Serverless Computing in the Edge Continuum Group 2

David Freina
d.freina@student.vu.nl
Distributed Systems (X_400130)
Faculty of Science, Vrije Universiteit Amsterdam

Maciej Kozub
m.p.kozub@student.vu.nl
Distributed Systems (X_400130)
Faculty of Science, Vrije Universiteit Amsterdam

Paulo Liedtke
p.liedtke@student.vu.nl
Distributed Systems (X_400130)
Faculty of Science, Vrije Universiteit Amsterdam

German Savchenko
g.savchenko@student.vu.nl
Distributed Systems (X_400130)
Faculty of Science, Vrije Universiteit Amsterdam

Jonas Wagner
j.a.wagner@student.vu.nl
Distributed Systems (X_400130)
Faculty of Science, Vrije Universiteit Amsterdam

Matthijs Jansen
m.s.jansen@vu.nl
Distributed Systems (X_400130)
Faculty of Science, Vrije Universiteit Amsterdam

Linus Wagner
l3.wagner@student.vu.nl
Distributed Systems (X_400130)
Faculty of Science, Vrije Universiteit Amsterdam

Alexandru Iosup
a.iosup@atlarge-research.com
Distributed Systems (X_400130)
Faculty of Science, Vrije Universiteit Amsterdam

***Abstract*—Together with the advent of large-scale distributed (eco)systems, cloud, edge, and serverless computing paradigms are quickly becoming dominant in the industry. Therefore there is a growing need for monitoring applications that can help to understand how complex distributed and decentralized systems perform. For this purpose, we propose a solution, a benchmarking tool developed for the computing continuum, and a series of experiments reasoning about the strengths and the weaknesses of serverless computing in the cloud and in the edge. Our solution is fully automated and provides an easy way to modify the experiments. During our work, we found out that edge computing benefits in low-compute intense tasks, due to lower communication overhead. However, as soon as the problem size grows, the cloud largely outperforms the edge.**

I. Introduction

Cloud computing represents the most influential paradigm shift in computing for some time already. It allows developers and software companies to focus on developing applications and doing research while all infrastructure management can be outsourced to the cloud provider. Edge computing, on the other hand, represents another approach, where computation is moved from large-scale centralized computing centers to a larger number of decentralized, heterogeneous, computing devices with limited resources. Cloud and edge computing can coexist in the so-called computing continuum. Development and seamless deployment for both of the infrastructures can be a challenge since these systems exhibit different properties.

Monitoring such systems is crucial and plays a critical role, for example, in real-time data-driven decisions and automation of workload placement [1]. Collecting and analyzing run time data of systems can help to ensure that computing resources are utilized optimally and that the performance requirements of the applications and services are met. By monitoring key metrics such as CPU utilization, memory usage, or network traffic, organizations can address issues from a performance perspective appropriately. Monitoring can further help to minimize maintenance costs by revealing optimization opportunities. Thus, it is essential today to further understand and analyze the behavior of the computing continuum under different workloads. Currently, there is a lack of such a universal tooling that allows for high-quality, real-time monitoring of these systems.

This paper contributes to developing a benchmark method for the computing continuum. Moreover, a monitoring tool is created that collects and displays execution-time data in real time. For pushing the cloud and the edge to its limits three different serverless functions were developed – aimed to stress different resources on the nodes. The objective is to understand how the computing continuum performs under different kinds of workloads. The examination was done in terms of end-to-end latency, system load, and total user time.

The paper goes through the background of the created tool, the requirements, and additional features (§ II) that had to be met. Subsequently, the system design is introduced with the individual components, the choice of frameworks, and how they cooperate to run the experiments (§ III). The experimental setup and results are then presented and documented (§ IV). Next discussion of the findings is presented, followed by an analysis of the different workloads and design choices (§ V). Finally, the paper concludes with the main findings and the suggestion for possible future work based on the results (§ VI).

II. Background on the project

Our goal is to develop a monitoring platform that extracts the logs from the running serverless functions in the compute continuum (cloud and edge environment). The project has as an objective the development of three serverless functions that simulate different workloads on the system, and the monitoring of their execution on a cluster provided by the *@Large Research*[2] group at VU Amsterdam. Furthermore, based on the results, we reason about the benefits and differences of executing various workloads in the aforementioned environments.

In addition to these standard requirements, we introduce extra features to facilitate the whole process of deploying, running, and benchmarking the workloads.

1) Automation: The application focuses on making the setup and execution process as seamless as possible, hence allowing the users to fully focus on designing new experiments without losing time on manually re-configuring all the necessary components for every run. The setup and experimental process are fully automated for both environments, cloud, and edge. It was also necessary to automate the components focusing on logs aggregation and visualization. The monitoring side is a key contribution to this project, and it helps the developer (as much as the reader) who wants to reproduce the experiments.

2) Variation of Workloads: The developed workloads are run in various setups. We execute them with different problem sizes in a sequential and parallel manner to see how the system handles these invocations and reacts to potential overloads. An overview of the workloads is available in Table I.

3) Open source: the code base of this project and this report are publicly available on GitHub[1].

III. System Design

In the following chapter, we present used technologies and how we combine them to run and measure experiments. The whole system was developed and tested on node four of the cluster mentioned in § II. This node has the following system specifications:

- CPU: Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
- RAM: 256 GB
- Bandwidth: 1 Gbit/s

[1]https://github.com/davidfreina/DistribSys-ServerlessEdge-Lab

A. Technologies

1) continuum: This framework[4] enables our system to automate the creation and simulation of cloud and edge environments in virtual machines. The user has to provide configuration files that set different parameters for the environment. Our system uses different configurations for the cloud and the edge environment. While the edge configuration supports low latencies (5ms) and low computational power (CPU quota of 0.5) over ten nodes with two cores each, the cloud configuration sets higher latencies (50ms) and a CPU quota of one over three nodes with six cores each.

2) Kubernetes: This open-source solution[5] orchestrates multiple containers within an environment. Kubernetes takes care of the availability of the container as well as the scaling and managing of deployments. This way Kubernetes, basically, provides the cloud/edge infrastructure in which the workloads are deployed and called on demand.

3) OpenFaaS: The open source function-as-a-service provider[6] gets automatically deployed by continuum and takes care of setting up a Kubernetes cluster according to the properties set in the configurations (s. subsubsection III-A1). The workloads (s. subsection IV-A) are available as images on Docker Hub [7] from where they are pulled and deployed on the Kubernetes cluster. Furthermore, it enables the user to manage the environment by providing an extensive command line tool. Depending on the system load OpenFaaS offers an auto-scaling subsection III-D option that instructs Kubernetes to adapt the available resources.

4) Prometheus: After everything is up and running we use this tool[8] to gather the metrics of the deployments on the cluster. Prometheus delivers relevant information for analyzing the workloads.

5) kube-prometheus: To combine Kubernetes and Prometheus in an easy and user-friendly way we use kube-prometheus[9]. It monitors the whole cluster and offers further functionalities like alerting rules and dashboards.

6) Loki: Each Kubernetes deployment manages zero to many container logging events. If a workload is finished and the cluster scales down, the logs are no longer available. To aggregate and persist these logs beyond the life of the container we use Loki[10]. Additionally, Loki provides an option to query relevant logs of the whole infrastructure.

7) Grafana: Finally, we want to monitor processes, workloads, logs, and benchmarks in one clear and highly adaptable user interface. Grafana[11] allows us to configure relevant dashboards to show live what is going on in the system. We provide screenshots of these dashboards in the appendix (s. subsection C).

8) hey: To generate specific workloads and simulate a lot of concurrent HTTP requests to the deployed functions we used hey[12]. This way we are able to gather the necessary data to generate in-depth visualizations.

9) curl: To generate some of the workloads we use *curl* to perform HTTP requests to the deployed functions.

10) multiprocessing in Python: For generating most of the workload, thus for most of the invocations of the serverless

Application name	Description	Used parameters	Aim
Fibonacci	Function calculates first *n* (where the *n* is passed as argument) values from the Fibonacci sequence.	Values used for different experiments: $n \in [5; 37], n \in [5; 45], n = 5$.	Mainly aimed to stress the CPU of the worker node.
Matmul	Function executes matrix multiplication of two matrices of size NxN filled with integers from the range [0; 10].	Values used for different experiments: $N \in [50; 450], N = 5$.	Functions aims to stress the CPU and the memory usage on the worker node.
File-upload	Function receives data (text, binary files, or other bytes that can be encoded as ASCII or UTF-8 character string), performs Base64 encoding, and returns 10 times the obtained encoded string.	We used 40 randomly selected JPEG images fetched from ESA[3]. ranging from 5KB to 10MB in size, as well as, 1.5KB PNG image.	This function aims to stress the bandwidth and latency of the connection to worker nodes.

TABLE I
OVERVIEW OF APPLICATIONS (SERVERLESS FUNCTIONS) USED IN EXPERIMENTS

function we use Python script with multiprocessing. This way we can simulate real-world-like load from end-users. Moreover, thanks to Python we can easily generate a load with a random or specific input for diversification.

11) pyinfra: The pyinfra is a powerful tool that allows for infrastructure automation at a massive scale. It is used for remote command execution, services deployment, services configuration, pulling and deploying docker images, experiments execution, and more. We opt for pyinfra instead of, for instance, *Ansible* because it is Python based and can be simply installed as *pip* package.

B. Experiments setup & execution

We have developed the proposed platform in such a way, that all experiments setup and execution are done automatically by running the prepared pyinfra[13] configurations. We decided to do this automation due to our own personal experience with projects in this domain which often require a lot of configuring. Furthermore, we are hoping to increase the reproducibility of our experiments by providing all the required configurations and deployments in an automated manner. There are two important files, *inventory.py* file which contains user-specific and Continuum environment-specific data (e.g. username, ssh credentials, or cloud-controller IP address). The second file called *deployment.py* contains all the steps that are executed in order to set the experiments environment and execute the designed tests. For instance, that includes installing apt packages, pulling needed repositories and images, opening port-forwarding, and finally deploying serverless functions with appropriate configuration, downloading test data, and executing all of the experiments. Furthermore, pyinfra is also responsible for setting up the logging and monitoring mentioned below.

C. Logging and Monitoring

We have set up an automated logging/monitoring platform to help measure the different metrics we need for our experiments. The project is structured as follows.

Our functions are deployed with *OpenFaaS*[6] which is installed in a *Kubernetes*[5] cluster deployed by *continuum*[4]. By deploying a ready-made monitoring platform called *kube-prometheus*[9] with our own configuration to the Kubernetes cluster we are able to collect various metrics from the functions deployed in OpenFaaS. Figure 1 gives an overview of the project architecture.

The collected metrics are stored in a *Prometheus*[8] database and include but are not limited to:

- CPU Usage per function and workload
- Memory Usage per function and workload
- Network bandwidth per workload

Additionally, we deploy a log aggregator called *Grafana Loki*[10] and configure OpenFaaS to persist its logs to said, log aggregator. Due to that, we are able to get accurate measurements for the function execution time from OpenFaaS.

Both of our data sources (Prometheus and Grafana Loki) are added to *Grafana*[11]. Grafana is used to visualize the measurements with different database queries and chart types. We have created two customized dashboards that show the most important metrics for our cluster. The first dashboard gives an overview of the resource usage of the executed function by using the aforementioned metrics stored in the Prometheus database while the second dashboard provides an insight into the function execution times by using the data from Grafana Loki.

D. Auto-scaling

By default, OpenFaaS deploy one instance of each deployed serverless function in the Kubernetes cluster. To really make use of all available resources on each node, as well as on the multi-node infrastructure, scaling needs to be configured. We found two ways of making functions scalable: one, using the built-in Kubernetes auto-scaling feature, and the other being the OpenFaas auto-scaling. During the development of the platform, we discovered that those two methods do not cooperate well together, and after some experiments, we decided to use OpenFaaS-provided scaling. This solution seems to be using the available nodes and their resources in a more efficient way. Moreover, the deployment of new instances on different nodes, happened faster while using OpenFaaS scaling. We decided to scale the pods based on the *capacity* factor - the number of concurrent requests each instance can handle. We also set the maximum number of function replicas to ten and the scaling factor to 50% - meaning that scaling happens firstly to six replicas and then to ten in case of such demand. This way we could fully utilize available resources

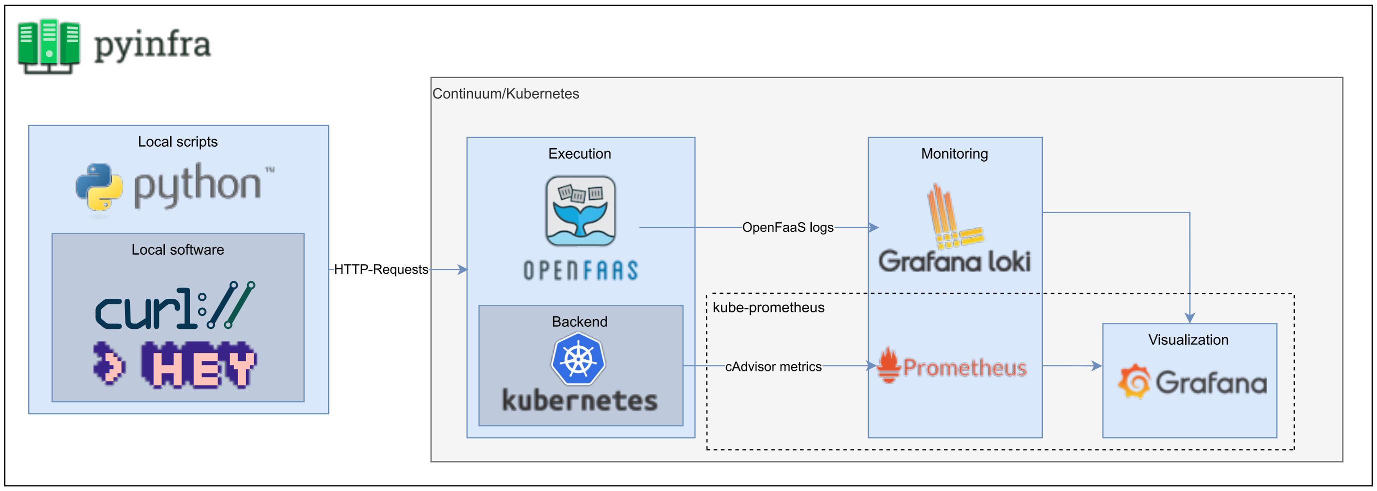


Fig. 1. Project architecture

in our edge configuration (ten nodes). We did not investigate the auto-scaling process more due to the limited amount of time for the project.

IV. Experimental Results

We conduct two different experiments in order to compare the cloud and edge environments as well as homogeneous and mixed workloads. Our experiments show the different properties of Edge and Cloud computing. Furthermore, our metrics indicate when the user is benefiting from using the Edge or the Cloud.

A. Approach to the Experiments

1) Cloud vs. Edge: In order to compare the cloud environment with the edge we are going to execute the same set of functions on both deployments. We are using an implementation of the Fibonacci algorithm to simulate low and high CPU-intensive workloads by changing which Fibonacci number we are going to calculate. By doing this we should be able to clearly show the benefits of the edge in low CPU-intensive workloads as well as the advantages that occur in the cloud for high CPU-intensive workloads. For comparing the transport overhead of the cloud and edge environments we are using file upload API. The function with API implementation receives the file or data and returns ten times more data back to the user. Therefore we aim to stress the connection between the computing node and the user.

2) Homogeneous vs. Mixed workloads: Apart from simply comparing the cloud and edge environment, we are also interested in seeing how each of the environments is able to handle different kinds of workloads. Therefore, we deploy three different functions which help us to simulate homogeneous and mixed workloads. The functions consist of the aforementioned Fibonacci algorithm, file upload API, and matrix multiplication algorithm. If we run the functions sequentially with a fixed set of input parameters we should be able to observe the total duration for all three workloads as well as the duration per workload. Those measurements should then be compared to a parallel run where we run all the functions at the same time with the same set of input parameters.

With all the measurements we gather from those executions we should be able to calculate how the execution time of the functions differs between homogeneous and mixed workloads. Thus exploring how cloud and edge environments behave in such conditions.

3) Experimenting with constant problem sizes: We conduct another experiment that simulates a high amount of simultaneous requests to the server. To achieve that we used hey as an HTTP request runner and constant problem sizes for our functions. For matrix multiplication, we chose the matrix of size: $5x5$, for Fibonacci calculation we chose the fifth Fibonacci number, and for file API we used a picture of size 1.5 KB. hey is used to invoke each function 1000 times by 50 workers. The above functions' invocation commands are executed five times in a sequential and parallel manner. An example output is provided in the appendix (subsection B).

All functions used in the experiments together with used parameters were summarized in the Table I. Table II on the other hand contains an overview of all conducted experiments.

B. Results

In the following subsections, we are going to explain the results of the experiments we have conducted. We are analyzing the user time in correlation with the CPU usage for the Fibonacci function as well as the user time variance in a mixed workload environment compared to a homogeneous execution of all three functions.

Similar to the detailed Fibonacci analysis we wanted to provide insight into how memory usage and user time correlate with each other by using our matrix multiplication implementation. However, due to limitations unknown to us, the function was unable to be successfully executed for matrices bigger than 500x500. Testing the function locally revealed no

Experiment	Workload type	Environment type	Functions	Metrics
Cloud vs. Edge	Homogeneous	Cloud, Edge	Fibonacci, File-upload	User Time, CPU usage, Memory usage, Rx/Tx Bandwidth
Homogeneous vs. Mixed workload	Homogeneous, Heterogeneous	Cloud, Edge	Fibonacci, File-upload, Matmul	User Time, Variance of User Time, CPU usage, Memory usage, Rx/Tx Bandwidth
Constant problem size	Heterogeneous	Cloud, Edge	Fibonacci, File-upload, Matmul	User Time, Variance of User Time, CPU usage, Memory usage, Rx/Tx Bandwidth

TABLE II
OVERVIEW OF THE CONDUCTED EXPERIMENTS

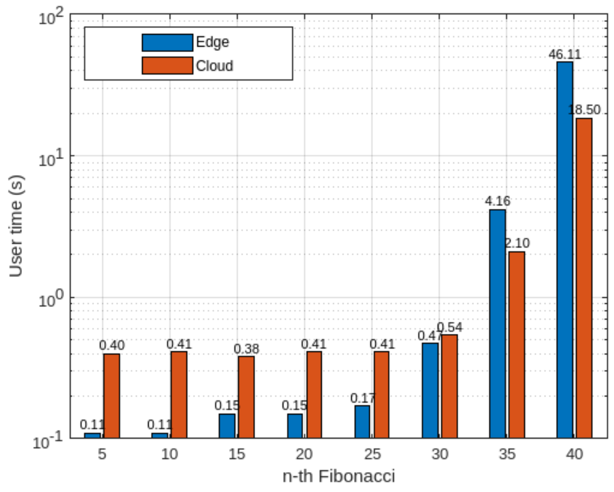


Fig. 2. User Time Fibonacci (Edge vs. Cloud) with logarithmic user time scale in seconds and input n for the Fibonacci function

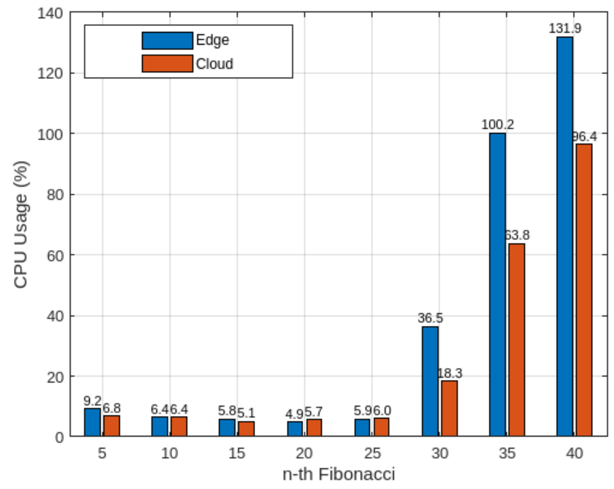


Fig. 3. CPU Usage Fibonacci (Edge vs. Cloud) with CPU usage in percent (100% being one core fully utilized) and input n for the Fibonacci function

errors in the function code and OpenFaaS does not provide any errors.

1) User Time Fibonacci: The system provides us with different metrics and logs. From this gathered data, we deduct Figure 2 which shows the user time for the calculation of the nth Fibonacci number. The experiment runs on the Edge infrastructure and on a Cloud cluster. While we observe almost constant user time on the cloud environment up to a problem size of $n = 30$ the user time increases slightly on the edge environment. Up to this point, we benefit more from the small latency to the edge nodes. The cloud is bottle-necked by its higher latency for these values of n. As soon as we execute the Fibonacci function for $n \approx 32$ the cloud is returning faster to the user than the edge. For $n \gtrapprox 33$ the edge nodes are bottle-necked by their fairly low computational power while cloud nodes are able to complete faster due to more computational resources available on the nodes.
The main takeaway of Figure 2: as long as the latency together with the run time on edge nodes is smaller than the overall latency of the cloud nodes (computational efforts are marginal for $n \lessapprox 30$), the user benefits from using the edge. As soon as the function run time exceeds circa 0.8 seconds (for problem size of $n \approx 33$) the user no longer benefits from executing the Fibonacci workload on edge nodes. For bigger problem sizes like $n = 40$ the difference in user time between edge and cloud grows exponentially (46.11s vs 18.5s).

This takeaway is further evidenced by looking at Figure 3. For input sized $n = 5$ up to $n = 25$ the CPU usage on the edge and cloud stays roughly the same. Beginning with $n = 30$ we can observe the CPU usage starting to differ between the edge and cloud and by looking at Figure 2 we can see that this is exactly the point where the cloud starts to gain on the edge. Input sizes $n = 35$ and $n = 40$ have a much higher CPU usage on the edge than on the cloud which is also reflected in the user time.

2) User Time Variance: Due to the load generated by the hey tool we are able to show the variances between a homogeneous and mixed workload passed to the nodes. Figure 4 shows the measurements obtained by executing the experiment described previously in subsubsection IV-A3. We are clearly able to observe that the sequential execution (homogeneous workload) total user-time suffers much less from deviation than in the case of the parallel execution of the functions (mixed workload). For the mixed workload on the edge, we can see a deviation of ± 20 sec. compared to the average execution time.

The aforementioned experiment also provided us with detailed measurements for every function and not just the total overall execution time. Therefore, with Figure 5 we are able to further prove that a mixed workload is much more complex and hard to handle than a homogeneous one. The boxes indicate the span between the 25th and 75th percentile of user time in seconds.

We observe very similar values on the edge environment for sequential and parallel workload execution while the overall deviation varies a lot compared to the cloud environment. Since the boxes are way smaller on the cloud we conclude that cloud computing is able to provide a way more stable execution while also being two to four times faster than the edge environment. In the most extreme cases, a single function invocation on the cloud can deviate by over eleven times the execution time between homogeneous and mixed workloads executions.

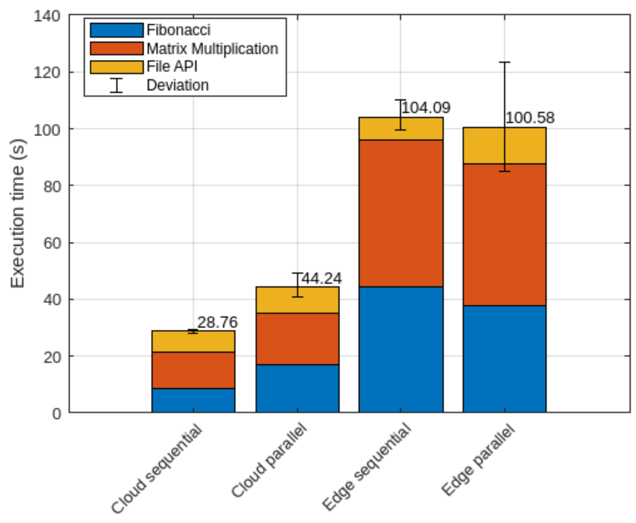


Fig. 4. Variance of the execution time in seconds for all experiments in sequential and parallel workload (for cloud and edge)

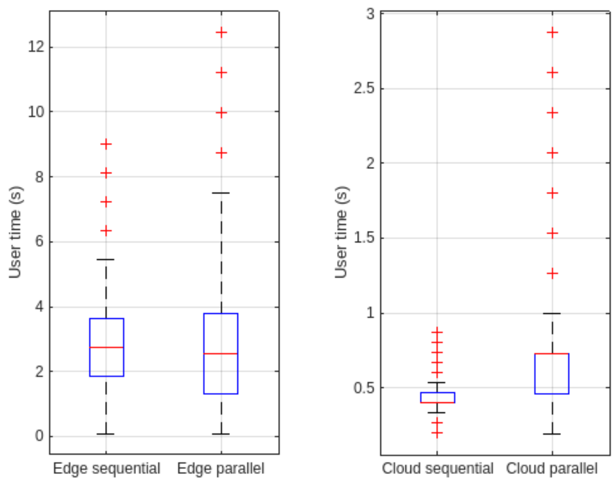


Fig. 5. Variance in user time in seconds for Fibonacci function for the sequential and parallel workload on edge and cloud environment

3) Overall Results: The gathered metrics of CPU utilization for the experiments run in the cloud are shown in the Figure 6. For a homogeneous workload, as expected, we observe the highest peaks of CPU usage, approximately 13 cores used, for the Matmul function. Followed by peaks of approximately 7 cores used generated by the Fibonacci function instances. The File-upload function never exceeds 2.5 cores of CPU utilization. Starting from the 14:49:00 point on the x-axis we can see the utilization of CPU resources generated by heterogeneous workload. Overall usage of CPU per function instance is lower, indicating the distribution of available resources among all function types. Matmul function instances peak at approximately 8.5 cores used and Fibonacci at slightly above five cores.

Figure 7 depicts the memory usage while executing the same experiment. While executing homogeneous workloads, we can note the stable load of approximately 115 MiB for the File-upload function, an average load of about 95 MiB for matrix multiplication calculations, and an average load of approximately 80 MiB for the Fibonacci function. While the execution of the heterogeneous workload the overall memory usage of each function type is significantly higher.

Finally, the bandwidth is mainly tested by the file transfer function as shown in the Figure 8. We observe peaks of approximately 1.2 MB/s for Rx bandwidth and peaks of approximately 10.2 MB/s for the Tx bandwidth while executing functions sequentially. During the parallel run of different function types the Rx and Tx bandwidth drops significantly.

The edge experiments are structured in the same way as the experiments performed on the cloud. Figure 9 shows peaks of six cores used for the Fibonacci function, around 8.4 cores for the matrix multiplication, and 1.5 cores used for the file API function. The memory usage peaks at about 260 MiB for matrix multiplication, and 150 MiB for Fibonacci (Figure 10). When it comes to the file-upload function, memory usage is stable and leveled around 20 MiB. Finally, there is an overall lower Rx and Tx bandwidth for the edge environment compared to the cloud. As Figure 11 shows: we observe peaks of 620 kB/s of Rx bandwidth and 6.1 MB/s of Tx bandwidth for the file transfer function. We can also see similar differences, between the execution of a homogeneous workload (described above) and the execution of a heterogeneous workload, as while running experiments on the cloud environment.

V. Discussion

In the following section, we reflect on the conducted experiments and discuss if they meet our expectations. We will further reason about our design choices and the limitations that arise from them. We also take a look into future extensions. It is important to mention, that we, with our experiments, want to demonstrate the capabilities of our monitoring platform to measure the performance of cloud and edge systems. Based on the results, performance issues and bottlenecks can be derived, and furthermore, actions for addressing those issues can be recommended. For example, for a particular function that can be easily deployed in OpenFaas, we can test whether it should be deployed in the cloud or in the edge environment.

Before conducting the experiments, we expected that for non-compute-heavy tasks, the response time of the edge would be smaller due to the low theoretical latency. Therefore the use of edge computing devices would be beneficial. This advantage would drop off for more compute-intensive tasks in favor of cloud computing, due to much more resources being available.

In the first experiment, we measured the actual time it takes to complete a task. Our experiment reflects the fact that edge computing performs better if the desired task does not require a lot of resources and computing power. The edge is designed to handle tasks that require low latency and fast response times and is typically located closer to the end user than the cloud environment. From this, it can be deduced that is necessary to consider the run time of the function of both infrastructures and the transfer overhead needed to send the result back to the user. If the total user time of the request on the edge is shorter than in the cloud, then edge computing is likely to be the better choice for a given function.

We also found that the deviation in total run time was higher for heterogeneous workloads compared to homogeneous workloads. In the cloud example, a function may take up to 11 times longer to execute. This suggests that different serverless functions executed in the cloud or on the edge are more mutually dependent when they are part of a heterogeneous workload, as opposed to being executed sequentially.

The experiments show that our system is able to collect metrics that reveal the performance of a function for a given infrastructure. Nevertheless, there are also some limitations to our system that should be considered when interpreting the results. We simulate infrastructure using the Continuum framework, which is a useful tool but does not fully reflect a real-world setting. In order to establish more robust and actual findings about cloud and edge computing, it would be useful to conduct further experiments using real-world applications and data. Additionally, we only tested one configuration per edge and cloud, and further investigations with different configurations could provide a more complete picture of the performance and capabilities of these technologies.

There are several directions in which our system could be enhanced in the future. For example, we could use created monitoring platform to classify applications and offload them to either the cloud or the edge depending on the resource load. This could allow us to better optimize the use of these technologies, and ensure that tasks are being run on the most appropriate platform. Additionally, further testing could be conducted to explore the use of edge computing in IoT environments, where data is generated locally and the transport of data is usually a bottleneck.

From what we learned about the computing continuum, it is an emerging challenge to properly build a robust and efficient continuum environment. The challenge lies in the proper task scheduling, hence offloading appropriate tasks to the right node workers so that we can use the full potential of available resources. For example, edge nodes could be used as data collectors and aggregation points, which would perform data preprocessing. This way the end-users and smart IoT devices could upload data with low latencies and later on, the preprocessed data would be passed to the cloud cluster. In such a case, we would make good use of low-latency edge machines and powerful compute nodes in the cloud. At the same time allowing end devices to offload the workload quickly.

VI. Conclusion

The purpose of this research project was the development of a fully automated real-time benchmarking and monitoring platform. We tested this system by conducting a series of experiments where we evaluated the performance of serverless functions deployed in a cloud and edge environment. We used two different configurations for each environment. We assigned less computing power, but lower latency to the edge than to the cloud environment. Furthermore, we deployed three functions to simulate different workloads and gathered the resulting metrics such as CPU usage, memory usage, and network bandwidth using our monitoring platform.

The results of our experiments indicated that using the edge for Fibonacci calculation was faster for problems of sizes $n < 33$ due to its lower latency. At the same time, the Edge proved to be limited by its weaker computational capabilities for more complex problems (Fibonacci for $n \geq 33$). We observed a significant increase in the difference in the total user time between the edge and cloud for larger Fibonacci numbers calculation.

In addition, we tested the collective impact of the functions on each other and found that sequential (homogeneous workload on the nodes) executions were less prone to deviation in execution time than parallel executions (mixed workload on nodes). Our experiments also show that a mixed workload is more challenging to handle than a homogeneous workload.

The Design and implementation of the automated solution make it possible for easy testing of various types of functions in the future. Currently, we are measuring three metrics, but it is possible to expand this in future work. Therefore, the platform could be used together with real-world applications to further research the compute continuum.

From the obtained results, it is clear that Edge and Cloud environments have different characteristics and should be used accordingly to it in order to perform well. Surly edge computing handles well small, non-compute demanding tasks very efficiently, and with low latencies. Whereas for compute-heavy, complex tasks cloud computing should be used since it can utilize more resources and therefore outperform the benefit of low-latency edge devices. Work done within this project may be used also for creating an automated classification platform, for example. Such a platform could be used for automating the decision-making process of whether particular tasks should be processed in the cloud or offloaded to the edge in order to obtain the maximum performance of the hybrid infrastructure.

References

[1] C. Renaud, "The Edge-to-Cloud Continuum," https://www.delltechnologies.com/asset/en-th/products/dell-technologies-cloud/industry-market/the-edge-to-cloud-continuum.pdf, 2021, [Online; accessed 15-December-2022].

[2] @Large Research Group, "Massivizing Computer Systems," https://atlarge-research.com/, 2022, [Online; accessed 21-December-2022].

[3] ESA, NASA, "Image set used in the project." https://esahubble.org/media/archives/media/zip/hst_media_0017.zip, 2022, [Online; accessed 21-December-2022].

[4] M. Jansen, A. Al-Dulaimy, A. V. Papadopoulos, A. Trivedi, and A. Iosup, "The spec-rg reference architecture for the edge continuum," 2022. [Online]. Available: https://arxiv.org/abs/2207.04159

[5] The Kubernetes Authors, "Kubernetes Documentation," https://kubernetes.io/docs/home/, 2022, [Online; accessed 15-December-2022].

[6] OpenFaaS Author(s), "OpenFaaS – Serverless Functions Made Simple," https://docs.openfaas.com/, 2021, [Online; accessed 15-December-2022].

[7] Docker Inc., "Docker Hub," https://hub.docker.com/, 2022, [Online; accessed 15-December-2022].

[8] Prometheus Authors, "Prometheus," https://prometheus.io/docs/introduction/overview/, 2022, [Online; accessed 15-December-2022].

[9] Prometheus Operator, "kube-prometheus," https://github.com/prometheus-operator/kube-prometheus, 2022, [Online; accessed 15-December-2022].

[10] Grafana Labs, "Grafana Loki Documentation," https://grafana.com/docs/loki/latest/, 2022, [Online; accessed 15-December-2022].

[11] ——, "Documentation," https://grafana.com/docs/, 2022, [Online; accessed 15-December-2022].

[12] Rakyll, "hey," https://github.com/rakyll/hey, 2021, [Online; accessed 16-December-2022].

[13] Fizzadar, "Pyinfra project," https://pyinfra.com/, 2022, [Online; accessed 15-December-2022].

[14] The Kubernetes Authors, "Kubernetes Documentation regarding milliCPU unit." https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu, 2022, [Online; accessed 21-December-2022].

APPENDIX

A. Time sheet

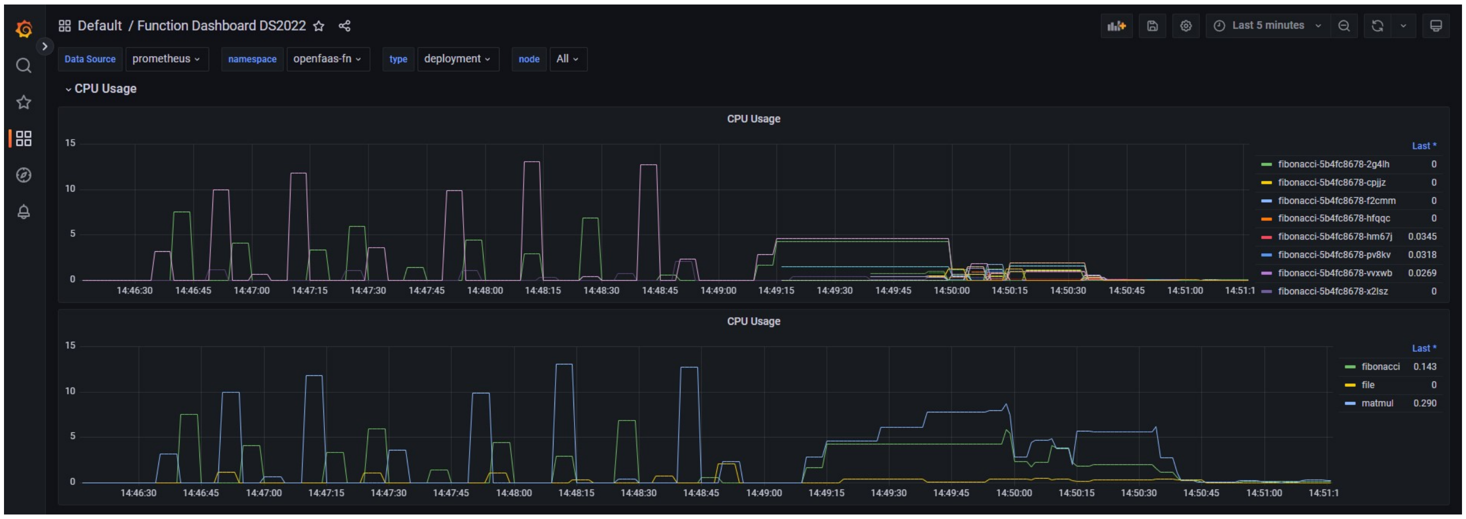
Task name	**Amount (h)**
Thinking	30
Developing	55
Experimenting	80
Analyzing	25
Writing	60
Meetings	10
Other activities (Teambuilding etc.)	15
Total	**275**

TABLE III
TIME SPENT FOR DIFFERENT TASKS IN HOURS

B. hey Output 5th Fibonacci

```
Summary:
  Total:        38.2428 secs
  Slowest:      7.8105 secs
  Fastest:      0.0572 secs
  Average:      1.8192 secs
  Requests/sec: 26.1487

  Total data:   43138 bytes
  Size/request: 43 bytes

Response time histogram:
  0.057 [1]     |
  0.833 [328]   |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  1.608 [158]   |xxxxxxxxxxxxxxxxxxx
  2.383 [184]   |xxxxxxxxxxxxxxxxxxxxxx
  3.159 [131]   |xxxxxxxxxxxxxxxx
  3.934 [113]   |xxxxxxxxxxxxxx
  4.709 [48]    |xxxxxx
  5.485 [20]    |xx
  6.260 [6]     |x
  7.035 [9]     |x
  7.810 [2]     |

Latency distribution:
  10% in 0.1495 secs
  25% in 0.3127 secs
  50% in 1.6660 secs
  75% in 2.8119 secs
  90% in 3.7904 secs
  95% in 4.5934 secs
  99% in 6.3813 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0006 secs, 0.0572 secs, 7.8105 secs
  DNS-lookup:   0.0000 secs, 0.0000 secs, 0.0000 secs
  req write:    0.0012 secs, 0.0000 secs, 0.0408 secs
  resp wait:    1.8170 secs, 0.0567 secs, 7.8104 secs
  resp read:    0.0003 secs, 0.0000 secs, 0.0447 secs

Status code distribution:
  [200] 1000 responses
```

C. Grafana Dashboards described in subsubsection IV-A3


Default / Function Dashboard DS2022

Data Source prometheus · namespace openfaas-fn · type deployment · node All

Fig. 6. Cloud Environment - CPU usage (1 unit on the y-axis is equivalent to 1000 milliCPUs [14])

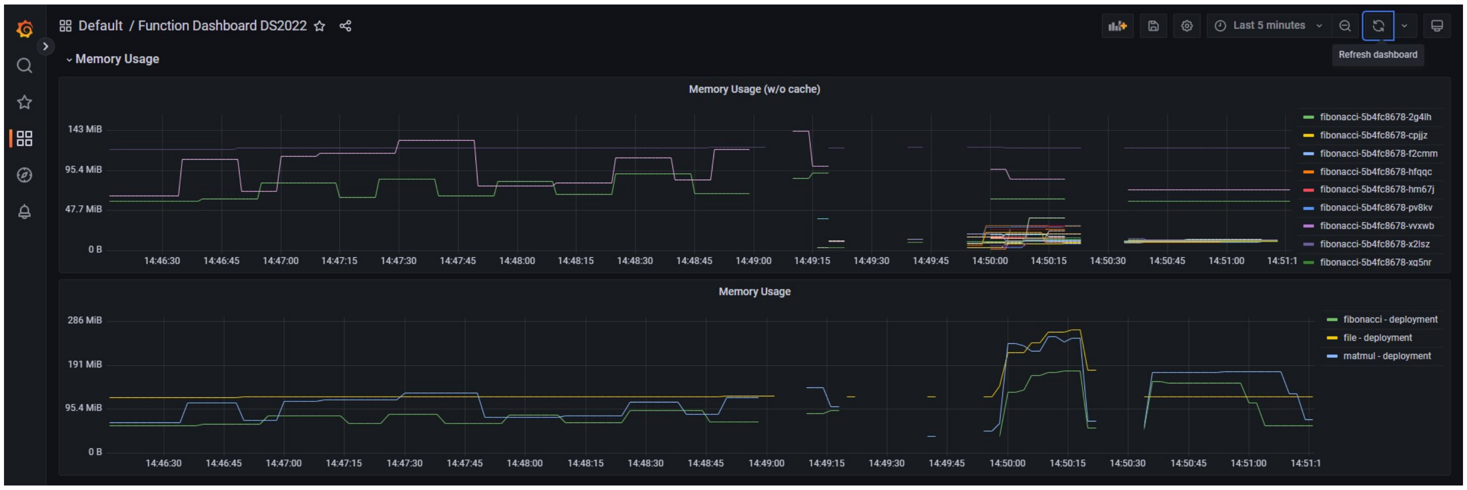

Default / Function Dashboard DS2022

Fig. 7. Cloud Environment - Memory usage

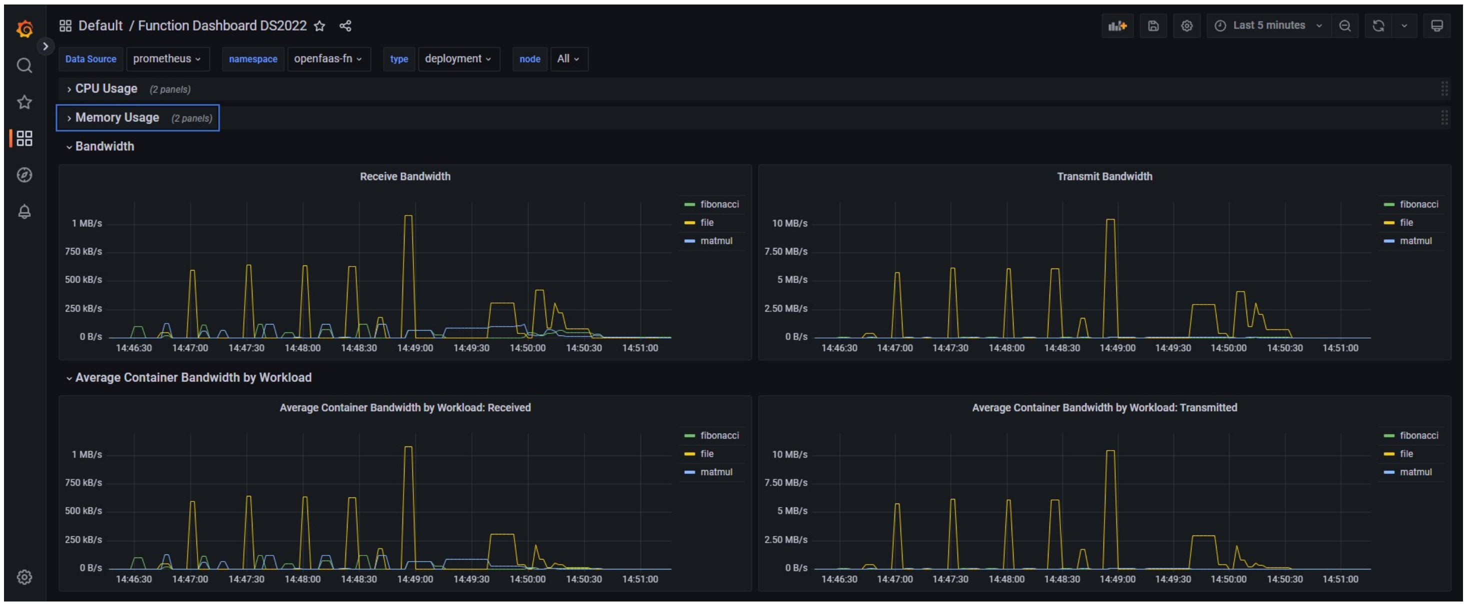


Fig. 8. Cloud Environment - Bandwidth

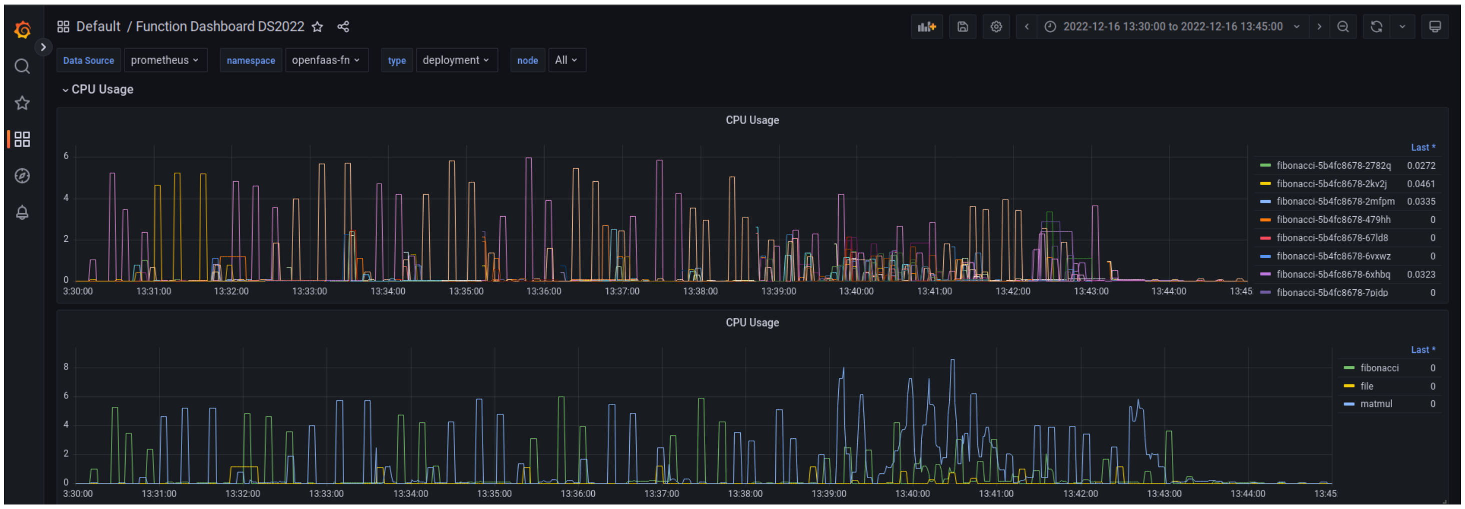

Default / Function Dashboard DS2022
2022-12-16 13:30:00 to 2022-12-16 13:45:00
Data Source
prometheus
namespace
openfaas-fn
type
deployment
node
All
CPU Usage
Last *
fibonacci-5b4fc8678-2782q 0.0272
fibonacci-5b4fc8678-2kv2j 0.0461
fibonacci-5b4fc8678-2mfpm 0.0335
fibonacci-5b4fc8678-479hh 0
fibonacci-5b4fc8678-67ld8 0
fibonacci-5b4fc8678-6vxwz 0
fibonacci-5b4fc8678-6xhbq 0.0323
fibonacci-5b4fc8678-7pjdp 0
fibonacci 0
file 0
matmul 0
13:30:00
13:31:00
13:32:00
13:33:00
13:34:00
13:35:00
13:36:00
13:37:00
13:38:00
13:39:00
13:40:00
13:41:00
13:42:00
13:43:00
13:44:00
13:45

Fig. 9. Edge Environment - CPU usage (1 unit on the y-axis is equivalent to 1000 milliCPUs [14])

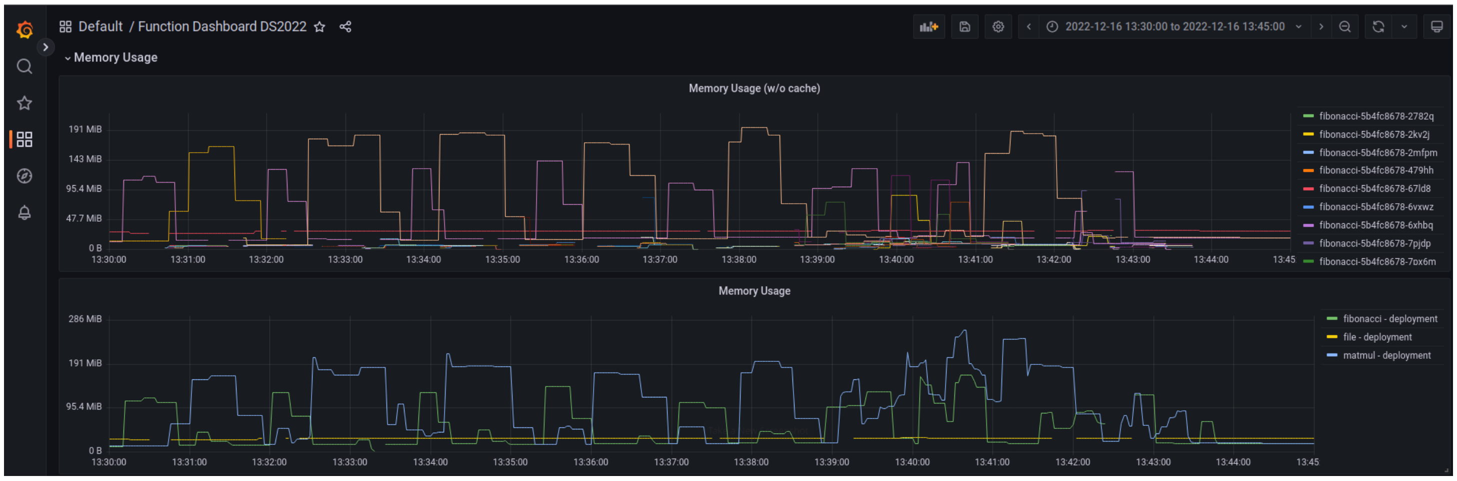

Default / Function Dashboard DS2022
2022-12-16 13:30:00 to 2022-12-16 13:45:00
Memory Usage
Memory Usage (w/o cache)
191 MiB
143 MiB
95.4 MiB
47.7 MiB
0 B
13:30:00
13:31:00
13:32:00
13:33:00
13:34:00
13:35:00
13:36:00
13:37:00
13:38:00
13:39:00
13:40:00
13:41:00
13:42:00
13:43:00
13:44:00
13:45
fibonacci-5b4fc8678-2782q
fibonacci-5b4fc8678-2kv2j
fibonacci-5b4fc8678-2mfpm
fibonacci-5b4fc8678-479hh
fibonacci-5b4fc8678-67ld8
fibonacci-5b4fc8678-6vxwz
fibonacci-5b4fc8678-6xhbq
fibonacci-5b4fc8678-7pjdp
fibonacci-5b4fc8678-7px6m
Memory Usage
286 MiB
191 MiB
95.4 MiB
0 B
fibonacci - deployment
file - deployment
matmul - deployment

Fig. 10. Edge Environment - Memory usage

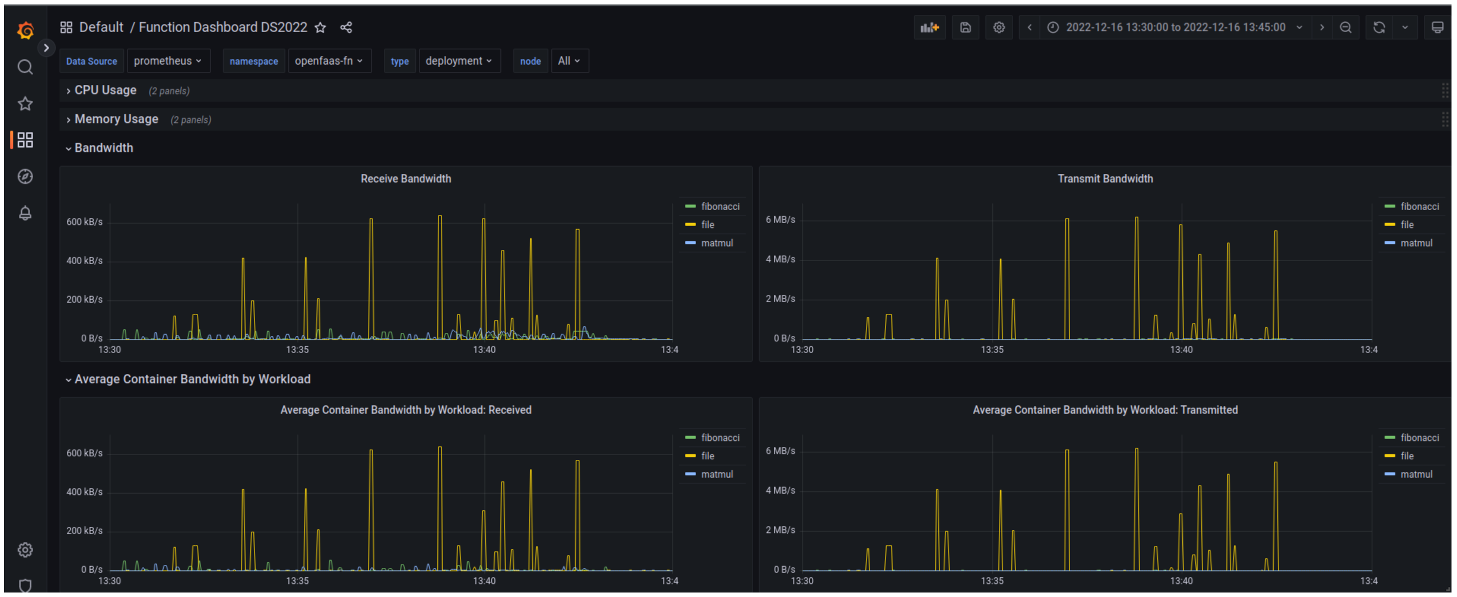


Fig. 11. Edge Environment - Bandwidth