WebAssembly at the Edge: Benchmarking a Serverless Platform for Private Edge Cloud Systems

Giuseppe De Palma and Saverio Giallorenzo, *Università di Bologna, 40126, Bologna, Italy, and INRIA, 06902, Sophia Antipolis Cedex, France*

Jacopo Mauro, *University of Southern Denmark, 5230, Odense, Denmark*

Matteo Trentin, *Università di Bologna, 40126, Bologna, Italy, INRIA, 06902, Sophia Antipolis Cedex, France, and University of Southern Denmark, 5230, Odense, Denmark*

Gianluigi Zavattaro, *Università di Bologna, Italy, and INRIA, 06902, Sophia Antipolis Cedex, France*

FunLess is a function-as-a-service (FaaS) platform tailored for private edge cloud systems. FunLess leverages WebAssembly as its runtime environment for performance, function isolation, and support for heterogeneous devices, crucial for extending the coverage of serverless computing to private edge cloud systems. We benchmark FunLess against three production-ready, widely adopted open source FaaS platforms—OpenFaaS, Fission, and Knative—under different deployment scenarios, characterized by the presence/absence of constrained-resource devices (Raspberry Pi 3B+) and the (in)accessibility of container orchestration technologies—Kubernetes. Our results confirm that FunLess is a suitable solution for FaaS private edge cloud systems since it achieves performance comparable to the considered FaaS alternatives while it is the only fully deployable alternative on constrained-resource devices.

Function as a service (FaaS), part of the serverless computing offering,[1] has become one of the state-of-the-art paradigms for distributed systems. In FaaS, programmers write and compose stateless functions, leaving to the platform the management of deployment and scaling. While emerged in the cloud, recent trends demonstrated the benefits and feasibility of applying FaaS to private and hybrid cloud scenarios,[2] including edge[3] and Internet-of-Things (IoT)[4] components. The main motivations for private edge cloud systems include reducing latency, increasing security and privacy, and improving bandwidth and usage of high-end devices.[4]

FunLess is a new open source serverless platform targeting private edge cloud systems.[a] The main advantage of FunLess over alternatives like OpenFaas, Knative, Fission, and OpenWhisk is that the former provides lightweight scalability, isolation, performance, and portability via the Erlang's BEAM Virtual Machine[5] and a WebAssembly[b] (Wasm) runtime for running functions. Thanks to these traits, users can run the whole FunLess distributed platform on resource-constrained edge devices; an option unachievable with the alternatives that use heavier runtimes (based on containers, e.g., Docker) and container-orchestration technologies (e.g., Kubernetes).

1089-7801 © 2024 IEEE
Digital Object Identifier 10.1109/MIC.2024.3513035
Date of publication 11 December 2024; date of current version 24 January 2025.

[a]FunLess' sources and documentation are publicly available resp.at https://github.com/funlessdev/funless and https://funless.dev/.

[b]https://webassembly.org/

The main novelty introduced by FunLess is that it can capture edge-only deployments, where the components of the platform reside on low-power edge devices, without having to rely on the cloud. One of the main use cases for edge-only serverless deployments regard systems that manage data which cannot leave the edge domain. Examples of such systems include e-Health, whose motto is "the data never leave the hospital," [6] smart factories, where real-time data from sensors and equipment can contain proprietary information about manufacturing processes or trade secrets that companies want to keep on-premises,[7] and remote-site production (e.g., offshore platforms), where limited connectivity prevents or substantially hinders accessing the cloud.

FunLess exploits Wasm's fast startup times and the small size of Wasm functions to mitigate cold starts[8] (delays in function execution due to the overhead of loading and initializing functions) with an efficient caching system. Moreover, since Wasm binaries can run on any platform that can execute a Wasm runtime, FunLess provides a consistent development and deployment experience across diverse private edge architectures. While FunLess dispenses the use of container orchestration, it is compatible with the latter, allowing users to choose the best solution for their case (e.g., if the system includes powerful nodes able to run Kubernetes).

To evaluate FunLess, we benchmark it against different deployment scenarios of typical cloud and edge FaaS workloads, contrasting it with production-ready FaaS platforms: OpenFaaS, Fission, and Knative.

Broadly, we find that FunLess is the only platform deployable on an edge-only cluster of constrained devices (Raspberry Pi 3B+)—the alternatives can run functions on edge devices, but need more powerful nodes to host their whole architecture. Indeed, FunLess's function artifacts are ca. 97% smaller than the alternatives, saving memory and bandwidth and FunLess' bare-metal modality (without Kubernetes) consumes the minimum amount of memory among the alternatives. Regarding heterogeneity, FunLess and Fission are the only platforms that support the seamless integration of different architectures—the others require architecture-specific function binaries.

Thus, FunLess is performance-wise comparable with the other platforms (that use binary native code) but the only fully deployable one on heterogeneous clusters of constrained edge devices.

PRELIMINARIES

Before presenting FunLess' architecture, we provide preliminary notions useful to contextualize our contribution: We introduce WebAssembly and summarize the selection process of the serverless platforms we compare against FunLess in our evaluation.

WebAssembly

The idea behind Wasm, a W3C standard since 2019, is to provide an assembly-like instruction set that can run efficiently within browsers.

Although Wasm's main target is browsers, initiatives like WebAssembly System Interface (WASI)[c] normed the implementation of Wasm runtimes to run Wasm code outside browsers with a set of application programming interfaces (APIs) for portable operating system interface capabilities, like file systems and networking.

Focusing on FaaS, Wasm provides a sandboxed runtime environment for functions akin to containers. However, one needs to build a container (for the same function) for each targeted architecture while the same Wasm binary can run on different architectures. Moreover, since they forgo prepackaged filesystems, Wasm binaries tend to be smaller than containers.

Alternative Open Source Serverless Platforms

Looking for alternatives to FunLess, the closest we found, targeting the edge cloud case, are Lucet,[d] Cloudflare Workers,[e] WOW,[9] and Lean OpenWhisk[10] (cf. Related Work). Unfortunately, we cannot include any of these options in our benchmarks. Indeed, Cloudflare Workers is a closed-source project while Lucet, WOW, and Lean OpenWhisk are open source projects. However, we could deploy none of them since they are no longer maintained and require deprecated dependencies or container images.

Given the previous results, we broaden our scope to widely adopted open source serverless solutions not necessarily adapted for the edge cloud case.

To select the candidates, we searched GitHub for the keyword "faas" (which, at the time of writing, returned 3.9k matches), and we followed four inclusion criteria for the selection: production-ready (used in industry, verified by looking at the commercial

[c]https://wasi.dev/
[d]https://github.com/bytecodealliance/lucet
[e]https://developers.cloudflare.com/workers/reference/how-workers-works/

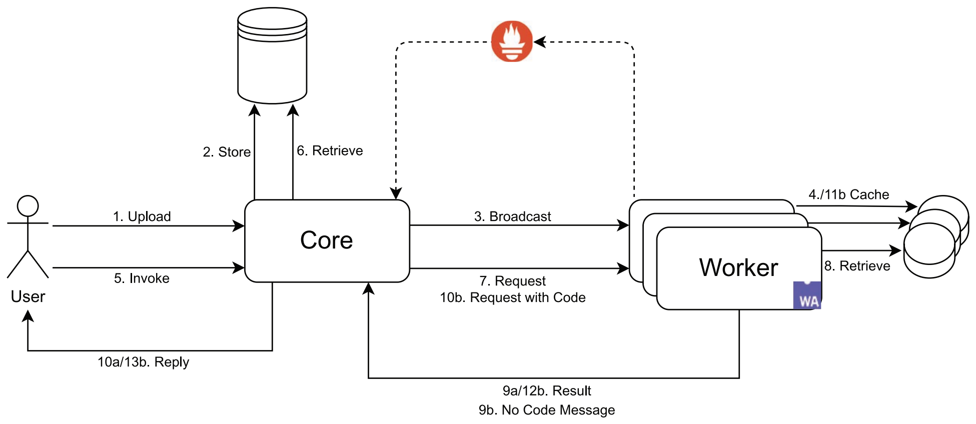

FIGURE 1. FunLess' architecture and function lifecycle schema.

testimonials found on the project's webpages), popular (above 5k stars on GitHub), actively developed (commits within the last quarter and with at least 100 contributors), and able to run on both AMD64 and ARM hardware (i.e., the most common hardware found in cloud and edge devices). We ranked the results by popularity (GitHub stars) and selected the first three. The selected platforms, in ranking order, are OpenFaaS (24k+ stars), Fission (8k+ stars), and Knative (5k+ stars).[f] Notably, the previous selection excludes Apache OpenWhisk, a popular (6k+ stars) serverless platform, because it lacks ARM container images for its main components (Controller and Invoker).

FUNLESS' ARCHITECTURE

The main design principle behind FunLess is the simplicity of both function development and platform deployment. FunLess's components are written in Elixir,[11] a functional, high-level programming language that runs on the BEAM virtual machine[5] (used by the Erlang language), which helps to remove the need for deployment orchestrators, reducing the weight on the nodes running the architecture.

We show FunLess' architecture and the flow of function creation and execution (commented later) in Figure 1. FunLess consists of mainly two components: the *Core* and the *Worker*. The Core acts as a user-facing API to 1) create, fetch, update, and delete functions and 2) schedule functions on workers. The Worker is the component deployed on every node tasked to run the functions; in the remainder, we refer to these nodes as Workers. Besides Core and Workers, FunLess includes a *Postgres* database to store functions and metadata and *Prometheus* to manage the metrics of the platform.[g]

Core

The Core controls the platform, exposing an HTTP REST API for user interaction, handling authentication and authorization, and managing functions' lifecycle and invocations. The Core can automatically discover Workers within the same network using Elixir's libcluster[h] library, employing the Multicast UDP Gossip algorithm for bare-metal deployments and Kubernetes' service discovery for containerized environments. Users can also manually connect Workers from other networks through simple messages, leveraging BEAM's built-in node connection capabilities.

Functionality-wise, users create functions by compiling source code to Wasm and uploading the binary to the Core, which stores it in the database with a name. Users can group functions in modules and specify memory requirements for function execution.

Upon receiving a function creation request (as shown in Figure 1, step 1. Upload), the Core stores the binary in the database (2. Store) and notifies the Workers (3. Broadcast) to cache a local copy (4. Cache), to reduce cold-start overheads. The components communicate via BEAM's distributed inter-process messaging system.

When a function invocation reaches the Core (5. Invoke), it retrieves it (if any) from the database (6. Retrieve). Using the latest metrics, the *Core* selects a Worker with enough memory to execute the function

[f] Resp.at https://github.com/OpenFaaS/faas, https://github.com/fission/fission, and https://github.com/knative.

[g] Resp.at https://www.postgresql.org/ and https://prometheus.io.

[h] https://hexdocs.pm/libcluster/readme.html

(7. Request)—returning an error if it finds none. Once it selects the Worker, the Core issues the execution of the function therein, waiting to receive the result back, which it relays to the user (10a/13b. Reply).

Worker

The Worker runs functions via Wasmtime, a WASI-compliant, security-oriented runtime for Wasm.

When a Worker receives the request to run a function (7. Request), it checks its cache for the function's binary (8. Retrieve). If it finds the binary, it runs the function and returns the result to the Core (9a. Result). Otherwise, the Worker informs the Core (9b. No Code Message), which sends the code (10b. Request with Code)—the one the Core fetched at step 8.—for caching (11b. Cache) and execution (12b. Result).

The previous logic supports efficient function fetching and execution with respect to containers, thanks to the small size of Wasm binaries compared to the larger/heavier container images. For caching and eviction, Workers have a configurable cache memory threshold.

EMPIRICAL EVALUATION METHODOLOGY

Test Infrastructure

We run all benchmarks on the following infrastructure. As edge devices, we use two Raspberry Pi 3B+ (1.4-GHz quad-core CPU and 1 GB of RAM), both with Debian 12 (Bookworm) ARM64 OS. For the cloud instances, we use Terraform to provision the virtual machines (VMs) and Ansible to install and configure the platforms.[i] Specifically, we provision up to five VMs in the Google Cloud Platform cloud (Europe West region with Ubuntu 22.04) and set up a virtual private network, to which we add the two nodes from the private edge (the Raspberry Pi devices). One e2-medium cloud node (2 vCPUs, 4-GB memory) completely dedicated to the Kubernetes control plane. In the cloud, we deploy the Core on an e2-medium node while the Workers are on n1-standard-1 (1 vCPU, 3.75 GB memory) VMs.

Deployment Configurations

We test four deployment configurations of the platforms as follows:

- *Edge-only*: Only edge devices, one hosts the core/controller of the platform and one acts as a worker, without Kubernetes.
- *Cloud-bare-edge*: The core/controller of the platform is in a cloud node and the two edge devices act as workers, without Kubernetes.
- *Cloud-edge*: Same as *cloud-bare-edge*, with Kubernetes.
- *Cloud-only*: Only cloud nodes, one hosts the core/controller and three act as workers, with Kubernetes.

Benchmarks

For all configurations, we collect the latencies of all platforms using the same set of benchmarks, drawn from the Serverless Benchmark Suite (SeBS),[12] including an additional compute-intensive benchmark (matrix multiplication), inspired by Gackstatter et al.[9] We measure memory footprints via a simple "hello world" function (described later).

The functions (1–3 from SeBS) are written in Go and JavaScript (JS) since these are the only officially supported languages by all the platforms, and include the following:

1) *Sleep* (JS), waits 3 s and returns a fixed response ("Slept for 3 seconds"). This benchmark tests a platform's capability of handling multiple functions running for several seconds and its requests queuing-management process.
2) *Network-benchmark* (Go), sends 16 HTTP requests with a timestamp and uploads this information to a cloud bucket. This test tracks how long each HTTP request takes to complete.
3) *Server-reply* (Go), sends a message to a server and waits for a reply, measuring the performance of the network stack and the latency of the platform with respect to the functions complete execution.
4) *MatrixMult* (JS), multiplies two 10^2 square matrices and returns its result. It measures the performance of handling compute-intensive functions.

For each platform, we define and build the functions following the approach suggested by their respective documentation. For FunLess, we compile JS using a customized javy[j] variant and Go using TinyGo.[k] Both compilers create a binary with a language-specific wrapper that performs input and output parsing at function invocation, simplifying the interaction with the Worker component (for JS, via javy's customization).

[i] Resp.at https://www.terraform.io/ and https://www.ansible.com/.

[j] https://github.com/bytecodealliance/javy/

[k] https://tinygo.org/

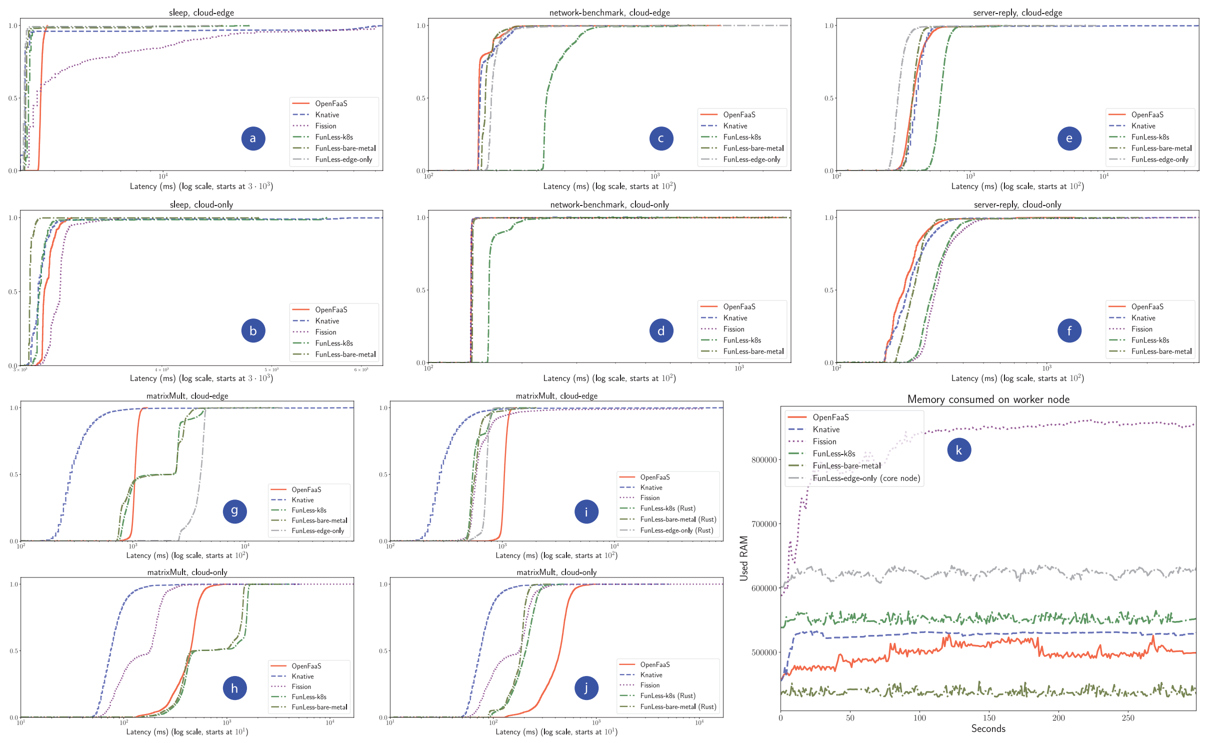

FIGURE 2. Plots ⓐ, ⓒ, ⓔ, and ⓖ show the cumulative distribution of the latencies under the cloud-edge configuration of resp. the sleep, network-benchmark, server-reply and matrixMult benchmarks; ⓑ, ⓓ, ⓕ, and ⓗ show the resp. performance under the cloud-only configuration—plots ⓘ and ⓙ show the matrixMult benchmarks but using a Rust variant of the function for FunLess (not used for comparison). Plot ⓚ shows the memory consumption of the hellojs benchmark on a Raspberry Pi 3B+.

We invoke the functions in parallel using JMeter,[i] measuring the latency between request delivery to the platform and the reception of the response—for network-benchmark, we perform sequential invocations to avoid issues from parallel bucket accesses.

Given the stable pattern of sleep, we use four parallel threads each sending 25 sequential requests (100 in total). For server-reply and matrixMult, we run four parallel threads, each sending 200 sequential requests (800 in total). For network-benchmark, we issue 50 sequential requests, each sending 16 HTTPS requests (800 in total). We repeat each benchmark five times.

We also track the memory usage on the edge devices by invoking a simple function. Namely, we write a "hello world" JS function, dubbed *hellojs*, which parses the input parameters and returns a string. We invoke the function continuously with four parallel threads for 5 min (i.e., each thread issues the next request after it receives the response from the previous one), keeping a sampling rate of one per second to track the memory used by the edge devices. We have one node (a Raspberry Pi) run the functions while the cloud nodes run the other components (e.g., schedulers, databases, web servers, and so on). For the edge-only deployment of FunLess, we measure the Core's memory usage on a second Raspberry Pi.

[i]https://jmeter.apache.org/

RESULTS

We plot the results of the benchmarks in Figure 2.

FunLess is the only platform able to run the edge-only and cloud-bare-edge configurations since the other platforms all require Kubernetes. For compactness, we report these results in the cloud-edge plots, resp. with the lines FunLess-edge-only and FunLess-bare-metal; the other line for FunLess in the plots is FunLess-k8s, which corresponds to Funless deployed using Kubernetes. As a reference point, the average latency between the edge devices and the cloud nodes is 31.5 ms, with a standard deviation of 6.3.

We start from the plots ⓐ and ⓑ relative to sleep. In the cloud-edge case, FunLess (the different modalities show small variations) and Knative are the best-performing platforms, although Knative has a few long-running outliers. Then, we find OpenFaaS with a narrow distribution but generally much slower than FunLess and Knative. The worst-performing platform is Fission, which presents a few good data points (closer to the origin), followed by a widely spread distribution of slow instances. In the cloud-edge case, the performance differences among the platforms narrow down. We attribute this behavior to OpenFaaS and Fission needing more powerful machines to run properly. In this configuration, FunLess-bare-metal has the best performance, followed by its Kubernetes variant and Knative. While the differences are small, this result indicates the higher load exerted by Kubernetes on the infrastructure.

Looking at plots ⓒ and ⓓ of network-benchmark, in both cloud-edge and cloud-only configurations, all platforms except FunLess perform similarly, with Knative and OpenFaaS scoring the best results on average. Note that the plot line for Fission is missing in the cloud-edge case, due to memory issues that prevent the platform from running this benchmark. FunLess-k8s is the worst-performing platform, while its bare-metal variant aligns with the alternatives. We attribute this result to the interplay overhead between Kubernetes' network stack and the FunLess' Wasm runtime (Wasmtime 12.0.1), which does not support native HTTP requests. Indeed, FunLess' Workers implement auxiliary operations that the Wasm functions can invoke to issue an HTTP request and get a response. These operations require Wasm to yield control to the BEAM, which handles the HTTP request-response procedure. The phenomenon is further exacerbated (higher latencies) by Docker's and Kubernetes' network layers in the related configurations.

Plots ⓔ and ⓕ of server-reply shows similar results, where FunLess-k8s performs worse than most of the alternatives, although Fission is the worst-performing one in the cloud-only configuration.

Plots ⓖ and ⓗ of matrixmult also report the performance of FunLess-edge-only in the cloud-edge plots; however, we also report ⓘ and ⓙ for this benchmark because we run it for FunLess with an alternative implementation of the function in Rust to investigate higher-than-expected latencies for FunLess. Indeed, looking at ⓖ and ⓗ, we notice that FunLess is the worst-performing platform. In particular, we note the "step" in FunLess' plots, which corresponds to almost half of the requests having a higher latency. Since we observe this phenomenon for both Funless-k8s and Funless-bare-edge, we can discard the hypothesis that the issue comes from the usage of Docker/Kubernetes. To assess whether this problem derives from the JS/Wasm runtime or some peculiarity of the platform, we run the Rust version of *matrixMult*, plotted in ⓘ and ⓙ. We stress that we do not use plots ⓘ and ⓙ to compare FunLess against the other platforms—it would be unfair since the other platforms cannot run that Rust function—but only to investigate if the performance issue is linked with the usage of JS. The results in ⓘ and ⓙ confirm our hypothesis (overhead due to the current FunLess' JavaScript runtime implementation) since the Rust variant obtains smaller, more consistent latencies.

Summarizing the results from the four benchmarks, FunLess generally has performance comparable with the other platforms, especially when deployed at the edge, without the support/weight of containerization technologies. Besides the results for the cloud-edge and cloud-only configurations, we underline that only FunLess can run in an edge-only configuration.

Next, we report in plot ⓚ of Figure 2 the memory consumption of the workers of the considered platforms (hence, not the core/controller) on the edge devices, including the memory required by the operating system (ca. 300 MB). For compactness and to provide a comprehensive view of the memory consumption profile of FunLess, we plotted in ⓚ also the consumption of FunLess' Core in managing the invocations of the hellojs benchmark, labeling the plot line FunLess-edge-only (core node). Since these results focus on just one component (the core/controller) of the platform, we comment on this plot line at the end of the section to avoid mixing its description with the ones comparing FunLess against the alternatives.

From the results, Fission requires the highest amount of memory—a consequence of the container pool used by the platform to reduce cold starts. While FunLess-k8s is the second-highest for memory occupancy (due to the stacking of the BEAM, Docker, and Kubernetes runtimes), FunLess-bare-edge requires the least amount of memory out of all the platforms (on average ca. 438 MB). Intuitively, this configuration can reach such a low memory footprint because it omits the overhead due to containers and container orchestration. While FunLess allows one to deploy the platform without the support of containers and container orchestrators, all the considered alternatives heavily rely on the latter, making it unfeasible to avoid their usage and prevent the overhead they generate. On the contrary, the deployment flexibility afforded by FunLess allows one to have a functioning Worker running with minimal overhead (e.g., that of the underlying operating system).

Looking at the plot line of FunLess' Core component, labeled FunLess-edge-only (core node) in ⓚ, the Core uses around 620 MB, including the operating system, the database (Postgres), the monitoring service

(Prometheus), and Docker, for an additional memory overhead of ca. 450 MB. One could further reduce FunLess' memory footprint by deploying the whole stack without Docker, but the platform makes it feasible to afford containers for the edge setting.

We close our comparison by contrasting the size of the function artifacts of the four benchmarks under the considered alternatives. Specifically, for both KNative and OpenFaaS we measure the size of the containerized functions they use for function deployment, which respectively average to 47.94 MB (26.84 stdev) and 20.63 MB (11.41 stdev). Since Fission injects functions into "environment" containers for their execution at runtime, we do not have function artifacts, and we take the size of these containers as a lower bound, with average 33.64 MB (1.72 stdev).[m] For FunLess, we measure the size of the Wasm binaries, with average 0.94 MB (0.07 stdev). Overall, Funless function artifacts are 97% smaller than the alternatives, which translate into a smaller footprint on memory and bandwidth.

RELATED WORK

Cassel et al.[13] highlight that 86% of serverless IoT/edge platforms rely on containers while only 2%–3% use alternatives, like FunLess does with Wasm.

We are not aware of other platforms like FunLess, where both the controller and the workers can run on resource-constrained edge nodes. However, other FaaS platforms share traits with FunLess.

Hall and Ramachandran[14] propose one of the first FaaS platforms that run Wasm functions. They use the V8 JavaScript engine for execution and isolation but observe a conspicuous toll on performance.

Gadepalli et al.[15] use Wasm to run and sandbox serverless functions, but their platform has limited portability for cloud/edge scenarios than FunLess since their deployment is single-host (the whole platform runs on one node) and does not support WASI.

WOW, an Apache OpenWhisk variant with a Wasm runtime by Gackstatter et al.,[9] targets edge computing but, contrary to FunLess, it requires the full deployment of the OpenWhisk platform, precluding the installation of the controller on constrained devices.

Regarding commercial solutions, both Lucet and Cloudflare Workers target the edge case and run Wasm functions. Lucet's project reached end-of-life. Cloudflare's platform is a closed-source solution that can run Wasm functions and supports WASI.

[m]For consistency, we consider AMD64 containers, which basically have the same size of their ARM64 counterparts.

CONCLUSION

We benchmark FunLess, a Wasm-powered serverless platform tailored for private edge cloud systems, against three alternatives from production-ready, widely adopted open source FaaS platforms—OpenFaaS, Fission, and Knative—and run representative cloud and edge FaaS benchmarks.

The results confirm that FunLess is a viable solution for FaaS private edge cloud systems, outperforming the considered alternatives in terms of memory footprint and support for heterogeneous devices.

Future work includes integrating into FunLess new versions of Wasmtime, e.g., providing native support for HTTP and the WASI runtime. Indeed, many current Wasm runtimes miss features like interface types, networking support in WASI multithreading, atomics, and garbage collectors. Besides Wasmtime, other projects are developing new, optimized, and extended Wasm runtimes, which FunLess can leverage to increase its performance and adapt to different application contexts. As an example, we conjecture that, by using a Wasm runtime that natively supports HTTP requests, FunLess will perform better in the network benchmark. Similarly, the support for garbage collection can improve JavaScript runtimes and enhance the performance of the matrix multiplication benchmark. Additionally, we will investigate whether these new approaches also impact the data interchange between the runtimes and the Wasm functions, as this is another critical factor in improving overall performance.

REFERENCES

1. E. Jonas et al., "Cloud programming simplified: A Berkeley view on serverless computing," 2019, *arXiv:1902.03383*.
2. P. Castro, V. Isahagian, V. Muthusamy, and A. Slominski, *Serverless Computing: Principles and Paradigms*, Cham, Switzerland: Springer-Verlag, 2023.
3. L. Baresi and D. F. Mendonca, "Towards a serverless platform for edge computing," in *Proc. IEEE Int. Conf. Fog Comput. (ICFC)*, Prague, Czech Republic, 2019, pp. 1–10, doi: 10.1109/ICFC.2019.00008.
4. H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Comput. Surv*, vol. 54, no. 11s, pp. 1–32, 2022, doi: 10.1145/3510611.
5. E. Stenman, "Beam: A virtual machine for handling millions of messages per second (invited talk)," in *Proc. 10th ACM SIGPLAN Int. Workshop Virtual Mach. Intermediate Lang. (VMIL)*, New York, NY, USA: Association for Computing Machinery, 2018, p. 4.

6. N. Rose, “The human brain project: Social and ethical challenges,” *Neuron,* vol. 82, no. 6, pp. 1212–1215, 2014, doi: 10.1016/j.neuron.2014.06.001.
7. M. Soori, B. Arezoo, and R. Dastres, “Internet of things for smart factories in industry 4.0, a review,” *Internet Things Cyber-Physical Syst.*, vol. 3, Apr. 2023, pp. 192–204, doi: 10.1016/j.iotcps.2023.04.006.
8. P. Vahidinia, B. J. Farahani, and F. S. Aliee, “Cold start in serverless computing: Current trends and mitigation strategies,” in *Proc. Int. Conf. Omni-layer Intell. Syst. (COINS)*, Barcelona, Spain, 2020, pp. 1–7, doi: 10.1109/COINS49042.2020.9191377.
9. P. Gackstatter, P. A. Frangoudis, and S. Dustdar, “Pushing serverless to the edge with web-assembly runtimes,” in *Proc. 22nd IEEE Int. Symp. Cluster, Cloud Internet Comput. (CCGrid)*, Taormina, Italy, 2022, pp. 140–149, doi: 10.1109/CCGrid54584.2022.00023.
10. A. Tzenetopoulos et al., *FaaS and Curious: Performance Implications of Serverless Functions on Edge Computing Platforms in ISC Workshops*. Berlin, Germany: Springer-Verlag, 2021.
11. S. Juric, *Elixir in Action*, Shelter Island, NY, USA: Manning, 2024.
12. M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, *SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing in Middleware*. New York, NY, USA: Association for Computing Machinery, 2021.
13. G. A. S. Cassel et al., “Serverless computing for Internet of Things: A systematic literature review,” *Future Gener. Comput. Syst.*, vol. 128, Mar. 2022, pp. 299–316, doi: 10.1016/j.future.2021.10.020.
14. A. Hall, and U. Ramachandran, “An execution model for serverless functions at the edge,” in *Proc. Int. Conf. Internet Things Des. Implementation (IoTDI)*, New York, NY, USA: Association for Computing Machinery, 2019, pp. 225–236, doi: 10.1145/3302505.3310084.
15. P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, “Sledge: A serverless-first, light-weight Wasm runtime for the edge” in *Proc. 21st Int. Middleware Conf.* ACM, 2020, 265–245.

GIUSEPPE DE PALMA is a Ph.D. student Dipartimento di Informatica- Scienza e Ingegneria, Università di Bologna, 40126, Bologna, Italy, and a member of the Olas team, INRIA, 06902, Sophia Antipolis Cedex, France. His research interests include software engineering, cloud and serverless computing and programming languages. De Palma received his M.Sc. degree in computer science. Contact him at giuseppe.depalma2@unibo.it.

SAVERIO GIALLORENZO is an assistant professor in the Dipartimento di Informatica-Scienza e Ingegneria, Università di Bologna, 40126, Bologna, Italy, and a member of the INRIA OLAS Team, INRIA, 06902, Sophia Antipolis Cedex, France. His research interests include programming languages, software engineering, and cybersecurity. Giallorenzo received his Ph.D. degree in computer science from Università di Bologna. Contact him at saverio.giallorenzo2@unibo.it.

JACOPO MAURO is a professor in the Department of Mathematics and Computer Science, University of Southern Denmark, 5230, Odense, Denmark. His research interest include cloud computing, cybersecurity, DevOps, formal methods, and optimization fields. Mauro received his Ph.D. degree in computer science from the University of Bologna. Contact him at mauro@imada.sdu.dk.

MATTEO TRENTIN is a Ph.D. student at Universitá di Bologna, Italy, under cotutelle with the Department of Mathematics and Computer Science, University of Southern Denmark, 5230, Odense, Denmark. His research interests include cloud and serverless computing, programming languages, artificial intelligence, and formal methods. Trentin received his M.Sc. degree in computer science from the University of Bologna. Contact him at matteo.trentin2@unibo.it.

GIANLUIGI ZAVATTARO is a professor Dipartimento di Informatica-Scienza e Ingegneria, Università di Bologna, 40126, Bologna, Italy, and a member of the INRIA OLAS team, INRIA, 06902, Sophia Antipolis, France. His research interests include concurrency theory, formal methods, service-oriented, and cloud computing. Zavattaro received his Ph.D. degree in computer science. Contact him at gianluigi.zavattaro@unibo.it.