, it was planned to evaluate additional platforms; however, due to issues that prevented meaningful benchmarking and assessment of these platforms, they were left out, further demonstrating the need to evaluate performance systematically. For instance, with respect to Fission (https://fission.io/ accessed on 27 May 2022), we submitted an issue (https://github.com/fission/environments/issues/157 accessed on 27 May 2022) that was later fixed by the community.

- Two serverless workloads designed with practical use cases in mind that can be configured to systematically test individual system components, and can easily be ported to additional platforms.
- A discussion of the implications that can be drawn from our benchmark results on the performance capabilities of serverless platforms and how these can be improved.

A more general overview of this work can be found in this master's thesis [13].

1.4. Outline

The rest of the paper is structured as follows: Section 2 explains the experimental setup, design of the tested workloads, and how the benchmarking was conducted; in Section 3, the performance of the platforms under study as measured through the benchmarks is presented; in Section 4, the implications of our results are discussed in relation to the features and flaws of the platforms; finally, the contributions and implications of this paper are summarized in Section 5.

2. Materials and Methods

2.1. Methods Outline

The overall goal of this study was to determine how well the two serverless platforms, OpenFaaS and Nuclio, would perform when handling HPC workloads. OpenFaaS was chosen because it is currently the most popular open-source serverless platform as measured in GitHub stars (https://github.com/openfaas/faas accessed on 27 May 2022). Nuclio was picked because it presents itself as a data processing-focused platform, and thus was expected to do well in an HPC context.

Evaluating the performance of the two platforms required designing and setting up benchmarks around representative workloads as well as experimenting with different settings for the platforms in order to find the optimal configuration. Furthermore, using different configurations helps to understand how features of the platforms can help them to excel at the benchmarks. Except for the serverless platforms deployed, the setups should be as identical as possible in order to ensure fairness when comparing the benchmarks.

To remain in line with the HPC context, the workloads should be implemented in C/C++, as this is a very common language in HPC software and is considered to perform relatively well. Nevertheless, this should not fully exclude alternative implementations if there is reason to suspect they will be able to deliver even better performance. Here, the main configuration variation in the experiments was the number of function instances used. Normally, serverless platforms handle scaling automatically by adding or removing function instances depending on the incoming load for a specific function. In our experiments, the number of function instances was kept constant for each benchmark run in order to maintain a focus on the platform's ability to enable the performance of their function instances rather than the ability of their auto-scaling systems to spawn new instances.

While the two platforms under study might be close or very different in terms of performance, this may nonetheless be far below anything that would be useful for real-world HPC use cases. Therefore, it was necessary to find a baseline for these new benchmarks as to what performance could be expected. Optimally, this would be done by running the benchmarks on an HPC cluster. However, due to limitations during the execution of our experiments, the baselines for the benchmarks were instead determined by executing them on a single host that had as many CPU cores to spare as were available to each of the serverless platforms. However, in such a comparison, it should be kept in mind that the platforms need to handle networking, run operating systems, run Kubernetes, and run themselves, while the bare-metal systems used for the baseline could carry out all computations locally with enough dedicated CPU cores.

2.2. Experiment Setup

The experiments were conducted using an OpenStack (https://openstack.org/ accessed on 27 May 2022) instance hosted by the GWDG (https://gwdg.de/ accessed

on 27 May 2022). OpenFaaS and Nuclio were each installed into a Kubernetes cluster of four VMs with four virtual CPU cores and 8 GB of memory per VM. Additionally, one more VM with 32 vCPU cores and 64 GB of memory was used as the function director (FD). The physical CPUs were AMD EPYC 2 7542 32-Core processors (https://www.amd.com/en/products/epyc accessed on 27 May 2022).

The network was handled by OpenStack and offered a theoretical upper speed limit of 10 Gbit/s between any two nodes. The speed between the FD VM and a container running in one of the Kubernetes clusters was measured at around 700 MB/s using iPerf (https://iperf.fr/ accessed on 27 May 2022). HPC workloads commonly use remote direct memory access (RDMA) technologies that enable distributed processes to bypass the CPU when writing to another host's memory. Such technologies were not used here, as the serverless platforms under study do not support RDMA; therefore, it was expected that the CPU speed of the VMs would limit the throughput rather than the network's bandwidth capacity.

Each request from the FD VM to a function instance running on any of the serverless platforms in the setup was handled as follows. First, the request is send out from the FD VM targeting the HTTP endpoint of a serverless function. The IP address of the endpoint resolves to the OpenStack load balancer, which in return forwards the request to the VMs running the Kubernetes cluster. Inside the cluster, the request is again load balanced by Kubernetes to the endpoints of the serverless platform, commonly referred to as Gateways. The gateways are responsible for resolving the URL path that identifies the target serverless function and load balancing incoming requests among all available serverless function instances. The experiment setup here used two gateway instances per cluster. Finally, the request reaches the VM running an available function instance, and is received by the web server of that function instance.

2.3. Workload Design

Two application-motivated workloads that could be easily ported to other FaaS systems were designed for the experiments.

First, an image processing workload that represents CPU-heavy tasks, which takes in images, performs a set of transformations on them, and outputs the resulting images. This workload is a placeholder for other workloads that do some form of data processing by taking in data, executing code on the data while relying on a C/C++ library, and saving the processed data. The image transformations were performed through ImageMagick (https://imagemagick.org accessed on 27 May 2022) using its C++ API. Furthermore, this workload comes in two variants, the regular processing variant and a null-processing (NP) variant that skips the transformations and sends back the images after loading them and sleeping for 1 s. The purpose of the NP variant is to provide an estimate of how the system would behave without having to perform the CPU-heavy task such that more throughput-focused tasks can be represented with this workload. The C++ implementations of the workload variants were the same for both OpenFaaS and Nuclio. The implementation of the regular variant can be seen in Appendix A.1. The C++ code was compiled with the following command: `c++ ‘Magick++-config --cxxflags --cppflags‘ -O2 -o imageproc imageproc.cpp ‘Magick++-config --ldflags --libs‘.`

The second sample workload focused on reversing of bytes as a simple task that requires fully loading the input before it can be sent back. The purpose of this workload is to additionally vary the number of bytes sent with each function invocation to determine the optimal input size for each platform. Moreover, this workload comes in two variants and two implementation languages. In the first variant, referred to as single, only a single request is sent and the time to respond to that request is measured. In the second variant, referred to as parallel, a number of requests are sent, similar to the image processing workload, and the time to process all requests is measured.

Implementation was carried out in C++ and in Go, and can be found in Appendices A.2 and A.3, respectively. As discussed earlier, the function implementations must be embedded

into function environments. Neither OpenFaaS nor Nuclio offers a dedicated C++ function environment, only general-purpose templates that can be used to deploy C++ functions. Internally, these templates work by running a web server written in Go, starting the user function by forking a sub process, and sending the input data to it via the standard input stream and receiving outputs via the standard output stream. When considering performance, this is not optimal, as it effectively means that the web server receives the input data and then has to copy it into the memory of the sub process.

This extra work could be avoided by allowing the user function to directly operate on the input data. The Nuclio template for Go does this by having its users implement their Go functions as Go plugins (https://pkg.go.dev/plugin accessed on 27 May 2022), which produces binaries that can be loaded at runtime by a Go program. This avoids the extra work of copying the input data, as these plugins become native extensions of the web server that can operate on the original input data. Therefore, it is reasonable to include a Go implementation of the workload for Nuclio. OpenFaaS uses the same streaming technique for Go templates as it does for the general purpose templates, thus, a Go implementation of the workload for OpenFaaS was left out. Furthermore, this was considered for the image processing workload; however, as it relies on ImageMagick, which is written in C, it has to use cgo (https://pkg.go.dev/cmd/cgo accessed on 27 May 2022) within a binder library (Go Imagick: https://github.com/gographics/imagick accessed on 27 May 2022) to translate the function calls, which would introduce new performance issues [14].

2.4. Test and Benchmark

A single benchmark run consisted of:

1. Setting up the correct number of function instances
2. Preparing the test data
3. Starting the benchmark and timer
4. Submitting the test data as a function invocations until all test data had been processed
5. Stopping the timer
6. Saving the results

The coordination of the benchmarks was carried out by a Python program running on the FD VM. The program sends in parallel requests to the HTTP endpoint of the function under test such that there is always enough load without overwhelming the Gateway of the serverless platform. Overall, each benchmark was repeated at least ten times.

For the image processing workload, the individual benchmarks were executed with 1, 2, 4, 8, and 16 active function instances in separate benchmark runs. The number of function instances was capped at 16 as each platform runs on four nodes with four vCPU cores each, meaning that with 16 active function instances, every function instance has on average one core to work with. The FD VM only tested the regular variant of this workload and not the NP variant.

The test data consisted of 1000 PNG images with a total size of 2.9 GiB taken from testimages.org [15]. The images were from the 16BIT RGB sampling set (https://sourceforge.net/projects/testimages/files/SAMPLING/16BIT/RGB accessed on 27 May 2022), with 960 of them having a resolution of 800×800 and the remaining 40 having a resolution of 600×600. This amounts to 3 MiB per image on average (1 GiB = 1024 MiB, that is, 2.9 GiB = 2969 MiB and $\frac{2969}{1000}$ MiB $\approx$ 3 MiB). The output had a total size of 842 MiB, which on average is 862 KiB per image. If the processing of an image failed, for example due to connection issues, then the image was retried until it could be processed.

The bytes reversal workload used 1, 2, 4, 8, and 16 active function instances in separate runs of the parallel variant. The single variant was limited to one and four function instances, as no parallel processing is required to respond to a single request and no performance gains can be expected from using additional instances. The number of requests for the parallel variant was set to 100, and was repeated ten times for each number of bytes. First, 1 B (2^0) was sent, then 2 B (2^1), etc, up to 16 MiB (2^{24}), for a total of 25,000 requests over 250 test runs. The actual number of requests is likely to be higher, as any failed requests

were retried until they succeeded. The single variant was repeated 1000 times for each number of bytes.

Finally, the resulting throughput rates were calculated based on the size of the input data, ignoring variations in the data size of the output as well as any failed requests. Furthermore, each benchmark was run until all of its requests were completed such that all tasks could be finished except for one (possibly the slowest task, on which the entire benchmark has to wait). This effect should be negligible, as the size of the individual tasks was relatively small.

Furthermore, the rates at which requests were sent out from the FD were limited in order not to overwhelm the platforms. For the image processing task, this was set as up to two active requests per function instance, while for the byte reversal workload it was set as up to four active requests per function. The rate was set as active requests instead of requests per second, as when using requests per second retries are not accounted for, and over time a backlog of failed requests may form. Limiting based on active requests avoids this, as the number of requests in transit is always known.

The experiments were conducted using only a single version for each of the applications used. Thus, it is not impossible that the versions under study might not be fully representative of the applications, as future versions could improve their performance or introduce new performance bottlenecks. This study thus needs to be understood as a point-in-time evaluation that is not fully representative of future versions of these applications; nonetheless, it delivers insight into which conceptual features are best able to improve application performance.

The concrete versions of the different applications and platforms used for our experiments are listed below:

- Ubuntu Server version 20.04.3
- OpenStack Queens release (Feb. 2018)
- Kubernetes version 1.20.8
- Docker version 20.10
- OpenFaaS version 0.21.1
- Nuclio version 1.6.18
- ImageMagick version 7.10-3 Q16
- OpenMP version 4.5

Instructions for reproducing this study, including all source code, can be found on the project's GitHub repository (https://github.com/Twalord/2022_Paper_Serverless_Performance_Evaluation accessed on 17 June 2022).

3. Results

3.1. Image Processing

The benchmark for the image processing workload measured the time required to process 1000 images in each variant and configuration. Analysis was carried out based on the number of images that were processed every second, calculated as $x = \frac{1000}{t}$, where x is the number of images processed per second and t is the time required to process all 1000 images. With 3 MiB per image on average, the data throughput was calculated as $x_b = x \cdot 3\,\text{MiB}$ where x_b is image data in MiB processed per second.

Figures 1 and 2 along with Table 1 show the results of our experiments. The figures show how many images per second OpenFaaS and Nuclio were able to process per second for various numbers of function instances as bar plots. The standard deviation of each bar is shown by the small marker on top of the bars. If the marker is not visible, this means that the standard deviation is too low to be shown in the figure. Table 1 notes the specific peak throughput rates for both variants of the workload, including the baseline computed on the FD node alone.

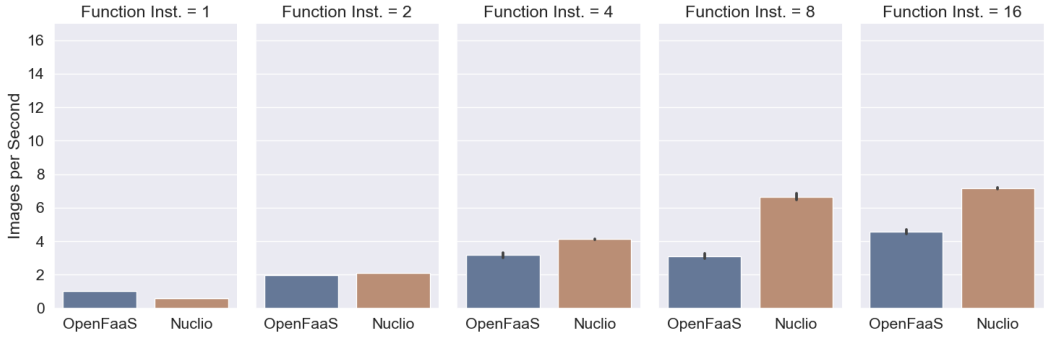

Figure 1. Image workload: OpenFaaS and Nuclio in comparison.

In an optimal environment, performance should scale linearly with the number of function instances. With the null-processing variant, the observed scalability is close to this expectation. However, with computation, the performance for Nuclio barely changes between 8 and 16 function instances. A detailed discussion of these results follows.

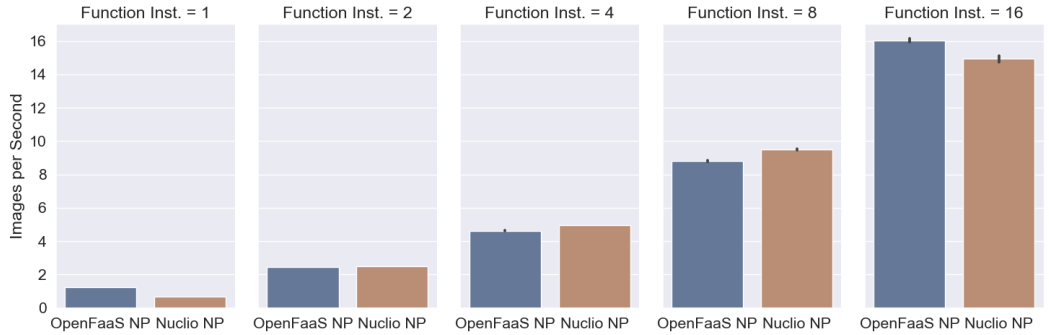

Figure 2. Image workload with null processing: OpenFaaS and Nuclio in comparison.

Table 1. Peak input processing rates for image processing workload in Images/s and MiB/s with 3 MiB per image on average.

Unit	Baseline	OpenFaaS	Nuclio	OpenFaaS NP	Nuclio NP
Images/s	12.58	4.58	7.17	16.04	14.93
MiB/s	37.74	13.74	21.51	48.12	44.79

From Figure 1, it can be observed that Nuclio clearly outperforms OpenFaaS for four or more function instances, with OpenFaaS peaking at 4.58 images per second compared to 7.17 images per second, as listed in Table 1. This means that Nuclio processes about 1.5 times more images per second compared to OpenFaaS. Meanwhile, the baseline sits at 12.58 images per second.

This difference in performance can most likely be attributed to Nuclio being better suited for computational tasks than OpenFaaS due to different implementations and features. One such feature is Nuclio's internal queue, which gathers requests sent to its HTTP endpoints and forwards them as soon as a worker is available (See Nuclio Documentation: https://nuclio.io/docs/latest/concepts/best-practices-and-common-pitfalls/ accessed on 27 May 2022). This reduces the idle time on the workers, as they do not need to wait for a new request to enter the system. While OpenFaaS has a similar queue, it only uses it for its asynchronous endpoints, which were not under test here.

Figure 1 shows that OpenFaaS and Nuclio reach their highest throughput rate when using sixteen function instances, that is, with one function instance per CPU core. However, the performance increase from eight function instances to sixteen is not as significant as one might expect, especially for Nuclio. This leads to the conclusion that each function instance can effectively support close to two function processes for this workload.

The NP variant of the workload shown in Figure 2 paints a somewhat different picture. Here, the throughput rates scale almost proportionally with the number of function instances, and OpenFaaS has a slight edge over Nuclio for sixteen function instances, with 16.04 images processed per second compared to 14.93. Here, the computational load seems to be so low as compared to the input size that the web servers in the function instances are too busy handling I/O for a single function process to be able to spawn additional function processes that better utilize the hardware.

Having to use additional function instances to handle a higher throughput is not desirable, as every additional function instance incurs additional overhead. Optimally, a single function instance would be able to fully utilize the resources of a host. This is currently not the case, as the function instances of both OpenFaaS and Nuclio handle I/O sequentially. In terms of achieved network throughput, the performance lags vastly behind our iPerf measurements of 700 MiB/s. Our analysis of network performance was carried out using the byte reversal workload.

3.2. Byte Reversal

For the byte reversal workload, we measured the time needed to complete 1 or 100 requests for the single and parallel variants of the workload, respectively. The requests carried from 1 B up to 16 MiB per request, with the Go variant only going up to 2 MiB as all requests using a larger request body failed. Furthermore, the parallel variants were tested using 1, 2, 4, 8, and 16 function instances, while the single variant was only tested for 1 and 4 function instances.

The analysis depends on the throughput in bytes per second calculated from the size of the requests and the time needed to successfully complete all requests of a test run. y bytes per second is calculated as $y = n \cdot \frac{b}{t}$, where n is the number of requests, b is bytes per request, and t is the time needed to process n requests of size b.

Figure 3 shows the results of the benchmarks as box plots, with OpenFaaS (Figure 3a,b), Nuclio (Figure 3c,d), and Nuclio Go (Figure 3e,f), respectively, in each row, single requests on the left side (Figure 3a,c,e), and parallel requests on the right side (Figure 3b,d,f). Each graph shows the throughput rate in bytes per second on the y-axis and the request size on the x-axis. As the request size was doubled after each test run, a logarithmic scale with base 2 is used on the x-axis.

Each graph shows the results for the number of function instances that reached the highest peak throughput rate. This is notable for the single variant with four function instances for OpenFaaS (Figure 3a) and Nuclio (Figure 3c) and only one function instance for Nuclio Go (Figure 3e), as well as for the parallel variants with four function instances for OpenFaaS (Figure 3b) and with eight function instances for Nuclio (Figure 3d) and Nuclio Go (Figure 3f). Furthermore, the concrete peak throughput rates in MiB/s are listed in Table 2 along with the number of function instances used.

Table 2. Peak average input processing rates in MiB/s for byte reversal workload, single and parallel variants.

Variant	OpenFaaS		Nuclio		Nuclio Go	
	Throughput	**Instances**	**Throughput**	**Instances**	**Throughput**	**Instances**
Single	7.47	4	9.64	4	12.71	1
Parallel	20.00	4	33.27	8	33.25	8

The graphs for the single request variant (Figure 3a,c,e) show a significant number of outliers, especially the Nuclio Go implementation, which is caused by the fact that every single box in each plot relies on 1000 data points. However, the outliers may offset the average throughput rate used to discuss the results. To ensure that a mean average-based discussion is valid, Table 3 shows the arithmetic mean (as Mean), geometric mean (as Gmean), and harmonic mean (as Hmean) for the single variant. The table shows that

the geometric mean and harmonic mean are all within 90% of the arithmetic mean, even for Nuclio Go. Therefore, the outliers do not significantly distort the results; thus, the remaining discussion uses the arithmetic mean.

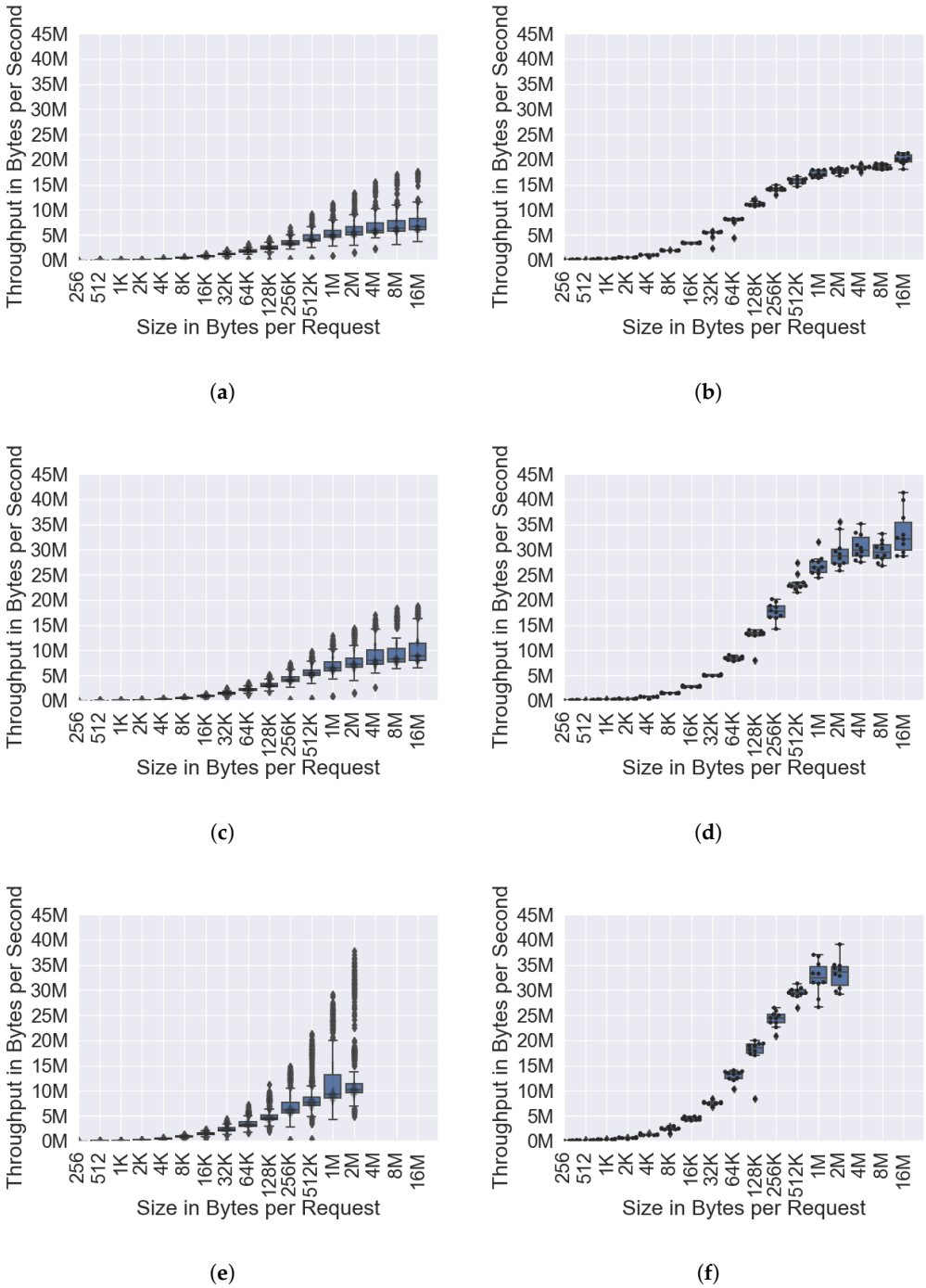

Figure 3. Throughput-focused workload: reversing input with OpenFaaS, Nuclio, and Nuclio Go. (**a**) OpenFaaS, single request, four function instances; (**b**) OpenFaaS, parallel, four function instances; (**c**) Nuclio, single request, four function instances; (**d**) Nuclio, parallel, eight function instances; (**e**) Nuclio Go, single request, one function instance; (**f**) Nuclio Go, parallel, eight function instances.

Table 3. Byte reversal single request: arithmetic mean, geometric mean and harmonic mean for throughput in MiB/s.

OpenFaaS			Nuclio			Nuclio Go		
Mean	**GMean**	**Hmean**	**Mean**	**Gmean**	**Hmean**	**Mean**	**Gmean**	**Hmean**
7.47	7.22	7.02	9.64	9.40	9.20	12.71	11.63	11.44

In Table 2, it can be seen that, for single requests, Nuclio processes about 29% more bytes per second than OpenFaaS, while Nuclio Go processes 32% more bytes per second than Nuclio and 59% more than OpenFaaS. Each of them reached their highest throughput rate for the largest number of bytes per request that they could handle, that is, 16 MiB for OpenFaaS and Nuclio and 2 MiB for Nuclio Go.

Single requests additionally favor low latency in response time, and the above observations align with the idea that Nuclio is faster at data processing than OpenFaaS. Furthermore, the theory that Nuclio Go is likely to outperform the other two implementations due to its zero copy approach is confirmed by these results. The fact that larger requests lead to better throughput rates can be explained through the nature of TCP connections. TCP needs time to scale up the transmission speed, and for larger requests that need to be send via multiple packages, it has more time to scale up and remain at the maximum transmission speed. It should be noted that Nuclio Go only transferred 2 MiB as its maximum transmission speed; thus, it can be assumed that its performance would further improve should it support larger request bodies. This problem is most likely due to a problem in the Nuclio Go SDK library.

The high number of outliers for single requests, especially for Nuclio Go, are most likely due to the large number of data points used for these benchmarks. Nevertheless, this does not explain how certain outliers for Nuclio Go are up to four times faster than the average. The cause of this is most likely that 2 MiB happens to fit very well into the L2 CPU cache of the AMD EPYC processors of the same size. Thus, in certain cases the throughput can be very fast when a request comes in, is stored in L2 cache, is not copied thanks to the native implementation, and then is handled and sent back while barely having to touch the main memory. This does not always occur, as other processes might interrupt the function execution such that the input data are partially evicted from the L2 cache. This explanation could not be confirmed, however, as actual cache eviction was not measured.

For the parallel variant, it can be seen in Table 2 that Nuclio and Nuclio Go outperform OpenFaaS by about 60%. Nuclio and Nuclio Go are almost identical; however, as Nuclio Go only used 2 MiB per request and Nuclio used up to 16 MiB, it can be assumed that Nuclio Go would, similar to Nuclio and OpenFaaS, gain performance from using larger requests.

It is notable that the graphs for the parallel variant resemble the "S"-shape of logisitic or Sigmoid curves. The reason for this is the same as the reason for the single variant's better transmission speed for larger request bodies: the time spent at the optimal transmission speed is improved. However, using very large request bodies has the disadvantage of requiring more data to be re-transmitted should a request fail.

For the image processing workload, it was observed in both variants that the throughput is optimal when using sixteen function instances. Here, the throughput peaks were for four or eight function instances instead of sixteen. The NP variant of the image processing workload scales the best with the number of function instances, which is due to the one-second sleep that this workload executes for every image, meaning that it can handle the I/O for other requests during the sleep time. For the byte reversal workload, each request can almost immediately be sent back when it arrives. This most likely causes a different bottleneck, probably with the routing mechanism of the platform. Using Nuclio's internal queue, it is likely that this enables Nuclio to utilize eight instead of four function instances for its peak throughput.

The throughput rates for the fastest variants and implementations of the two workloads were capped at 44.79 MiB/s for Nuclio NP in the image processing workload and

33.25 MiB/s for the parallel Nuclio Go byte reversal workload. Both of these are far below the 700 MiB/s theoretical throughput speed. While reaching close to 700 MiB/s would be unrealistic, it would have been far more promising if any of the workloads reached at least 100 MiB/s. It is notable that the parallel Nuclio Go byte reversal workload had a 40% lower throughput rate than the Nuclio NP variant in the image processing workload. This could be attributed to the byte reversal operation being more computationally expensive than anticipated compared to the load image operation, or that the one-second sleep enables the system to better parallelize the workload.

4. Discussion

4.1. Implications

The performance of the serverless platforms seems promising in several aspects; however, is not yet at the level of a valid alternative for classic HPC systems. In order for serverless platforms to be more efficient, they must reduce the overhead for function invocations and improve parallelization of function instances. Using native integration of user functions is a good approach, which Nuclio already supports for Go and a few others templates. Allowing for native integration requires the developers of serverless platforms to provide a form of SDK library for use in developing user functions.

Having developers provide SDK libraries leads to another issue, that of vendor lock-in. If every serverless platform builds its own SDK for every language it supports, it will be difficult to transfer function implementations between platforms, as the code would need to be adjusted for the SDK of the new platform. This could be avoided by having a single open-source project for developing SDK libraries for various languages, meaning that user functions would only need to work with this standard SDK project and could be deployed to any platform that supports said project's SDKs.

Parallelization of I/O is problematic as well. It would be optimal if only a single function instance was necessary for each node, as the function instance would then be able to handle multiple parallel function invocations. This is already the case for OpenFaaS and Nuclio function instances; however, the I/O part of the function instances that is responsible for starting new function invocations and forwarding input data to them is not able to keep up unless the tasks require time to process without needing very much CPU, such as the NP variant of the image processing workload. By having function instances handle I/O in parallel, it becomes unnecessary to deploy more than one function instance from the same function on a node, removing the overhead of running multiple instances of the same container.

Requiring only a single function instance for a function per node brings further advantages. The issue of slow function starts as described by Jonas et al. [2] to handle traffic spikes would be resolved, as no more than one function instance need be deployed to utilize a node's resources. This also means that resources that must be initialized (e.g., libraries, database connections) can be shared among all function invocations on the same node.

Van Eyk et al. [16] and Schleier-Smith et al. [1] describe the prediction of serverless function performance as a major issue. Serverless users are unaware of what hardware resources are available, and platform providers do not know about the resource requirements of user functions. Having only up to one function instance per node would potentially enable the platform to better control the dynamic resource allocation for function invocations while minimizing over-subscription of CPU cores and memory. However, this requires users to state how many CPU cores and how much memory a single function invocation may require. Depending on the nature of a function, this might require dynamic adjustment, for example, where a function's memory consumption scales with its input size, such as for an image processing function. In such a case, the serverless platform would require a formula from the user as to how much memory to reserve depending on the input size. Instead of performing round-robin scheduling for all available function instances, the scheduler that distributes requests to function instances would need to consider re-

source requirements and available resources. This would represent a step towards a "*smart scheduler*" as described by van Eyk et al. [16].

4.2. Error Analysis

According to Scheuner et al. [4], while using self-hosted serverless platforms provides full control over the platforms for experiments, it requires the experimenters to handle setup, configuration, and maintenance of the platforms. This study is no exception, meaning that random and systematic error need to be accounted for.

Random error can be assessed by inspecting the spread of the test results. In this study, each configuration was tested at least ten times, and the analysis always included the standard deviation. All variability in the benchmark results is either acceptable, or at least explainable. Due to the nature of random error, there remains a non-zero chance that all results are collectively offset to an unknown degree.

The aspect of systematic error is more difficult to assess. The affects of systematic error can be indistinguishable from actual test results if they do not lead to any crashes or errors in program execution. To detect systematic error, a baseline was established for both workloads in order to determine what results would be realistic. Nevertheless, it is possible that one or more sources of systematic error influenced the results such that the serverless platforms appear slower than they actually are. However, as the platforms were left mostly in their default configurations, the benchmarks should be representative of the performance a user of these platforms can expect. Even if the results are offset by systematic errors, such as misconfiguration, the arguments made above remain valid.

4.3. Future Work

Although this study only considered the serverless platforms OpenFaaS and Nuclio, there are many other mature open-source serverless platforms that could be benchmarked in comparison. By gathering a larger sample size of studied platforms, it will become possible to form appropriate statements about the state of open-source serverless as a whole.

Furthermore, the sample workloads should be expanded in order to better stress the features and problem areas of serverless platforms. In this regard, it is of interest to look into other approaches to benchmarking serverless, such as [17]. Along with improving the workloads, the measuring process should be improved to include tools allowing for profiling of function invocations.

Another aspect of HPC is the usage of accelerators; for a move towards HPC, these need to be supported in serverless environments. While there are studies that have investigated support for the GPU acceleration of serverless functions, such as [18], no studies have investigated the overall state of GPU support for open-source serverless platforms. This is interesting, as NVIDIA GPUs support hardware-level partitioning through their MIG feature (https://github.com/nvidia/mig-parted accessed on 27 May 2022), which aligns well with the security requirements for serverless.

5. Conclusions

The use of open-source serverless platforms for HPC workloads could be a significant gain in terms of finding more usable ways of interacting with HPC clusters. Furthermore, it could enable clusters to achieve better hardware utilization through dynamic sharing of resources compared to the reservation-based systems commonly used in HPC, such as Slurm. This study investigated the performance of OpenFaaS and Nuclio using two data processing-focused sample workloads in multiple variants to conduct benchmarks on a test setup.

The benchmarks revealed that Nuclio is able to outperform OpenFaaS by a factor of about 1.5 in terms of data throughput. Nuclio's performance in a compute task was adequate compared to a baseline, although lower than expected for a throughput-focused task. Our experiments confirmed the theory that natively integrating user functions with function instances can improve performance by avoiding extra copy operations.

However, there are limitations that need to be overcome before the existing open-source serverless platforms can represent a valid option to deliver reasonable performance for HPC usage. The community needs benchmarking efforts such as ours to improve their systems in the quest to obtain bare-metal performance. Due to problems with Fission, which were submitted as an issue and later fixed by the community, this study could only investigate two prominent open-source serverless platforms on Kubernetes; a more comprehensive study is necessary before any absolute statements can be made on the state of open-source serverless. Hopefully, future advancements in the field of serverless computing will either see existing open-source platforms adopt better support for performance focused use cases, or see the emergence of new serverless platforms with dedicated support for HPC.

Author Contributions: Conceptualization, J.D. and J.M.K.; methodology, J.D., J.M.K. and P.K.; software, J.D.; validation, J.D.; formal analysis, J.D., J.M.K. and P.K.; investigation, J.D.; Resources, J.D. and P.K.; data curation, J.D.; writing—original draft preparation, J.D.; writing—review and editing, J.D. and J.M.K.; visualization, J.D.; supervision, J.M.K. and P.K.; project administration, J.D. and J.M.K.; funding acquisition, P.K. All authors have read and agreed to the published version of the manuscript

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The computing systems for setting up and running the tests in this study were kindly made available by the GWDG (https://gwdg.de accessed on 27 May 2022).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

API	Application Programming Interface
AWS	Amazon Web Services
FaaS	Function-as-a-Service
FD	Function Director
HPC	High-Performance Computing
MDPI	Multidisciplinary Digital Publishing Institute
MPI	Message Passing Interface
NP	Null Processing
RDMA	Remote Direct Memory Access
SDK	Software Development Kit
VM	Virtual Machine

Appendix A

The full source code with instruction for reproducing this study can be found here: https://github.com/Twalord/2022_Paper_Serverless_Performance_Evaluation (accessed on 17 June 2022).

Appendix A.1

Listing A1. Image processing serverless function code. The function first receives the input data via stdin (Line 10-13) and then loads it into a `Blob` (Line 15-17), which is a data type defined by ImageMagick for holding binary image data. Afterwards, it creates an `Image` object from the `Blob` (Line 18), which is another ImageMagick data type. Using this `Image` object, a set of image operations is applied (Line 20-28) that require roughly 1.3 seconds of compute time on the target CPU. Afterwards, it extracts the now modified `Blob` from the `Image` object (Line 30-32) and writes it back as binary data via stdout (Line 33-37). The NP variant uses the same code, except instead of performing image transformations (Line 20-28), it calls `sleep(1)`.

```
1  #include <Magick++.h>
2  #include <iostream>
3
4  using namespace Magick;
5
6  int main()
7  {
8      try{
9          // Do not skip whitespaces while reading
10         std::cin >> std::noskipws;
11         std::istream_iterator<char> it(std::cin);
12         std::istream_iterator<char> end;
13         std::string input(it, end);
14         // Convert string to unsigned char* to const void* so Blob can use it
15         auto input_c = reinterpret_cast<const unsigned char*>(input.c_str());
16         const void* input_cv = static_cast<const void*>(input_c);
17         Blob blob(input_cv, input.size());
18         Image image(blob);
19         // Perform image transformations
20         image.blur(1,0.5);
21         image.addNoise(MultiplicativeGaussianNoise, 1);
22         image.addNoise(LaplacianNoise, 1);
23         image.enhance();
24         image.negate();
25         image.normalize();
26         image.reduceNoise();
27         image.swirl(37);
28         image.wave();
29         // Convert image back to blob
30         Blob out;
31         image.magick("PNG");
32         image.write(&out);
33         const void* out_cv = out.data();
34         size_t out_size = out.length();
35         // And write to stdout
36         std::string out_str(static_cast<const char*>(out_cv), out_size);
37         std::cout << out_str;
38         return 0;
39     }
40     catch( Exception &error_ )
41     {
42         std::cout << "Caught exception: " << error_.what() << std::endl;
43         return 1;
44     }
45 }
```

Appendix A.2

Listing A2. Byte reversal workload—Serverless C++ function code. The C++ implementation of the byte reversal serverless function uses the same stream mechanisms for reading and writing to and from stdin and sdout as the image processing implementation shown in Appendix A.1. Reversal of bytes is performed using the reverse function from the C++ standard library (Line 13).

```
#include <iostream>
#include <bits/stdc++.h>

int main(int argc, char **argv)
{
  try {
    // Do not skip whitespaces while reading
    std::cin >> std::noskipws;
    std::istream_iterator<char> it(std::cin);
    std::istream_iterator<char> end;
    std::string input(it, end);
    // Reverse the input and send it back
    reverse(input.begin(), input.end());
    std::cout << input;
    }
    catch( Exception &error_ )
    {
        std::cout << "Caught exception: " << error_.what() << std::endl;
        return 1;
    }
  return 0;
}
```

Appendix A.3

Listing A3. Byte reversal serverless Go function code. The Go implementation of the byte reversal workload does not need to handle streams and can instead directly handle object references passed to it as function parameters. The user function implements an interface defined by the Nuclio Go SDK (Line 7). As Go has no reverse function in its standard library, this is instead implemented as a loop (Line 10-12).

```
package main

import (
    "github.com/nuclio/nuclio-sdk-go"
)

func Handler(context *nuclio.Context,event nuclio.Event)(interface{},error){
    b := event.GetBody()
    // Reverse bytes of input
    for i,j := 0, len(b) - 1; i<j; i,j = i+1, j-1 {
        b[i],b[j] = b[j], b[i]
    }
    return nuclio.Response{
        StatusCode:  200,
        ContentType: "application/text",
        Body:        b,
    }, nil
}
```

References

1. Schleier-Smith, J.; Sreekanti, V.; Khandelwal, A.; Carreira, J.; Yadwadkar, N.J.; Popa, R.A.; Gonzalez, J.E.; Stoica, I.; Patterson, D.A. What Serverless Computing Is and Should Become: The next Phase of Cloud Computing. *Commun. ACM* **2021**, *64*, 76–84.
2. Jonas, E.; Schleier-Smith, J.; Sreekanti, V.; Tsai, C.C.; Khandelwal, A.; Pu, Q.; Shankar, V.; Carreira, J.; Krauth, K.; Yadwadkar, N.; et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv* **2019**, arXiv:cs/1902.03383.

3. Spillner, J.; Mateos, C.; Monge, D.A. FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC. In Proceedings of the High Performance Computing, Dresden, Germany, 11–12 September 2017; Mocskos, E., Nesmachnow, S., Eds.; Communications in Computer and Information Science; Springer International Publishing: Cham, Switzerland, 2018; pp. 154–168. [CrossRef]
4. Scheuner, J.; Leitner, P. Function-as-a-Service Performance Evaluation: A Multivocal Literature Review. *J. Syst. Softw.* **2020**, *170*, 110708. [CrossRef]
5. Van Eyk, E.; Scheuner, J.; Eismann, S.; Abad, C.L.; Iosup, A. Beyond Microbenchmarks: The SPEC-RG Vision for a Comprehensive Serverless Benchmark. In Proceedings of the Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE '20), Edmonton, AB, Canada, 25–30 April 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 26–31. [CrossRef]
6. Beltre, A.M.; Saha, P.; Govindaraju, M.; Younge, A.; Grant, R.E. Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms. In Proceedings of the 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), Denver, CO, USA, 18 November 2019; pp. 11–20. [CrossRef]
7. Kuhlenkamp, J.; Werner, S. Benchmarking FaaS Platforms: Call for Community Participation. In Proceedings of the 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), Zurich, Switzerland, 17–20 December 2018; pp. 189–194. [CrossRef]
8. Van Eyk, E.; Grohmann, J.; Eismann, S.; Bauer, A.; Versluis, L.; Toader, L.; Schmitt, N.; Herbst, N.; Abad, C.L.; Iosup, A. The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms. *IEEE Internet Comput.* **2019**, *23*, 7–18. [CrossRef]
9. Eismann, S.; Scheuner, J.; van Eyk, E.; Schwinger, M.; Grohmann, J.; Herbst, N.; Abad, C.L.; Iosup, A. Serverless Applications: Why, When, and How? *IEEE Softw.* **2021**, *38*, 32–39. [CrossRef]
10. Jain, P.; Munjal, Y.; Gera, J.; Gupta, P. Performance Analysis of Various Server Hosting Techniques. *Procedia Comput. Sci.* **2020**, *173*, 70–77. [CrossRef]
11. Malla, S.; Christensen, K. HPC in the Cloud: Performance Comparison of Function as a Service (FaaS) vs Infrastructure as a Service (IaaS). *Internet Technol. Lett.* **2020**, *3*, e137. [CrossRef]
12. Zhou, N.; Georgiou, Y.; Pospieszny, M.; Zhong, L.; Zhou, H.; Niethammer, C.; Pejak, B.; Marko, O.; Hoppe, D. Container Orchestration on HPC Systems through Kubernetes. *J. Cloud Comput.* **2021**, *10*, 16. [CrossRef]
13. Decker, J. *The Potential of Serverless Kubernetes-Based FaaS Platforms for Scientific Computing Workloads*; Göttingen Research Online/Data 2022. Available online: https://data.goettingen-research-online.de/dataset.xhtml?persistentId=doi:10.25625/6GSJSE (accessed on 17 June 2022).
14. Cheney, D. *Cgo | The Acme of Foolishness*; 2016. Available online: https://dave.cheney.net/2016/01/18/cgo-is-not-go (accessed on 17 June 2022).
15. Asuni, N.; Giachetti, A. TESTIMAGES: A Large Data Archive For Display and Algorithm Testing. *J. Graph. Tools* **2013**, *17*, 113–125. [CrossRef]
16. Van Eyk, E.; Iosup, A.; Abad, C.L.; Grohmann, J.; Eismann, S. A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures. In Proceedings of the Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18), Berlin, Germany, 9–13 April 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 21–24. [CrossRef]
17. Copik, M.; Kwasniewski, G.; Besta, M.; Podstawski, M.; Hoefler, T. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In Proceedings of the 22nd International Middleware Conference (Middleware '21), Québec, QC, Canada, 6–10 December 2021; Association for Computing Machinery: New York, NY, USA, 2021; pp. 64–78. [CrossRef]
18. Satzke, K.; Akkus, I.E.; Chen, R.; Rimac, I.; Stein, M.; Beck, A.; Aditya, P.; Vanga, M.; Hilt, V. Efficient GPU Sharing for Serverless Workflows. In Proceedings of the 1st Workshop on High Performance Serverless Computing (HiPS '21), Stockholm, Sweden, 23–26 June 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 17–24. [CrossRef]