An End-to-End Framework for Benchmarking Edge-Cloud Cluster Management Techniques

Philipp Raith, Thomas Rausch, Paul Prüller, Alireza Furutanpey, Schahram Dustdar
Distributed Systems Group, TU Wien
Vienna, Austria
lastname@dsg.tuwien.ac.at

***Abstract*—This paper presents a framework for defining, performing, and analyzing distributed load testing experiments for benchmarking edge-cloud clusters. This end-to-end workflow helps researchers build reproducible environments to evaluate cluster management techniques. Our implementation extends the open source tool Galileo by adding support for distributed execution on Kubernetes clusters, additional system monitoring instruments, as well as out-of-the box experiment workloads. We focus on providing tools that run across popular CPU architectures and provide a set of representative workloads, such as edge AI functions. We demonstrate our framework's capabilities in a set of experiments based on use cases commonly found in edge computing systems research. Additionally, we show that the resource usage of our system is minimal and that it can run on resource-constrained devices.**

***Index Terms*—Edge computing, Cloud computing, Benchmarking, Cluster management**

I. Introduction

To evaluate edge-cloud cluster management techniques, researchers often rely on building testbeds that are tailored to evaluate the particular technique [1]–[4]. In the absence of publicly available edge infrastructure, testbeds are, next to simulations [5] and emulations [6], the only way to evaluate systems approaches. However, there is currently no generally accepted way of creating such testbeds or representative benchmark workloads.This makes it hard for other researchers to reproduce results and prevents the community from comparing works easily. Reproducible experiments for edge-cloud clusters aid in evaluating cluster management techniques and include the following steps: (1) setting up the testbed, (2) generate workload patterns, (3) manage application deployments, (4) orchestrate experiments, (5) instrumentation and monitoring, (6) analyze experiment data. Performing these steps manually is error-prone and impedes reproducibility [7]. An experiment and analytics framework allows researchers to evaluate their systems approaches while opening up other possibilities crucial to developing and testing cluster management techniques. For example, some cluster management techniques are based on optimizations that rely on historical data [8]. An experiment framework gathers data in a streamlined way and significantly reduces manual steps and allows scaling in terms of applications and cluster sizes.

Reducing performance degradation caused by the interference of co-locating monitoring applications on the same device is challenging for cloud deployments and more so for edge-cloud clusters [9] that are typically heterogeneous and resource-constrained. Hence, we require hardware architecture agnostic, lightweight and non-intrusive instrumentation tools to measure resource usage and power consumption. Further, trace-driven simulations require real-world data to run realistic simulations [10]. An experiment framework helps record this data and enables the community to perform complex simulations based on real-world data.

Based on the abovementioned observations and requirements, we conclude that such a framework should provide scalable and reproducible experiments, simple configuration of workload and applications, fine-grained monitoring telemetry, storage for analysis data, and standardised analysis techniques.

To this end, we built *Galileo* [11], a distributed load testing framework for edge computing environments. However, while Galileo allows the definition and execution of experiments, it cannot operationalize those experiments on existing cluster testbeds nor provides a streamlined way to instrument and collect monitoring data from such clusters.

In this paper, we present an end-to-end experiment framework based on an extension of Galileo and the combination of different components. We motivate our work by introducing different use cases typically encountered in evaluating cluster management techniques, demonstrating our framework's efficacy by executing them, and showcasing the results.

Edge Run[1] is a collection of projects aimed to help developers create, monitor, and test applications for the edge-cloud continuum. Part of the Edge Run project is Galileo[2], a distributed load testing framework [11]. Galileo can start clients, generate configurable requests, and store trace and telemetry data in a SQL database, which is unfit for high volumes of time-series data. The caveat of Galileo is that it requires a manual setup and does not offer any kind of integration with modern orchestration services such as Kubernetes. In this work, we introduce a framework that extends Galileo, solves these issues and provides an analytic framework that completes the end-to-end framework. Further, we introduce introduce components mimicking the characteristics of edge-cloud clusters.

[1]https://github.com/edgerun
[2]https://github.com/edgerun/galileo

Specifically, our contributions are as follows:

- An end-to-end experiment and analytics framework using a container orchestration service based on Galileo.
- An open-source repository of benchmark applications that can be used within our framework.

To evaluate and demonstrate the end-to-end process, we conduct experiments based on representative use cases.

II. Galileo experiments

This section describes the aim of our framework based on the experiments and applications we intend to support and the characteristics of edge-cloud clusters.

A. Aim of the framework

The framework aims to provide tools to perform benchmarks on edge-cloud compute clusters. Benchmarks aim to profile single devices or evaluate cluster management techniques in scenario experiments. The edge-cloud continuum deploys different types of hardware; therefore, all framework components must run on various hardware architectures. Components are tailored to run from resource-constrained devices to off-the-shelf Virtual Machines hosted in the cloud. Particularly monitoring components should not interfere with running applications and cause minimal overhead. Additionally, the framework uses a container orchestration service to automate the deployment of components and execution of experiments. Lastly, an integrated monitoring strategy is necessary to provide tools for analyzing cluster management techniques. Based on the collected monitoring data, the framework offers features to analyze experiments. It supports commonly observed key performance indicators (i.e., resource usage, CPU consumption, scheduling behavior) out-of-the-box. We present and evaluate in Section IV-B use cases that include these indicators. To summarise, our end-to-end framework aims to support users in performing experiments from the deployment to analyzing them.

B. Galileo

Galileo [11] was created to develop and evaluate cluster management techniques using Symmetry, a custom cluster management system. Galileo's main tasks are to coordinate clients to generate requests as well as monitor and store monitoring data in a MySQL instance.

Based on the description of our view of a modern end-to-end experiment and analytics framework, *Galileo* requires adaptions and extensions. Specifically, we intend the integration into a modern container orchestration service such as Kubernetes. This allows us to deploy framework components into the cluster, reducing manual steps required to run experiments. Further, we provide a representative set of ready-to-deploy applications (i.e., AI inference functions) to the community for evaluation purposes and as examples for creating custom applications. Additionally, we add to the SQL-based data storage the time-series database InfluxDBv2[3]. The SQL storage stores experiment metadata (i.e., start, end) and data gathered during runtime is stored in InfluxDB. The fine-grained monitoring tool, *telemd*, has also been extended since its first release and now supports Kubernetes pods as well as the recently released Pressure Stall Information (PSI)[4]. PSI is available for CPU, I/O, and network. It shows the time some or all processes have been waiting for a specific resource. We believe using these new metrics can lead to many novel cluster management techniques.

[3]https://www.influxdata.com/

C. Experiments

The experimentation and analytic framework offer a streamlined approach for executing and measuring benchmarks. We consider two types of experiments in this paper: *scenario* and *profiling*.

Scenario experiments are complex scenarios to perform resource management actions (i.e., scaling). These experiments allow users to specify multiple application instances on nodes. Request generation is based on workload profiles that *Galileo* clients execute.

Profiling experiments are similar to *scenarios* but offer reduced functionality and concentrate on executing one type of application on one node. Profiling experiments aim to rapidly test new applications to estimate resource usage and performance across devices.

D. Applications

Our framework supports any containerized application which exposes an HTTP endpoint. Though, our primary focus while developing it was using OpenFaaS[5] based functions on different hardware. Therefore, we offer ready-to-use functions that support all common architectures (i.e,. amd64, arm64v8 and arm32v7). The functions use the OpenFaaS watchdog[6], which uses a Go-based proxy and calls an internal Python Flask web server to invoke the function. At the time of writing, our function repository[7] primarily consists of AI inference functions. For example, the functions can perform object detection (i.e., gun, human, mask), object classification, and pose estimation.

Custom applications require implementing methods to call the relevant service and a Kubernetes Pod factory for the container image. This guarantees flexibility concerning future applications and requires minimal coding efforts. Section III-C presents details about the individual projects.

E. Edge-cloud testbeds

To ensure the reproducibility of experiments, we require tools for setting up testbeds. However, our vision of an edge-cloud system consists of multiple heterogeneous interconnected clusters with varying and adverse network conditions. Consequently, we cannot rely on existing work on cloud testbeds characterized by relative homogeneity of resources

[4]https://www.kernel.org/doc/html/latest/accounting/psi.html

[5]https://docs.openfaas.com/

[6]https://github.com/openfaas/of-watchdog

[7]https://github.com/edgerun/galileo-experiments-functions

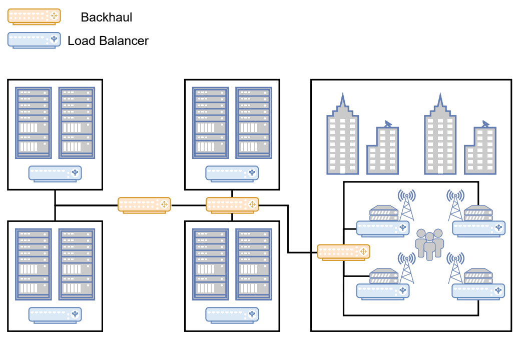

Fig. 1: Edge-cloud system consisting of multiple clusters

and static and optimal network conditions. Figure 1 depicts such an edge-cloud system with multiple clusters, where each is assigned a load balancer.

Cluster management techniques must consider user mobility since it is an important characteristic of edge-cloud systems [12]. For example, clients move around the city and connect to radio towers, which act as the first entry point. Each load balancer can redirect requests to other clusters. Internet backhauls inter-connect all clusters but introduce significant network latency [13]. Our architecture is comparable to works presented in [1] and [2], which evaluate cluster management techniques in geo-distributed systems using public cloud offerings. As mentioned, we use a container orchestration service to set up the cluster. Thus, we create deployment files that must be applied to the cluster once. This includes clients, monitoring agents as well as load balancers, enabling users to set up our framework on their testbed. Network latency is emulated via linux tools and mimics the behavior of geo-distributed edge-cloud clusters. Reproducibility is guaranteed through the usage of linux tools and an orchestration service.

III. System

This section introduces our experimentation framework by describing the system architecture, explaining the experiment workflow, highlighting important implementation details, and presenting the testbed used to demonstrate our framework.

A. Architecture

The architecture facilitates generating requests, hosting applications, distributing messages, and storing data. We use the lightweight edge-oriented K3s[8] Kubernetes distribution to host applications and clients. All nodes join the same cluster; while not intuitive at first, considering we want to create a testbed with multiple clusters, it avoids the overhead of setting up a cluster federation. We use labels to tag controllers, client nodes, and workers, which assign each node cluster. Each cluster has one controller node acting as a load balancer. WireGuard [14], a lean VPN, is used to form a homogeneous network and *tc*[9], the Linux network shaping

[8]https://k3s.io/
[9]https://man7.org/linux/man-pages/man8/tc.8.html

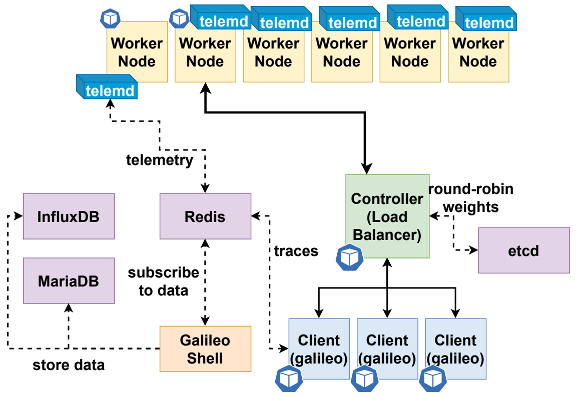

Fig. 2: A high level overview of the experiment setup

tool, to introduce latency between clusters. Section IV-C presents more details about the testbed setup. Figure 2 depicts the described system. Workers host applications, while client nodes run *Galileo* and wait for workload generation requests published via Redis. Clients send HTTP requests to the load balancer, which forwards them to application instances. The controller hosts the load balancer instance. We provide a Go-based load balancer (*go-load-balancer*) which implements a weighted round-robin based load balancing strategy. Though, users can choose to use any other L7-layer load balancer (i.e., Traefik[10]). Our experimentation framework sets initial weights for load balancers. Users can modify weights via etcd[11], a key-value storage used by Kubernetes to store state. Each load balancer instance watches the etcd instance for changes and updates its internal weights accordingly. The *galileo shell* is the component that initiates the experiment and subscribes to Redis, and stores data in InfluxDB and MariaDB for post-experiment analysis.

B. Experiment workflow

In the following, we explain how users can define and run experiments. Figure 3 splits the experiment into three phases: pre-experiment, runtime, and post-experiment. Users only have to interact during the pre and post-experiment phases, while our framework handles the system during the runtime phase. Users have to supply the following input: workload (i.e., a list of inter-arrival times), applications (i.e., container image), clients (i.e., implement a method to create the HTTP request), nodes (i.e., the testbed to perform experiments on). In the pre-experiment phase, users can generate workloads. The framework supports two types of workloads: parameterized and profile-based. The parameterized approach requires three arguments: n the number of requests, ia the inter-arrival time, and the number of clients. Clients are *asynchronous* and send n requests with a fixed inter-arrival time ia. Consequently, experiments will stop without waiting for responses. The

[10]https://traefik.io/
[11]https://etcd.io/

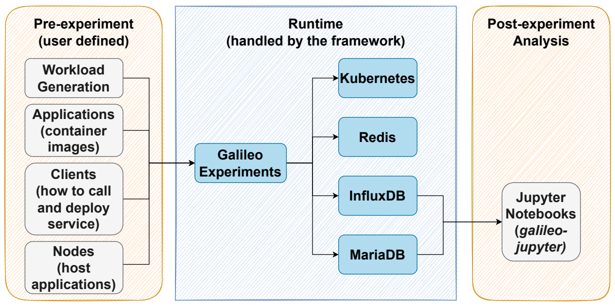
Pre-experiment (user defined)
Workload Generation
Applications (container images)
Clients (how to call and deploy service)
Nodes (host applications)
Runtime (handled by the framework)
Galileo Experiments
Kubernetes
Redis
InfluxDB
MariaDB
Post-experiment Analysis
Jupyter Notebooks (galileo-jupyter)

Fig. 3: Experiment workflow

profile-based variant expects an array that contains inter-arrival times, each array representing one client. For example, $[0.5, 1, 0.5]$ will send three requests: one after 0.5 seconds, another one after 1 second, and the last after another 0.5 seconds.

Profiling and *scenario* experiments, as described in Section III-B, have different parameters. The former takes both types of workload, while the latter only supports the profile-based approach. Additionally, profiling experiments take the application container image, the number of application instances, the host to profile, and in which cluster clients generate requests. In scenario experiments, users create a mapping between nodes and container images, including the number of instances. Further, scenario experiments can start clients in different clusters, making it possible to define complex scenarios (i.e., migrating users from one to another cluster).

After, users start the framework and enter the runtime. The framework sets up load balancers, spawns applications, configures clients, and starts the *galileo shell* to record all telemetry and trace data. Finally, in the post-experiment phase, users can analyze the results using *galileo-jupyter*. We categorize

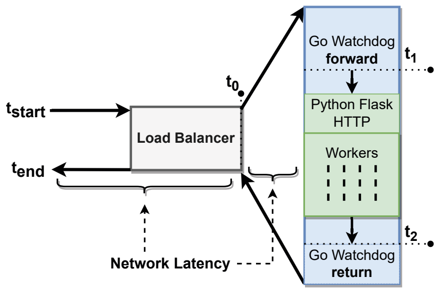

Fig. 4: A detailed look into traces

analytics data into three main categories: metadata, traces, and resource usage. Metadata consists of an experiment ID, the creator, start and end. Additionally, the framework also saves information about all nodes. This includes information detailed in the *telemd* repository and Kubernetes labels of each node and their allocatable resources. Figure 4 provides a detailed description of one trace. We record: the time the trace was created, the client sent it (t_{start}), it was forwarded from the load balancer ($t0$), it was received by the function t_1, returned t_2 and arrived back at the client t_{end}. Based on these timestamps, we can calculate and estimate further measures: round-trip-time (RTT, $t_{end} - t_{start}$), network latency between client and load balancer ($t_0 - t_1$), the network latency between load balancer and application instance ($t_1 - t_0$) and the execution time ($t_2 - t_1$). The execution time includes any queuing caused by the static number of worker threads the Python Flask server employs to handle requests. Functions can return the function execution time in the response body. Telemetry data consists of node-based resource usage (i.e., CPU usage) and per container (i.e., CPU time). We refer readers to look up the project's homepage for an exhaustive list of monitored resources[12].

In scenario experiments, users must deploy their own cluster management techniques. They can create and teardown Pods via the Kubernetes API, and the *telemd-kubernetes-adapter* publishes these events, as well as *telemd* will monitor newly created Pods. Further, if our *go-load-balancer* is used, users must update weights manually to include new or deleted applications.

C. Implementation

Our projects mainly use Python. Only the load balancer, a Kubernetes adapter, and the monitoring agent are written in Go. All projects are open source and available in the Edge Run project. In the following, we want to give a high-level overview of the different repositories and their functionality.

Galileo does two things: start clients and record experiment data. It offers the essential working tools but is not automated. Users must deploy an experiment setup manually.

Galileo Experiments contains code that invokes *galileo* to start and record experiments. It also includes deployment files for necessary components and describes in detail which other services are required to start an experiment.

Galileo Experiments Extensions provides client implementations for our functions and serves as entrypoint for experiment execution.

Galileo Experiments Functions contains OpenFaaS based functions that can be deployed and benchmarked using clients implemented in the *Extensions* project.

Galileo Jupyter contains gateways that allow users to easily access data recorded during the experiments. Specifically the *K3sGateway* is implemented to analyze Galileo experiments. The *galileo-jupyter* project internally uses *galileo-db* to access the data storages. Both projects have been extended to support InfluxDB.

The *Request Generator* project provides functions to generate workload profiles that can be used in the experiments.

Go Load Balancer is a weighted round robin Load Balancer which watches etcd for weight changes. It sets HTTP headers to later analyze the path each request has taken to finally arrive at an application instance.

Telemd is used as a lightweight push-based fine-grained monitoring agent and supports resource monitoring on node

[12]https://github.com/edgerun/telemd/

and application level. Users can set the interval for each resource separately.

The framework automatically deploys the *Telemd Kubernetes Adapter* which watches Kubernetes Pods using the official Go Kubernetes client. Any Pod lifecycle event is reported via Redis and gets recorded during experiments. This allows users to analyze scale events and associate telemetry with Pods.

IV. Demonstration

This section describes possible use cases to motivate our framework and we evaluate in Section IV-B. Afterwards, we present our applications used to evaluate selected use cases to show the results of framework on our testbed.

A. Use cases

Profiling experiments can be used to gather performance measures and resource usage to help developers better understand application implementations and estimate how well a deployment might perform [7]. Especially in heterogeneous environments, with resource-constrained devices and modern hardware accelerators, it is essential to understand hardware and applications [15]. Another use case for a *profiling* experiment is the measurement of performance interference due to multi-tenancy [9]. The *scenario* experiments differ in configuration capabilities and let users define a range of initial application instances and their location. The framework does not offer any support for cluster management techniques, and users must start schedulers or scalers that interact with the cluster. Nevertheless, during the experiment, any changes to the clusters are recorded, meaning that users can perform cluster management actions during runtime and use our framework to analyze the experiments. For example, an analytics technique is to display when and where application instances have been created and shut down [1].

B. Applications

We are deploying an OpenFaaS-based function, which serves the Mobilenet neural network [16] using TFLite in CPU mode. TFLite is a version of Tensorflow tailored towards resource-constrained devices. The function performs object classification on base64-encoded images. The function, as well as the *galileo experiment* client application, are available on Github. You can find the code for the demonstration on Github[13] as well. We also include an evaluation of resource usage of *telemd*, our monitoring agent, the only experiment component running on end-devices (except for *kubelet*).

C. Testbed

Table I displays a selection of devices in the testbed. *NX*, *TX2* and *Nano* are all Nvidia Jetson devices[14]. There are three clusters in total: *IoT Box*, *Cloudlet* and *Cloud*. The edge-based clusters are derived from [17].

TABLE I: Testbed

Device	Arch	CPU	Memory	Cluster
1x AsRock	x86	8x Ryzen @ 2 GHz	33GB	IoT Box
1x RPI 4	arm32v7	4x Cortex @ 1.5 GHz	1 GB	IoT Box
1x NX	arm64v8	4x Cortex @ 2 Ghz	8 GB	IoT Box
1x TX2	arm64v8	4x Cortex @ 2 Ghz	8 GB	IoT Box
1x Nano	arm64v8	4x Cortex @ 1.4 GHz	4 GB	IoT Box
1x Xeon	x86	4x Xeon @ 4.6GHz	16 GB	Cloudlet
4x VM	x86	4x vCPU @ 2Ghz	8 GB	Cloud
2x NUC	x86	4x i5 @ 2.2 GHz	16 GB	*Clients*

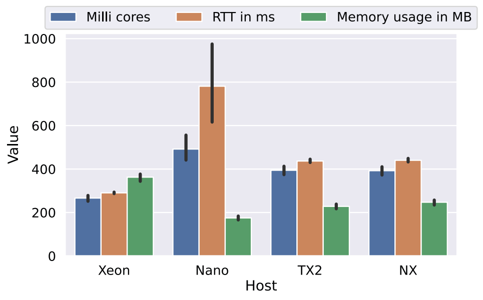

Fig. 5: Profiling results of the Mobilenet function

D. Experiments

1) Profiling experiment: The profiling experiment aims to evaluate performance and resource usage across devices. One client sends 100 requests with an inter-arrival time of one second. We execute the experiment on four devices: *Xeon*, *NX*, *TX2* and *Nano*. Figure 5 shows the milli cores used by the container, the RTT in ms, and the memory usage in megabytes for each device. The error bars indicate the 95th mean confidence interval. While *NX* and *TX2* have comparable performance, the *Xeon* node has the lowest RTT. The *Nano* performs worse, and outliers are as long as 6 seconds. A milli core value of 1000 equals one fully utilized CPU core. The *Nano* performs again the worst and has the highest CPU usage. Interestingly, the *Xeon* device has the highest memory usage, while the *Nano* has the lowest. It is out of scope to further discuss the results as we only intend to demonstrate possible use cases for our experimentation and analytics framework. The *galileo-jupyter* project offers many convenience functions (i.e,. `preprocessed_traces`, `preprocessed_telemetry`) that enable developers to plot this chart based on the results (i.e., with Jupyter Notebooks[15]).

2) Scenario: We now turn to the scenario experiment. Specifically, we start three clients spawning 60 requests with an inter-arrival time of 1 second. Two clients are situated in the *IoT-Box* cluster and one in the *Cloudlet* cluster. We also start a Python program that randomly scales up and down in an interval of 5 seconds. With a probability of 10% it does nothing, with 40% it starts one new function

[13]https://github.com/edgerun/galileo-experiments-tdis-2022
[14]https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/
[15]https://jupyter.org/

instance, and with 50% it tears one instance down. All load balancers distributed requests in a round-robin fashion for this experiment (i.e., from *Cloudlet* to *Cloud*). Figure 6 shows how many function replicas have been running across the cluster. This plot illustrates the behavior of cluster management techniques and analyzing it serves as an essential task during development. Further, our framework enables users to analyze where requests have been generated and where they have been processed. To sum up, our framework can perform experiments that allow requests to be processed across the edge-cloud continuum and offers functionality to analyze them. As before, *galileo-jupyter* exposes convenience functions to support users in developing and testing cluster management techniques (i.e., `get_replica_schedule_statistics`).

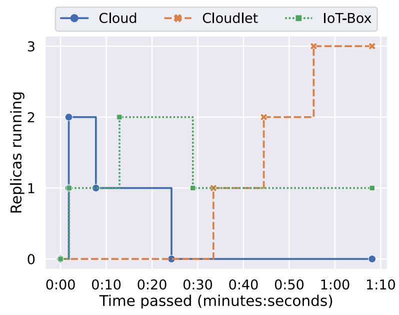

Fig. 6: Function replicas running across clusters

3) Telemd evaluation: To conclude this section, we show our monitoring agent's resource usage while it emits resource metrics every second. Figure 7 displays results recorded during a 100-second experiment. The figure shows the relative container CPU usage of *telemd*. 100% CPU usage means that the container (i.e., *telemd*) fully utilized all cores of the device. Figure 7 also shows the memory usage in megabytes. Both results show that resource usage is relatively low on all tested devices, ranging from a Raspberry Pi 4 to a Xeon-based PC. Users can set a different monitoring interval for each resource to reduce resource usage further.

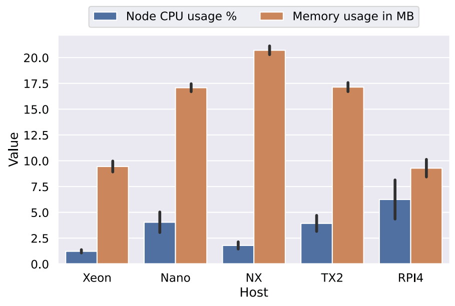

Fig. 7: Resource usage of *telemd*

V. Related Work

Works have been published that provide applications (i.e., functions) to evaluate systems but do not propose an end-to-end framework [18]. We can use these functions with our framework if they expose an HTTP endpoint. Grambow et al. [19] present a benchmarking framework for FaaS platforms. They focus on supporting public FaaS offerings while we present a framework for edge-cloud testbeds. Gao et al. [20] present LinkLab, a scalable IoT testbed that supports experimentation and remote development. In contrast to their work, our framework aims to support users in evaluating cluster management techniques and profiling applications, while theirs focuses on developing and evaluating IoT applications. Yang et al. [21] present EdgeTB, a hybrid testbed to evaluate distributed machine learning applications at the edge. EdgeTB combines emulated nodes with real ones to create a high-fidelity and large-scale evaluation. In contrast to our work, their framework is for AI training, while ours is flexible regarding applications. Das et al. [22] present a benchmark suite to evaluate commercial edge computing platforms. To summarize, many approaches exist that build benchmarks for FaaS or edge-cloud computing, but we specifically aim to support the development of cluster management techniques.

VI. Future Work

The testbed configuration (i.e., labeling the nodes) is not yet fully automatic and configuring network latency between clusters has to be done manually. For future work, we plan on extending *Ether*, Edge Run's topology synthesizer [17], to model the real-world testbed as code and perform configurations (i.e., WireGuard) as well as set the network latency between them. Another extension we are working on right now is adding power measurements and using them to train models that can predict energy consumption based on resource usage. Using our framework, we can fully automate and perform different experiments to gather enough data to train our models.

VII. Conclusion

Evaluating edge computing systems is challenging given both the nascence of the field, as well as the heterogeneity of infrastructure such systems operate on. Benchmarks and testbeds used to evaluate cluster management techniques are often tailored to the specific technique, making it difficult to reproduce them and therefore compare approaches. We extend *Galileo*, a system for distributed load testing that enables users to define and run reproducible benchmarks. *Galileo* lacks tools to operationalize experiments, users must manually deploy other tools to scale workers, instrument their testbed and collect data. We extend *Galileo* to use Kubernetes to deploy runtime components, and extend and create tools to offer an end-to-end experiment and analytics framework for edge-cloud clusters. Future work includes a code based approach to define testbeds and dynamically create different cluster setups with varying network latency.

Acknowledgment

We thank Alexander Knoll for supporting us in setting up the hardware infrastructure, and Jacob Palecek for the code contributions.

References

[1] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, vol. 159, pp. 161–174, 2020.

[2] M. A. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth, "mck8s: An orchestration platform for geo-distributed multi-cluster environments," in *2021 International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2021, pp. 1–10.

[3] B. Shayesteh, C. Fu, A. Ebrahimzadeh, and R. Glitho, "Auto-adaptive fault prediction system for edge cloud environments in the presence of concept drift," in *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2021, pp. 217–223.

[4] V. Kjorveziroski and S. Filiposka, "Kubernetes distributions for the edge: serverless performance evaluation," *The Journal of Supercomputing*, pp. 1–28, 2022.

[5] K. Alwasel, D. N. Jha, F. Habeeb, U. Demirbaga, O. Rana, T. Baker, S. Dustdar, M. Villari, P. James, E. Solaiman *et al.*, "Iotsim-osmosis: A framework for modeling and simulating iot applications over an edge-cloud continuum," *Journal of Systems Architecture*, vol. 116, p. 101956, 2021.

[6] R. Mayer, L. Graser, H. Gupta, E. Saurez, and U. Ramachandran, "Emufog: Extensible and scalable emulation of large-scale fog computing infrastructures," in *2017 IEEE Fog World Congress (FWC)*. IEEE, 2017, pp. 1–6.

[7] D. N. Jha, M. Nee, Z. Wen, A. Zomaya, and R. Ranjan, "Smartdbo: smart docker benchmarking orchestrator for web-application," in *The World Wide Web Conference*, 2019, pp. 3555–3559.

[8] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," *Future Generation Computer Systems*, vol. 114, pp. 259–271, 2021.

[9] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, and R. Ranjan, "A holistic evaluation of docker containers for interfering microservices," in *2018 IEEE International Conference on Services Computing (SCC)*. IEEE, 2018, pp. 33–40.

[10] T. Rausch, W. Hummer, and V. Muthusamy, "Pipesim: Trace-driven simulation of large-scale ai operations platforms," *arXiv preprint arXiv:2006.12587*, 2020.

[11] T. Rausch, P. Raith, P. Pillai, and S. Dustdar, "A system for operating energy-aware cloudlets," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 307–309.

[12] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 10, pp. 2333–2345, 2018.

[13] T. Braud, Z. Pengyuan, J. Kangasharju, and H. Pan, "Multipath computation offloading for mobile augmented reality," in *2020 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 2020, pp. 1–10.

[14] J. A. Donenfeld, "Wireguard: next generation kernel network tunnel." in *NDSS*, 2017, pp. 1–12.

[15] S. P. Baller, A. Jindal, M. Chadha, and M. Gerndt, "Deepedgebench: Benchmarking deep neural networks on edge devices," in *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2021, pp. 20–30.

[16] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[17] T. Rausch, C. Lachner, P. A. Frangoudis, P. Raith, and S. Dustdar, "Synthesizing plausible infrastructure configurations for evaluating edge computing systems," in *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.

[18] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 502–504.

[19] M. Grambow, T. Pfandzelter, L. Burchard, C. Schubert, M. Zhao, and D. Bermbach, "Befaas: An application-centric benchmarking framework for faas platforms," in *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2021, pp. 1–8.

[20] Y. Gao, J. Zhang, G. Guan, and W. Dong, "Linklab: A scalable and heterogeneous testbed for remotely developing and experimenting iot applications," in *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE, 2020, pp. 176–188.

[21] L. Yang, F. Wen, J. Cao, and Z. Wang, "Edgetb: A hybrid testbed for distributed machine learning at the edge with high fidelity," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 10, pp. 2540–2553, 2022.

[22] A. Das, S. Patterson, and M. Wittie, "Edgebench: Benchmarking edge computing platforms," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 2018, pp. 175–180.