Serverless Computing in the Edge Continuum Group 2

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

Mathijs 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

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

***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]. Thus, it is essential today to further understand and analyze the behavior of the computing continuum under different workloads. Further, there is currently a lack of 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 that collects and displays execution-time data in real-time is created. For pushing the cloud and the edge to its limits three different serverless functions were created - 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 rest of the paper goes through the background of the created tool, the requirements (§ II) that had to be met, and motivated behind them. 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 and tradeoffs 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). The project has as an objective the development of three serverless functions that simulate different workloads on the system, and monitor their execution on the university DAS-5 cluster.

The project comes with some pre-defined requirements.

Design around pre-defined framework stack: the scope of this project is to develop the system using OpenFaas [2] and Continuum [3]. Every other component of this paper is a personal choice of the authors.

Experiment design: the experiments essentially consist of the target system, cloud or edge, and variations of workloads. The same metrics (introduced below) are measured in each of the experiments. The workloads are represented by the three serverless functions: *Matmul, Fibonacci* and *File-upload*. The execution of these functions is executed in a way to simulate both homogeneous and mixed workloads. Parallel runs of all the functions are performed to simulate multiple users using the system at the same time.

Set of metrics: the metrics included in the monitoring platform are *CPU usage, Memory usage*, *Network bandwidth* and user-time.

Analysis of the experiments: insight is discussed based on the data collected during the experiments. The goal is to find clear distinctions between the performance of the cloud deployment compared to the edge. Furthermore, the study also focuses on understanding how the system reacts to significant variances in system load, and how fast the system adapts (upscale and downscale).

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 properly re-configuring all the necessary components for every run. *Continuum* automates the setup process of the cloud and edge environments. It was also necessary to automate the components focusing on logs aggregation and visualization. The monitoring side of the project is a key contribution to this project and is achieved with help of *pyinfra*[4].

III. System Design

In the following chapter, we present used technologies and how we combine them to run and measure experiments. The whole system runs on a cluster provided by VU Amsterdam.

A. Technologies

1) continuum: This framework[3] 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 edge environment. While the edge configuration supports low latencies (5ms) and low computational power (CPU quota of 0.5) over 10 nodes with 2 cores each, the cloud configuration sets higher latencies (50ms) and a CPU quota of 1 over 3 nodes with 6 cores each.

2) OpenFaaS: The open source function-as-a-service provider[2] 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. subsubsection IV-A2) are available as images on Docker Hub [5] from where 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.

3) Kubernetes: This open-source solution[6] 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.

4) Prometheus: After everything is up and running we use this tool[7] 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[8]. 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[9]. 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[10] 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 B).

8) hey: To generate specific workloads and simulate a lot of concurrent HTTP requests to the deployed functions we used hey[11]. 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 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. Compared to the *Ansible* 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[4] configurations. There are two important files, *inventory.py* file which contains user-specific and Continuum environment-specific data, such as username, ssh credentials, or cloud-controller IP address. The second file called *deployment.py* contains all 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. 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*[2] which is installed in a *Kubernetes*[6] cluster deployed by *continuum*[3]. By deploying a ready-made monitoring platform called *kube-prometheus*[8] 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*[7] 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*[9] 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*[10]. 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 resources available on each node as well as 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 10 and the scaling factor to 50% - meaning that scaling happens firstly to 6 replicas and then to 10 in case of such demand. This way we could fully utilize available resources in our edge configuration (10 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.

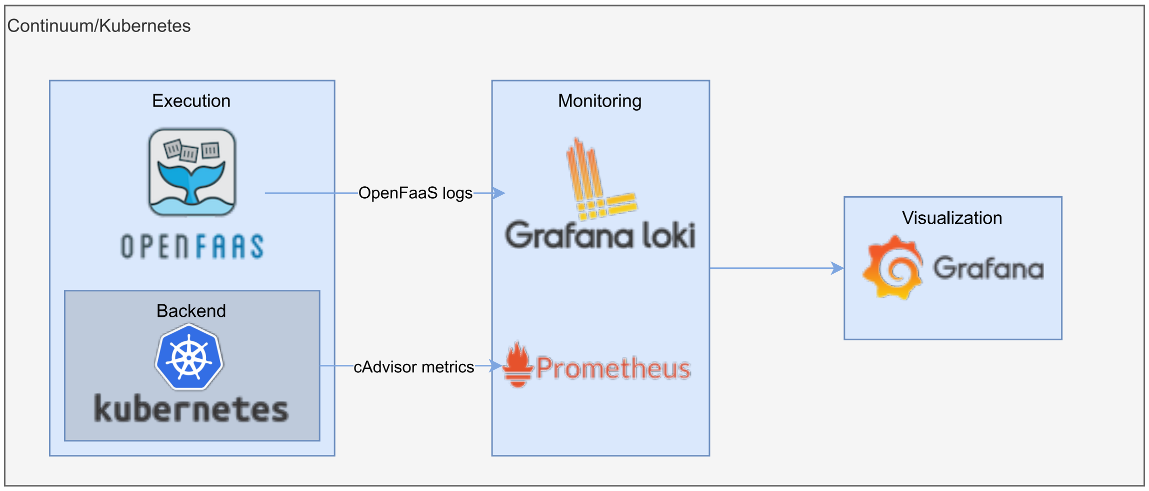


Fig. 1. Project architecture

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 subsubsection III-A8 as 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 5th 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 5 times in a sequential and parallel manner. An example output is provided in subsection A.

B. Results

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 $n \sim 32$ the cloud is returning faster to the user than the edge. For $n \geq\sim 33$ 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 $n \leq\sim 30$), the user benefits from using the edge. As soon as the function run time exceeds circa 0.8 seconds (for problem size of $n \sim$) 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).

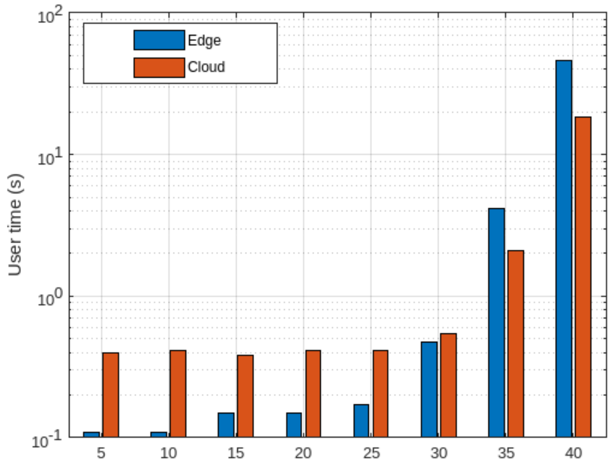


Fig. 2. User Time Fibonacci (Edge vs. Cloud)

2) User Time Variance: Due to the load generated by the hey tool (s. subsubsection III-A8) we are able to show the variances between a homogeneous and mixed workload passed to the nodes. Figure 3 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.

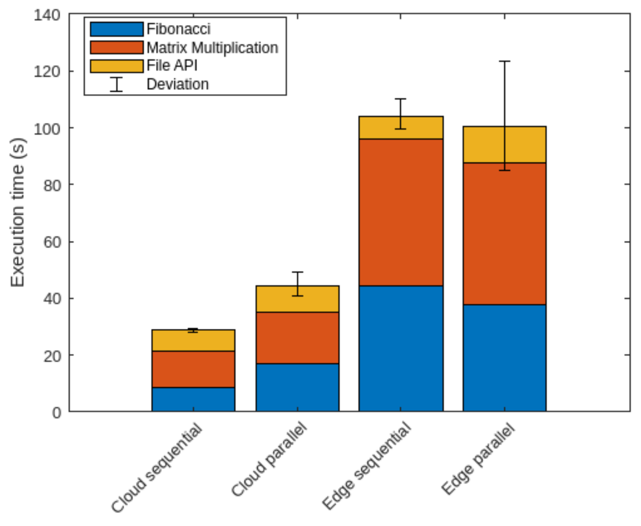


Fig. 3. Variance in user time for all experiments

The aforementioned experiment also provided us with detailed measurements for every function and not just the total overall execution time. Therefore, with Figure 4 we are able to further prove that a mixed workload is much more complex and hard to handle than a homogeneous one. In the most extreme cases, a single function invocation on the cloud can deviate by over 11 times the execution time between homogeneous and mixed workloads executions.

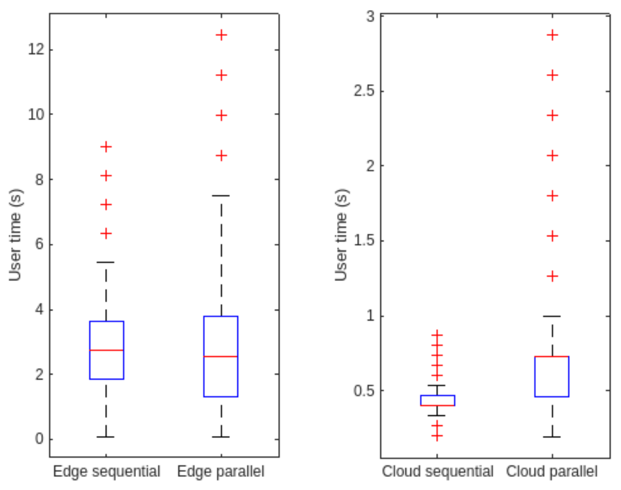


Fig. 4. Variance in user time for Fibonacci function

V. Discussion

Our experiments reflect the fact that edge computing performs better if the desired task does not require a lot of resources and computing power. This is in line with our expectations, as edge computing 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. In order to determine whether edge computing or cloud computing is the more suitable option for a given task, it 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.

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. Parallel calls of the serverless functions should therefore be avoided.

In designing our system, we wanted to operate from the user's perspective, so we only sent requests from our platform to the simulated edge or cloud. We simulated high and low CPU usage by using variable parameters for the Fibonacci function to simulate fluctuating workloads. In a real-world system, the load is also only predictable to a certain extent.

There are some limitations to our system that should be considered when interpreting the results. We used a simulator called Continuum to run our experiments, 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 pre-processing. This way the end-users and smart IoT end devices could upload data with low latencies and later on, the pre-processed data would be passed to the cloud cluster. In such a case, we would make good use of low-latency, low compute power 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 was to conduct a series of experiments to evaluate the performance of serverless functions deployed in the cloud and in the edge environment. To accomplish this, we developed a fully automated real-time benchmarking and monitoring platform and ran the experiments in a simulated environment. We used two different configurations to simulate the cloud and edge infrastructure on the DAS5 cluster. We deployed three functions that simulate different workload types in a real-world setting, including CPU usage, memory usage, and network bandwidth.

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 $n >= 33$). 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.

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

From 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 in the future for creating an automated classification platform. Such a platform could be used for deciding whether particular tasks should be processed in the cloud or offloaded to the edge in order to obtain the maximum performance of the compute continuum.

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

[3] 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

[4] Fizzadar, “Pyinfra project,” https://pyinfra.com/, 2022, [Online; accessed 15-December-2022].

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

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

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

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

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

[10] ——, “Documentation,” https://grafana.com/docs/, 2022, [Online; accessed 15-December-2022].

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

Appendix

A. 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
```

B. Grafana Dashboards of subsubsection IV-A3

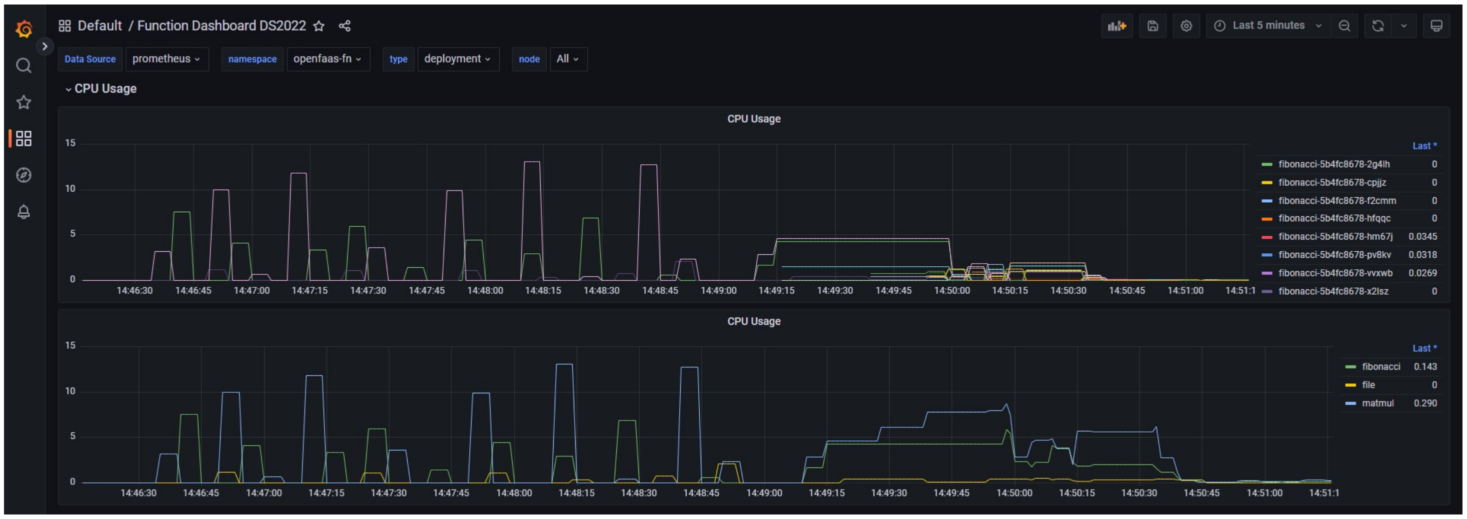


Fig. 5. Cloud Environment - CPU usage

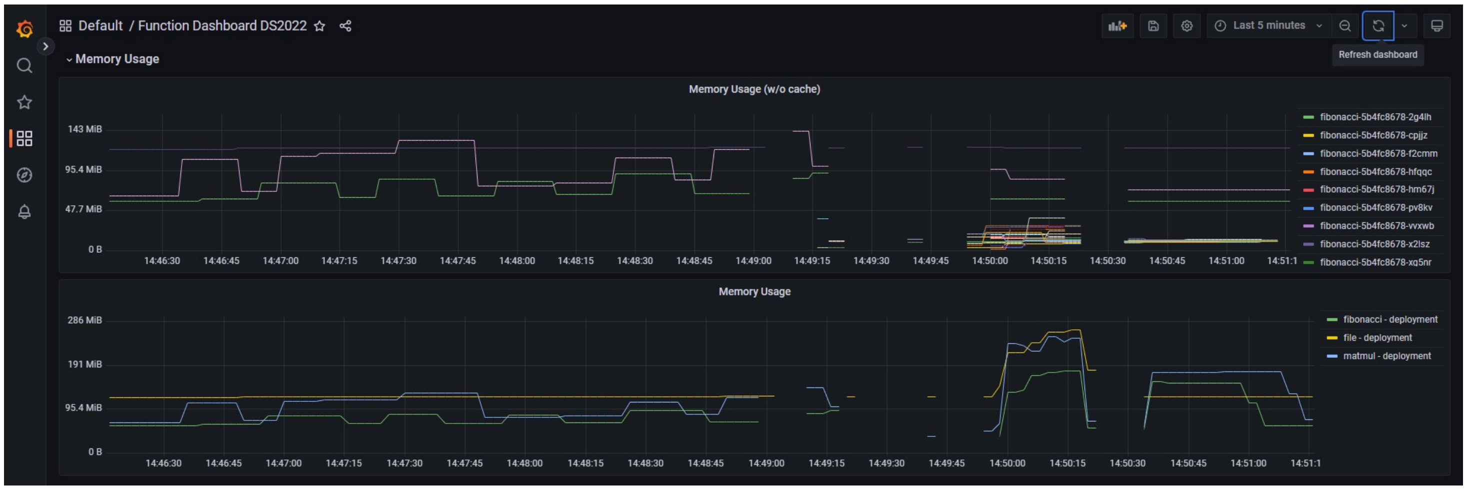


Fig. 6. Cloud Environment - Memory usage

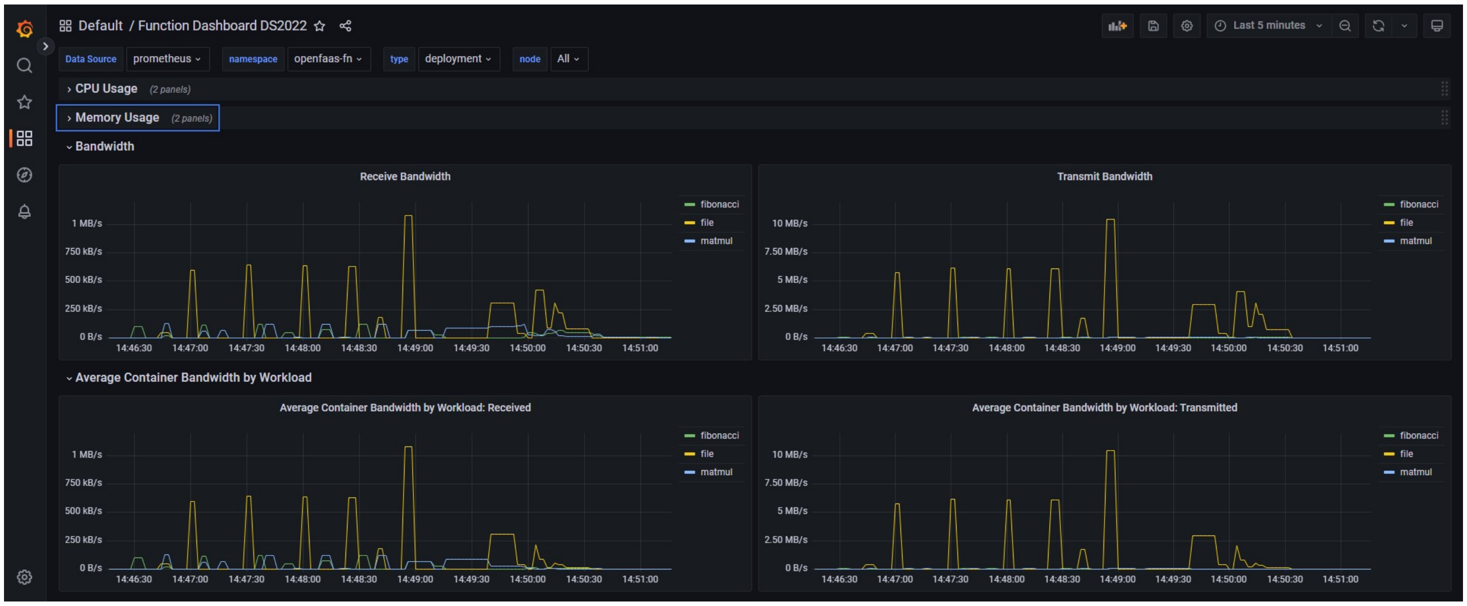
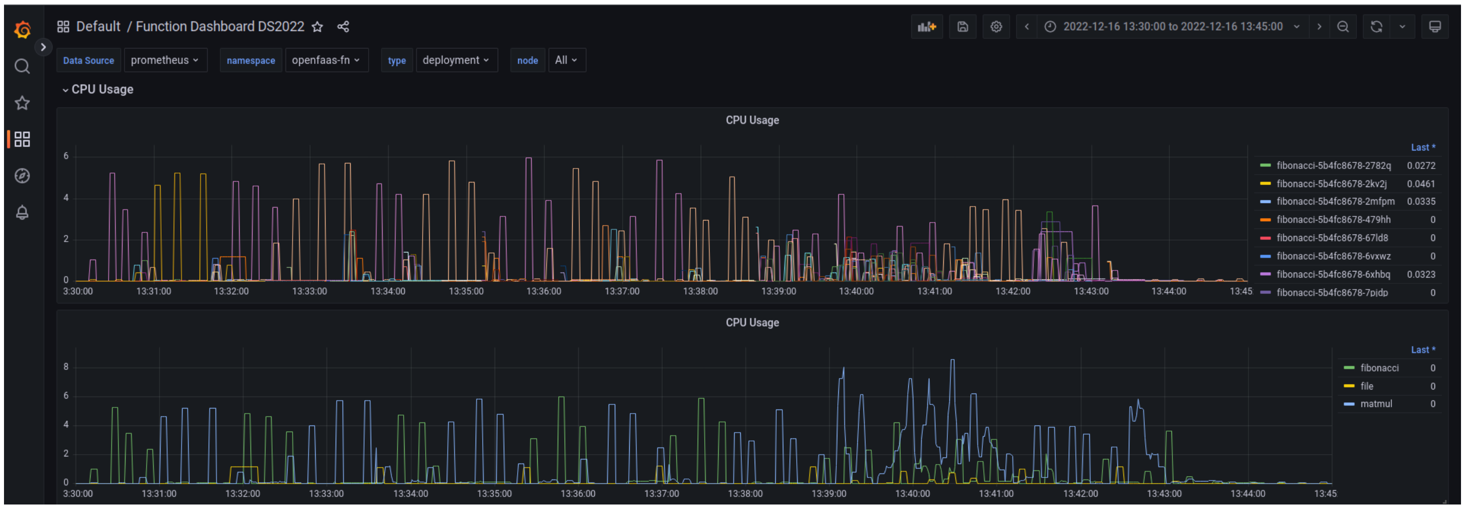


Fig. 7. Cloud Environment - Bandwidth



Default / Function Dashboard DS2022
Data Source prometheus
namespace openfaas-fn
type deployment
node All
2022-12-16 13:30:00 to 2022-12-16 13:45:00
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

Fig. 8. Edge Environment - CPU usage

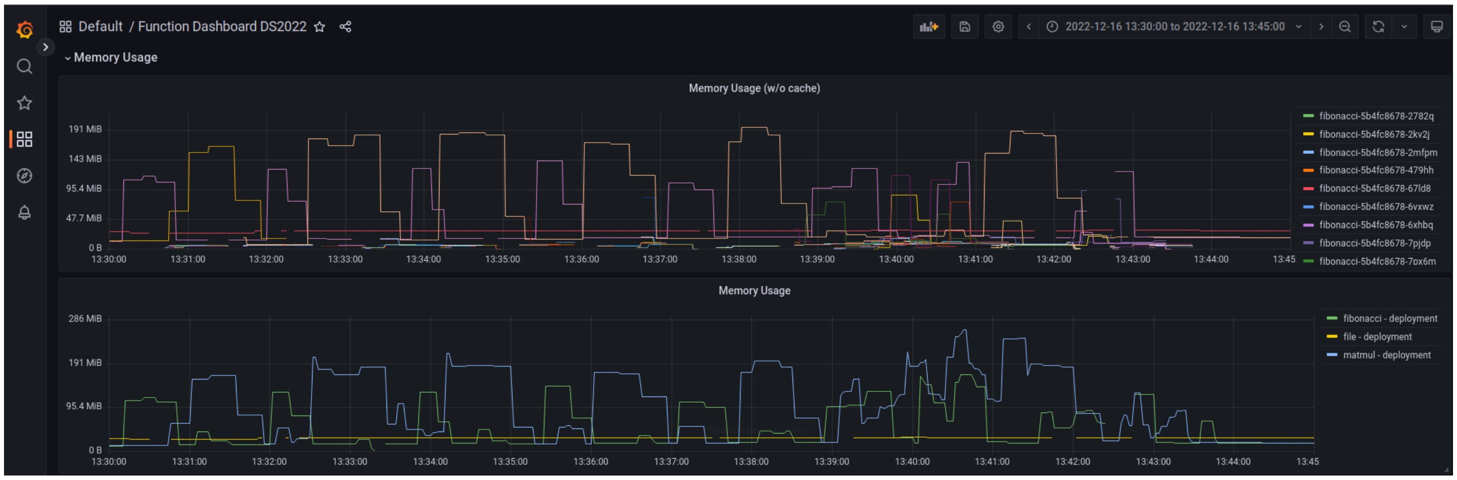

Default / Function Dashboard DS2022
2022-12-16 13:30:00 to 2022-12-16 13:45:00
Memory Usage
Memory Usage (w/o cache)
191 MiB
143 MiB
95.4 MiB
47.7 MiB
0 B
13:30:00
13:31:00
13:32:00
13:33:00
13:34:00
13:35:00
13:36:00
13:37:00
13:38:00
13:39:00
13:40:00
13:41:00
13:42:00
13:43:00
13:44:00
13:45
fibonacci-5b4fc8678-2782q
fibonacci-5b4fc8678-2kv2j
fibonacci-5b4fc8678-2mfpm
fibonacci-5b4fc8678-479hh
fibonacci-5b4fc8678-67ld8
fibonacci-5b4fc8678-6vxwz
fibonacci-5b4fc8678-6xhbq
fibonacci-5b4fc8678-7pjdp
fibonacci-5b4fc8678-7ox6m
Memory Usage
286 MiB
191 MiB
95.4 MiB
0 B
fibonacci - deployment
file - deployment
matmul - deployment

Fig. 9. Edge Environment - Memory usage

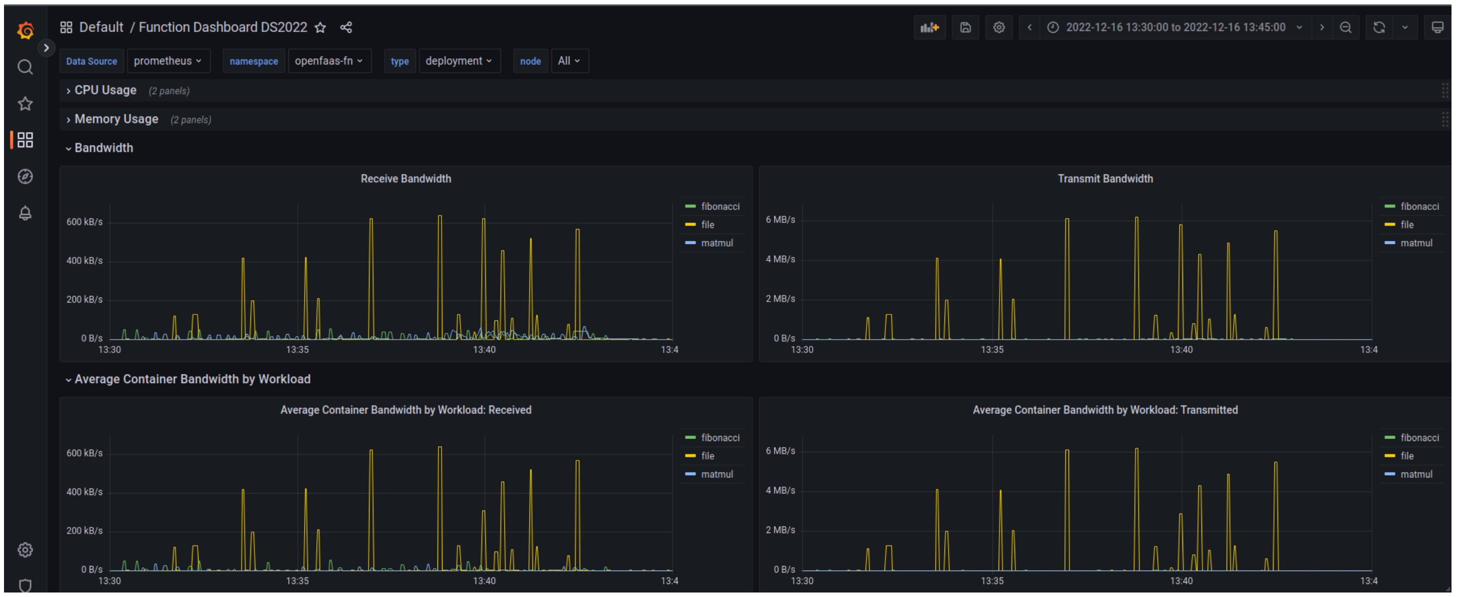


Fig. 10. Edge Environment - Bandwidth