FunLess: Functions-as-a-Service for Private Edge Cloud Systems

Giuseppe De Palma*†, Saverio Giallorenzo*†, Jacopo Mauro‡, Matteo Trentin*†‡ Gianluigi Zavattaro*†
* *Alma Mater Studiorum - Università di Bologna, Italy*
† *OLAS team, INRIA, Sophia-Antipolis, France*
‡ *University of Southern Denmark, Denmark*

***Abstract*—Serverless computing has extended its reach to encompass private edge cloud systems, aiming to enhance latency, security, and privacy while optimising resource usage. However, this extension comes with challenges such as running platforms and functions on disparate and resource-constrained devices. To respond to the challenges, we present FunLess, a Function-as-a-Service (FaaS) platform tailored for private edge cloud systems. Unlike conventional solutions relying on container technologies for function invocation, FunLess leverages WebAssembly (Wasm) as its runtime environment. This choice offers several advantages, including inherent security and isolation mechanisms crucial for data integrity and confidentiality, portability and consistent development and deployment, and a reduced memory footprint that allows functions to run on constrained edge devices.**

***Index Terms*—Private Edge Cloud Systems, Serverless, Function-as-a-Service, WebAssembly**

I. Introduction

The advent of serverless computing [1] introduced a paradigmatic shift in the development of distributed systems, called Function-as-a-Service (FaaS). In FaaS, programmers write and compose stateless functions, leaving to the serverless platform the management of deployment and scaling. FaaS was first proposed as a deployment modality for cloud architectures [1] that pushed to the extreme the per-usage model of cloud computing, letting users pay only for the computing resources used at each function invocation.

Private Edge Cloud FaaS. While public clouds are the birthplace of serverless computing, recent industrial and academic proposals demonstrated the desirability, benefits and feasibility of moving FaaS outside public clouds. These solutions are tailored for private, public, and mixed (where the infrastructure includes parts from public and private) cloud scenarios [2] and include edge [3] and Internet-of-Things [4] components. From the industrial point of view, several FaaS platforms are designed for edge computing (e.g., AWS Greengrass[1], Cloudfare Workers[2]).

This work has been partially supported by the research project FREEDA (CUP: I53D23003550006) funded by the framework PRIN 2022 (MUR, Italy), the French ANR project SmartCloud ANR-23-CE25-0012, and from ICSC – Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing, funded by European Union – NextGenerationEU.

[1]https://aws.amazon.com/greengrass/.

[2]https://www.cloudflare.com/en-gb/learning/serverless/glossary/serverless-and-cloudflare-workers/.

In contrast to public edge-cloud computing solutions, *private edge cloud systems* have the benefit of further reducing latency, increasing security and privacy, and improving bandwidth and usage of high-end devices [4]. More precisely, private edge cloud systems are small-scale cloud data centres in a local physical area, such as a house, an office, a factory, or a small geographic area, where mobile devices, such as drones, mobile robots, smartphones and fixed devices, such as sensors/actuators, workstations, and servers are interconnected through single or multiple local area networks.

In this paper, we address the challenge of supporting FaaS in private edge cloud systems. Off-the-shelf solutions to this challenge consist of deploying popular open-source FaaS platforms (e.g., OpenFaas, Knative, Fission, OpenWhisk) on top of container orchestration technologies (e.g., Kubernetes). However, these technologies, which usually rely on containers and container orchestration solutions, entail performance and resource overheads which can create issues on devices with constrained resources—they might not have enough memory to host containers or computational power to effectively run functions, especially in low-latency application contexts.

These problems motivated researchers and practitioners to consider alternatives and propose runtimes that provide the isolation and parallel execution of existing FaaS platforms yet mediate the heavy toll of the mentioned more complex runtimes. Examples of these proposals include using virtual machines like that of Java [5] and Python [6] or embedding functions in unikernels [7]. Unfortunately, while these solutions achieve the goal of reducing the overhead of containers, they respectively miss fundamental features. Java/Python VMs do not provide high-performing runtimes [8] and properly isolate functions (e.g., exposing the users to security risks). Unikernels are still a niche technology whose usage requires specific engineering knowledge (e.g., to define the minimal OS stack needed to run high-level functions).

A promising alternative is WebAssembly[3] (Wasm) for lightweight FaaS environments [9] (introduced in more detail in Section II). Indeed, Wasm comes with a stack-based virtual machine designed for running programs in a sandbox environment with performance close to native code and fast load times. Wasm proved to be a valid candidate for FaaS,

[3]https://webassembly.org/.

providing lightweight sandboxing at the edge with both small latencies and startup times [10], [11]—recently, providers like Cloudflare proposed closed-source solutions based on Wasm[4].

FunLess. Building on these results, we propose FunLess, a FaaS platform designed for (mixed) edge-cloud scenarios. FunLess uses Wasm to run functions, providing many pros:

- *Security.* Wasm's inherent security and isolation mechanisms make it well-suited for scenarios where data integrity and confidentiality are critical.
- *Memory and CPU footprint.* FunLess does not require a container runtime (e.g., Docker) and orchestrator (e.g., Kubernetes). Hence, the "bare-metal" deployment of FunLess frees resources essential for running functions on memory-constrained or low-power edge devices.
- *Cold starts.* FunLess leverages Wasm to mitigate the problem of cold starts [12], i.e., delays in function execution due to the overhead of loading and initialising functions—an issue that constrained-resource edge devices can accentuate. Cold-start mitigations usually rely on caching or keeping "warm" function instances. However, the size of containers can make these solutions unfeasible on constrained-resource devices. FunLess's use of Wasm minimise the cost of function caching (and even fetch-and-load roundtrips), making cold-start mitigations more affordable. Moreover, Wasm runtimes provide fast startup times (Wasm's main use case is in-browser execution, where responsiveness is crucial), allowing FunLess to achieve small cold-start overheads.
- *Consistent function development and deployment environment.* Since Wasm abstracts away the hardware and environment it runs within, FunLess provides a consistent development and deployment experience across the diverse private edge architectures, offering a built-in solution for the challenges of variability in hardware and software environments of private edge-cloud scenarios. Similarly to Java bytecode, Wasm binaries can run on any platform that can execute a (dedicated) Wasm runtime. As illustrated in Section III, the developers only need to write once their functions[5], compile them into Wasm binaries, and load them into the platform. FunLess handles the task of running them on the possible diverse devices and architectures of the given cloud/edge infrastructure.
- *Simple and flexible platform deployments.* FunLess can use existing containerisation solutions (e.g., Kubernetes) to streamline and ease its deployment. When container orchestration technologies are not affordable/available, users can install FunLess by running a *Core* component (with metrics and storage services, e.g., resp. Prometheus and Postgres) on a node and a *Worker* component on the nodes tasked to run the functions (cf. Section III). This flexibility derives from WebAssembly (the binaries do not need an ulterior container for their isolation), and FunLess' communication mechanism between nodes.

In the following, in Section II, we present WebAssembly in more detail. We detail FunLess' architecture in Section III and comment on the relevant traits that distinguish our proposal from alternative FaaS solutions in Section IV. We conclude and draw future work directions in Section V.

II. WebAssembly

We dedicate this section to providing the preliminary notions useful to contextualise our contribution. Specifically, we introduce WebAssembly—the technology underpinning the FunLess execution runtime (cf. Section III).

The WebAssembly [13] technology, Wasm for short, is a W3C standard since 2019, maintained with contributions from Apple, Google, Microsoft, Mozilla, and other companies.

The idea behind Wasm is to provide a simple assembly-like instruction set which can run efficiently within a browser. At its core, Wasm includes a binary instruction format and a stack-based virtual machine that supports functions and control flow abstractions like loops and conditionals.

Although browsers are the main target of Wasm, recent initiatives, like WebAssembly System Interface [14] (WASI), norm the implementation of Wasm runtimes to support the execution of Wasm code outside the browser with a set of APIs that provide POSIX capabilities (e.g., file system, network, and process management). Some examples of open-source and proprietary WASI-compliant runtimes are Wasmtime [15], Wasmer [16], and WasmEdge [17].

Focussing on FaaS, Wasm provides a sandboxed runtime environment for functions, akin to containers. However, while one needs to build a container (for the same function) for each targeted architecture, the same Wasm binary can run on different architectures thanks to the hardware abstraction provided by the Wasm runtime. Moreover, Wasm binaries tend to be more lightweight than containers, thanks to the fact that they do not need to include a pre-packaged filesystem.

III. FunLess Platform Architecture

We now present the principles and technologies behind FunLess and its architecture. We also discuss FunLess' design choices, trade-offs, and limitations.

The main principles behind the design of FunLess are the simplicity of both function development and platform deployment and the flexibility of hardware and deployment automation. In particular, FunLess is independent of the underlying deployment orchestrators (if any), which avoids potential overheads and allows users to install the entire platform on resource-constrained, low-power edge devices. For the implementation of the platform, we used Elixir [18], which is a functional, concurrent, high-level general-purpose programming language that runs on the BEAM virtual machine [19] (used by the Erlang language). Elixir and the BEAM allowed us to simplify the creation and deployment of FunLess' distributed architecture with high performance, fault-tolerance, and resilience without relying on container orchestration technologies—the BEAM's scheduler and lightweight processes are optimised for concurrent, distributed systems.

[4]https://developers.cloudflare.com/workers/runtime-apis/webassembly/.

[5]FunLess users can write functions in any language supported by the platform, currently Rust, Go, and JavaScript.

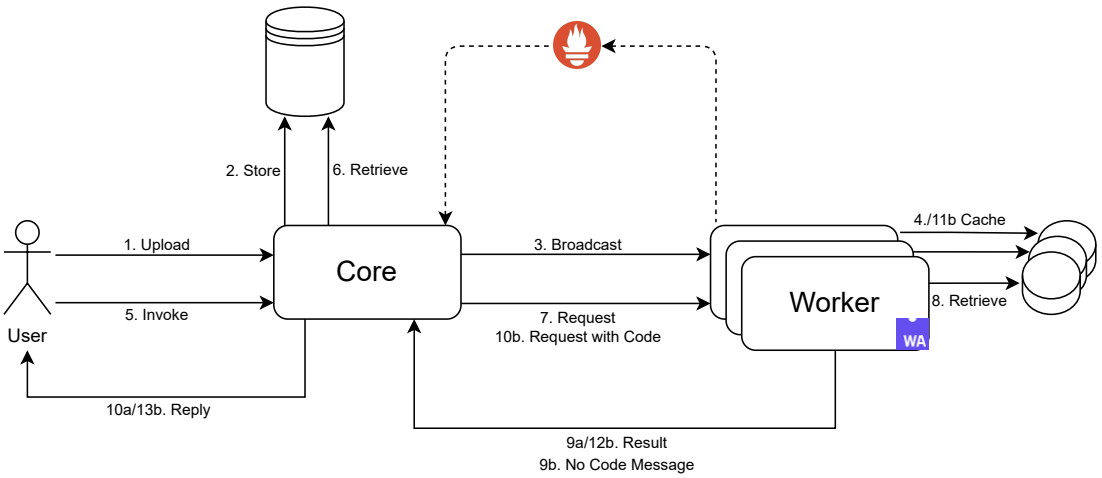

Fig. 1. Architecture of the FunLess platform with the function flow from creation to invocation.

We represent in Figure 1 both the components that make up the platform's architecture and the flow developers and users follow to create and invoke functions. Architecture-wise, FunLess consists of mainly two components: the *Core* and the *Worker*, detailed in the next parts of this section. Briefly, the *Core* acts as an user-facing API to *i*) create, fetch, update, and delete functions and *ii*) 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 *Worker*s. In addition to these two components, FunLess includes a *Postgres* database to store functions and metadata and *Prometheus* to collect metrics from the platform.[6]

FunLess is an open-source project and both its source code [20] and documentation [21] are publicly available.

A. Core

The *Core* is the controller of the platform. It exposes an HTTP REST API to the users, handles authentication and authorization, and manages functions' lifecycle and invocations.

Although the *Core* implements the main coordination logic and functionalities of FunLess, it is a lightweight component. For instance, on a Raspberry Pi 3B+ its local bare-metal deployment (that includes the database, the monitoring system and the underlying operating system and services) occupies ca. 600 MB of RAM when idle.

Functionality-wise, FunLess users create a new function by compiling its source code to Wasm—with either Rust's default compiler or dedicated ones for Go and JavaScript—and uploading the compiled binary, paired with a name, to the *Core*. Users can group functions in modules and, when uploading a function, they can specify which module the function belongs to. Moreover, users should also specify the amount of memory reserved for the execution of the function.

Looking at the steps reported in Fig. 1, once the *Core* receives the request to create a function (1. Upload), it stores its binary in the database (2. Store). Fetch, update, and deletion happen via the assigned function name. When the *Core* successfully creates a function, it notifies the *Worker*s (3. Broadcast) to store a local copy of the function binary (4. Cache) compiled from the source code with the given metadata (i.e., module and function names). This push strategy helps to reduce part of the overhead of cold starts. Indeed, most FaaS platforms follow a pull policy where, if the execution nodes do not have the function in their cache (e.g., it is the first time they execute it), they fetch, cache, and load the code of the function, undergoing latency. The small occupancy of Wasm binaries makes it affordable for FunLess to employ a push strategy, helping to reduce cold-start overheads.

Since both the *Core* and the *Worker*s run on the BEAM, these components communicate via the BEAM's built-in lightweight distributed inter-process messaging system, avoiding the need (complexity, weight) for additional dependencies for data formatting, transmission, and component connection.

After receiving a function invocation (5. Invoke), the *Core* checks if the function is in the database and preemptively retrieves its code (6. Retrieve) in case the *Worker* that will run the function does not have it in its cache.

Then, the *Core* uses the most recent metrics—we represent the pushing of the data, updated every 5s by default, from Prometheus to the Core with the dashed line in Fig. 1—to select on which of the available *Worker*s to allocate the function (7. Request). The selection algorithm starts from the *Worker* with the largest amount of free memory to the one with the smallest. If no worker has enough memory to host the function, the invocation returns an error.

After the *Worker* successfully ran the function (we detail this part of the workflow in the section about *Worker*s, below) it sends back to the *Core* the result (if any), which the *Core* relays back to the user (10a/13b. Reply). If no *Worker* is available at scheduling time or there are errors during the execution, the *Core* returns an error.

Another important feature of FunLess is that the *Core*

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

can automatically discover the *Worker*s in its same network. This feature derives from Elixir's libcluster library[7], which provides a mechanism for automatically forming clusters of BEAM/Erlang nodes. Technically, when deployed on bare metal, FunLess follows the Multicast UDP Gossip algorithm of the library, to automatically find available *Workers*. Instead, when deployed using Kubernetes, FunLess relies on the service discovery capabilities of the container orchestration engine to connect the *Core* with the *Worker*s, paired with the "Kubernetes" modality of the library. Users can manually connect *Worker*s from other networks via a simple message (e.g., a ping) thanks to the BEAM's built-in capability of connecting to other BEAM nodes.

B. Worker

The *Worker* executes the functions per *Core*'s requests. The *Worker*s run functions via Wasmtime, a WASI-compliant Wasm runtime by the Bytecode Alliance [22]. The main reasons behind using Wasmtime include ease of integration, amount of contributors, and security-oriented focus of the project. While *Worker*s use Wasmtime, we modelled them to abstract away the peculiarities of specific Wasm runtimes so that future variants can use different runtimes and even extend support for multiple ones (e.g., specified by the users).

When a *Worker* receives a request to execute a function (7. Request), it first checks whether it has a cached version of the function's binary (8. Retrieve). If that is the case, it loads and runs the function's binary and returns to the *Core* the result of the computation (9a. Result). To reduce unnecessary data transfers between the nodes, the *Code* does not send to the *Worker* the function's code right away (7. Request). If the *Worker* does not find the code of the function in its local cache, it contacts the *Core* (9b. No Code Message), which responds with a request that carries the code of the function to the *Worker* (10b. Request with Code), which was initially retrieved by the *Core* (6. Retrieve). Upon reception, the *Worker* compiles the code, caches the binary (11b. Cache), loads it to run the function, and relays the result to the *Core* (12b. Result).

The above mechanism is an important advantage afforded by FunLess for the edge case. Function fetching (if needed) transmits small pieces of binary code (rather than heavyweight containers). Wasm binaries achieve the two-fold objective of having *Worker*s run functions on different hardware architectures (e.g., AMD64, ARM) and allowing users to write their functions once, knowing that they will execute irrespective of the hardware of the *Worker*.

Summarising, fetching and precompiling (if any, depending on cache status) constitutes most of the "cold start" overhead in FunLess, which the platform greatly reduces w.r.t. alternatives that rely on bandwidth- and memory-heavier containers.

Regarding caching and eviction, *Worker*s set a threshold size for the cache memory (configurable at deployment time) which, when exceeded triggers the *Worker* to evict the function(s) with the longest period of inactivity (invocation- or update-wise). Additionally, *Worker*s evict functions if inactive for a set amount of time (by default, 45 minutes).

At function updates, the *Core* pushes the updated code to all *Worker*s. Similarly, the *Core* propagates function deletions to all *Worker*s, requesting the removal of the deleted functions' code from their caches.

C. Design Choices and Limitations

Since a small resource footprint and simplicity are the driving principles behind FunLess' implementation, we favoured design choices (both w.r.t. the components in the architecture and the internal implementation) that introduce the least complexity while affording flexibility (of implementation and deployment). Below, we discuss the main aspects that FunLess trades off for the above benefits.

Language support. To run functions, FunLess requires users to compile them to Wasm. Technically, a *Worker* interacts with a function by having the latter expose a "wrapper" that performs input and output (de)serialisation. Therefore, FunLess provides implementations of these wrappers for each language it supports; depending on the language, a wrapper can be a library, macro or compiler extension. While extending support for different languages is non-essential to this introductory presentation, FunLess supports three languages: Rust, Go and JavaScript—and we plan to support more in the future. Specifically, we choose Rust for its performance, its growing developer community, and its ease of compiling to Wasm; similarly, Go is famous for its performance and widespread use in cloud systems; lastly, JavaScript is one of the most popular programming languages.

Resilience. The *Core* component, which acts as the sole scheduler and holder of the architecture's state, reduces the footprint of the platform by centralising its control. However, having one *Core* makes it a single point of failure of the architecture. The BEAM opportunely guarantees fault-tolerance, so that the *Core* can recover from software crashes, losing only the invocations in transit (which the users would notice as timed out) while the rest of the system would recover (normal functionality, connections to the *Worker*s, metrics, and storage) following the connection protocols mentioned above. Contrarily, if the hardware hosting the *Core* failed the platform would stop working properly.

Robustness. FunLess implements an at-most-once message relay policy, hence, lost messages between the *Core* and *Worker*s imply the failure of the invocation. Implementing more robust semantics, like at least or exactly once, would require the inclusion of a message broker, increasing the load on nodes and the architecture's complexity.

Retry policies. The *Core* does not implement retry policies. Thus, if a function's execution failed on the chosen *Worker* or the latter became unresponsive, the *Core* would not retry running the function on another *Worker*. Implementing retry policies would increase the complexity platform-wide. Specifically, the *Core* would need to keep track of the state of function invocations, increasing the amount of coordination/messages with the *Worker*s. This extension would also increase the

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

amount of data and interactions with the database (needed to enforce the transactional management of functions' state and stave off the risk of losing this data due to crashes) and further complicate the *Core*'s implementation to manage back-off strategies and execution time limits. Nonetheless, we plan to implement "opt-in" retries (the BEAM already provides some building blocks for the task, used to implement function timeouts and monitoring), giving users the flexibility to choose between a lighter setup or increased reliability.

IV. Related Work

We start our review of related work by looking at widely-adopted, open-source implementations of FunLess alternatives. Specifically, we focus on platforms whose sources are available on GitHub, which allows us to quantify their popularity (e.g., via GitHub stars) and activeness (e.g., commits).

At the time of writing, GitHub has 3.9k matches when searching for the keyword "faas". When focusing on FaaS solutions that are production-ready (used in industry, verified by looking at the commercial 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 the cloud and edge devices), the first three platforms ranked by popularity (GitHub stars) are OpenFaaS (24k+ stars), Fission (8k+ stars), and Knative (5k+ stars).[8]

To draw our comparison, we highlight each platform's main traits that contrast with FunLess.

OpenFaaS: OpenFaaS builds on Kubernetes, and it takes advantage of Kubernetes' scheduler for function allocation and scaling—specifically, functions are pods, i.e., application containers that enclose the function's code and runtime environment. Since pods are generic containers, OpenFaaS supports different languages by providing language-specific template containers with example source files that users can extend to implement their functions and include the necessary dependencies. The free version of OpenFaaS has several limitations, in particular, it lacks scale-to-zero, which inevitably wastes resources by keeping at least one running pod for each function at all times irrespective of idle periods.

Fission: Like OpenFaaS, also Fission builds on Kubernetes. However, Fission does not rely on user-built containers for functions, allowing users to directly upload their source code. Functions run through the use of "environments", which essentially define which pre-built containers the platforms shall use to run the functions' sources (as compiled binaries or via an interpreter). One of the strengths of Fission is the small cold-start times it affords for functions deployed as source code (i.e., not binary executables). To achieve this result, Fission initialises "general-purpose" containers for the language environment of the deployed functions. At function invocation, Fission uses one of these "warm" containers by injecting and running therein the code of the function. Although this approach reduces cold-start times, it consumes resources (CPU, memory, energy) to keep the pool of warm containers, an issue that can hinder performance on constrained devices.

Knative: Knative is also a Kubernetes-based serverless platform. The main difference with OpenFaaS and Fission is that Knative adopts a more low-level approach to function development. Essentially, developers implement their functions as containerised microservices (the state-of-the-art programming style complementary to serverless for cloud-based architectures [23]), which Knative executes in a serverless-like fashion (managing event-based allocation and scaling).

FunLess stands out among these alternatives primarily thanks to its focus on Wasm for the function runtime, which reduces cold-start times and overhead due to containerisation technologies. Wasm also allows developers to write functions in any of the languages FunLess supports, leaving to the platform the duty of executing them on heterogeneous architectures without requiring a dedicated compilation. Since FunLess forgoes middleware like Kubernetes, it affords low-resource requirements (and deployment complexity) and allows users to deploy it on low-power, resource-constrained edge nodes. In particular, we are not aware of other serverless platforms that support a Wasm runtime and where both the controller and the workers can run on resource-constrained edge nodes.

Looking at the work from the literature most closely related to FunLess, we have several proposals targeting edge and cloud scenarios. From the review by Cassel et al. [24], most of the solutions (86%) for IoT/edge rely on some container technology while promising technologies like WebAssembly and Unikernels represent only 2-3% of the proposals.

Focusing on serverless platforms supporting Wasm runtimes, Hall and Ramachandran [10] are among the first to advocate WebAssembly as the enabling technology to avoid the overhead of containers, which substantially weigh on the limited hardware resources of edge computing environments. The authors presented a serverless platform that runs WebAssembly code within the V8 JavaScript engine for execution and sandboxing of functions. Differently from FunLess, they use a NodeJS runtime that embeds V8 for the running Wasm code. As the authors note [10], the nesting of these layers takes a conspicuous toll on the performance of the system.

Gadepalli et al. [11] use WebAssembly to run and sandbox serverless functions. They target only single-host deployments, requiring the deployment of the entire platform on one node only. Moreover, they do not support WASI [14], thus making their system potentially less portable.

Gackstatter et al. [25] propose WOW, a WebAssembly-based runtime environment for serverless edge computing integrated within the Apache OpenWhisk platform. The authors introduce a new layer between OpenWhisk and different Wasm runtimes which enable the execution of Wasm functions. Compared to FunLess, WOW requires the deployment of a full installation (of a custom version) of the OpenWhisk platform

[8]Resp. found at https://github.com/OpenFaaS/faas, https://github.com/fission/fission, and https://github.com/knative. Notably, Apache OpenWhisk, a popular (6k+ stars) serverless platform, misses the podium because it has no ARM images for its main components (Controller and Invoker).

which precludes the installation of the controller to low-power and memory-restricted edge devices.[9]

Lucet [26] was used by Fastly to run Wasm on their commercial Compute platform. Lucet translated WebAssembly to native code, which was then executed using Lucet's runtime also on edge devices. Unfortunately, Lucet has reached end-of-life and is no longer maintained. Cloudflare Workers [27] is also a commercial serverless platform that supports the possibility of defining functions in Wasm and has native support for WASI since 2022.[10] Although the runtime part of this project has recently been made open-source,[11] the serverless platform is proprietary and closed-source.

It is worth mentioning the work by Shillaker and Pietzuch [28] that, tangential to our proposal, concerns a Wasm-based serverless runtime that uses Wasm to achieve state sharing across functions—they allow the execution of functions that share memory regions in the same address space for possible performance benefits. On a similar note, Zhao et al. [29] present an OpenWhisk extension for confidential serverless computing that integrates a Wasm runtime. The authors propose a solution to construct reusable enclaves that enable rapid enclave reset and robust security to reduce cold start times. Although these kinds of proposals are orthogonal to FunLess, we see them as future optimisations that the usage of a Wasm function runtime can unlock for FunLess.

Kjorveziroski and Filiposka [30] focus on serverless orchestration using Wasm and introduce a variant of Kubernetes that can orchestrate Wasm modules that are executed without containers. Interestingly, also Kjorveziroski and Filiposka report that Wasm tasks enjoy faster deployment times (two-fold) and at least one order of magnitude smaller artefact sizes, while still offering comparable execution performance.

Finally, Tzenetopoulos et al. [31] analyse the performance of *Lean OpenWhisk*, an edge-focused variant of the Apache OpenWhisk serverless platform. Their variant of the platform coalesces the scheduling and execution components in a single entity, removes the message broker (Apache Kafka) from the deployment, and introduces changes to reduce OpenWhisk's overhead, making it better suited for resource-constrained devices.

V. Conclusion

We present FunLess, a FaaS platform tailored to respond to recent trends in serverless computing that advocate for extending FaaS to cover private edge cloud systems, including Internet-of-Things devices. The motivation behind the shift towards private edge cloud systems includes reduced latency, enhanced security, and improved resource usage. Unlike existing solutions that rely on containers and container orchestration technologies for function invocation, FunLess leverages Wasm as its function-execution runtime environment. The reason behind this choice is to reduce performance overheads that can prevent resource-constrained devices from running FaaS systems. Wasm's fundamental feature exploited by FunLess is its lightweight, sandboxed runtime, which allows the platform to run efficiently functions in isolation on constrained devices at the edge. Thus, Wasm provides a portable, homogeneous way for developers to implement and deploy their functions among clusters of heterogeneous devices (write once, run everywhere), simplifying platform deployments, offering flexibility in deployment options, and mitigating cold start issues.

We started to benchmark FunLess against existing serverless platforms and several deployment scenarios, considering private, public, and mixed cloud-edge configurations [32]. These preliminary experiments show that, particularly in edge scenarios, FunLess outperforms alternatives like OpenFaaS, Fission, and Knative in terms of memory footprint without substantial performance degradation.

As future work, we plan to integrate new versions of Wasmtime and, with it, native support for HTTP and other optimisations and features of the new releases and support for the WASI runtime. Indeed, many current Wasm runtime implementations miss features like interface types, networking support in WASI multi-threading, atomics, and garbage collectors. Besides Wasmtime, other projects are developing new, optimised, and extended Wasm runtimes, which FunLess can leverage to increase its performance (and adapt it to different application contexts). For example, the support for garbage collection can lead to improved JavaScript runtimes and increase the performance of this kind of functions.

From the point of view of feature support, we deem supporting function composition in FunLess both important for the users and beneficial for performance. Indeed, FunLess currently supports function composition by publicly exposing the functions in a flow and chaining them via their public endpoints. In the future, we propose to study how technologies like FaaSFlow [33], Palette [34], AWS Step Functions [35], Azure Durable Functions [36] work and integrate them into FunLess. In particular, since FunLess uses Wasm, an interesting direction is exploiting memory sharing to have Wasm functions in a flow to avoid the overhead of network communication by letting chained Wasm functions work on the same memory block to store and retrieve their data.

We also plan to improve the reliability of the platform, allowing the support of retry policies for failed invocations, at-least-once message delivery, and the replication of the *Core* components. Following the principles of simplicity and versatility that guided the development of FunLess, we propose to tackle these extensions as optional features to support flexible deployments, adaptable to the different application contexts (cloud, edge, on resource-constrained devices).

Finally, we plan to ease the deployment of FunLess by supporting other tools like, e.g., Nomad [37] and optimise the platform for edge devices by using, e.g., Nerves [38] to further minimise the overhead on bare-metal deployment.

[9] We tried to deploy WOW on a multi-host cloud configuration for comparison purposes. Unfortunately, the deployment failed (the platform relies on an old and modified version of OpenWhisk that is not supported anymore, i.e., the last commit in the project is older than 2 years).

[10] https://blog.cloudflare.com/announcing-wasi-on-workers

[11] https://blog.cloudflare.com/workerd-open-source-workers-runtime/

References

[1] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," *CoRR*, vol. abs/1902.03383, 2019.

[2] P. Castro, V. Isahagian, V. Muthusamy, and A. Slominski, *Hybrid Serverless Computing: Opportunities and Challenges*. Cham: Springer International Publishing, 2023, pp. 43–77.

[3] L. Baresi and D. F. Mendonça, "Towards a serverless platform for edge computing," in *2019 IEEE ICFC*. IEEE, 2019, pp. 1–10.

[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. 239:1–239:32, 2022. [Online]. Available: https://doi.org/10.1145/3510611

[5] M. Salehe, Z. Hu, S. H. Mortazavi, I. Mohomed, and T. Capes, "Videopipe: Building video stream processing pipelines at the edge," in *Proceedings of the 20th International Middleware Conference Industrial Track, Davis, CA, USA, December 9-13, 2019*, D. S. Milojicic and V. Muthusamy, Eds. ACM, 2019, pp. 43–49. [Online]. Available: https://doi.org/10.1145/3366626.3368131

[6] G. George, F. Bakir, R. Wolski, and C. Krintz, "Nanolambda: Implementing functions as a service at all resource scales for the internet of things," in *5th IEEE/ACM Symposium on Edge Computing, SEC 2020, San Jose, CA, USA, November 12-14, 2020*. IEEE, 2020, pp. 220–231. [Online]. Available: https://doi.org/10.1109/SEC50012.2020.00035

[7] C. Mistry, B. Stelea, V. Kumar, and T. F. J. Pasquier, "Demonstrating the practicality of unikernels to build a serverless platform at the edge," in *12th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2020, Bangkok, Thailand, December 14-17, 2020*. IEEE, 2020, pp. 25–32. [Online]. Available: https://doi.org/10.1109/CloudCom49646.2020.00001

[8] A. Jangda, D. Pinckney, Y. Brun, and A. Guha, "Formal foundations of serverless computing," *Proc. of ACM on Prog. Lang.*, vol. 3, no. OOPSLA, pp. 1–26, 2019.

[9] V. Kjorveziroski and S. Filiposka, "Webassembly as an enabler for next generation serverless computing," *J. Grid Comput.*, vol. 21, no. 3, p. 34, 2023. [Online]. Available: https://doi.org/10.1007/s10723-023-09669-8

[10] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation, IoTDI 2019, Montreal, QC, Canada, April 15-18, 2019*, O. Landsiedel and K. Nahrstedt, Eds. ACM, 2019, pp. 225–236. [Online]. Available: https://doi.org/10.1145/3302505.3310084

[11] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: a serverless-first, light-weight wasm runtime for the edge," in *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*, D. D. Silva and R. Kapitza, Eds. ACM, 2020, pp. 265–279. [Online]. Available: https://doi.org/10.1145/3423211.3425680

[12] P. Vahidinia, B. J. Farahani, and F. S. Aliee, "Cold start in serverless computing: Current trends and mitigation strategies," in *2020 International Conference on Omni-layer Intelligent Systems, COINS 2020, Barcelona, Spain, August 31 - September 2, 2020*. IEEE, 2020, pp. 1–7. [Online]. Available: https://doi.org/10.1109/COINS49042.2020.9191377

[13] "Webassembly," https://webassembly.org/, 8 2023.

[14] "Webassembly system interface," https://wasi.dev/, 8 2023.

[15] "Wasmtime," https://wasmtime.dev/, 8 2023.

[16] "Wasmer," https://wasmer.io/, 8 2023.

[17] "Wasmedge," https://wasmedge.org/, 8 2023.

[18] S. Juriè, *Elixir in action*. Manning, 2024.

[19] E. Stenman, "Beam: a virtual machine for handling millions of messages per second (invited talk)," in *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, ser. VMIL 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 4. [Online]. Available: https://doi.org/10.1145/3281287.3281289

[20] "FunLess Repository," Link omitted for blind review purposes., 2024.

[21] "FunLess Website," Link omitted for blind review purposes., 2024.

[22] "Bytecode alliance," https://bytecodealliance.org/, 2024.

[23] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[24] G. A. S. Cassel, V. F. Rodrigues, R. da Rosa Righi, M. R. Bez, A. C. Nepomuceno, and C. A. da Costa, "Serverless computing for internet of things: A systematic literature review," *Future Gener. Comput. Syst.*, vol. 128, pp. 299–316, 2022. [Online]. Available: https://doi.org/10.1016/j.future.2021.10.020

[25] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing serverless to the edge with webassembly runtimes," in *22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2022, Taormina, Italy, May 16-19, 2022*. IEEE, 2022, pp. 140–149. [Online]. Available: https://doi.org/10.1109/CCGrid54584.2022.00023

[26] "Lucet," https://github.com/bytecodealliance/lucet, 2020.

[27] Cloudflare, "How Workers works," https://developers.cloudflare.com/workers/reference/how-workers-works/, 1 2024.

[28] S. Shillaker and P. R. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, A. Gavrilovska and E. Zadok, Eds. USENIX Association, 2020, pp. 419–433.

[29] S. Zhao, P. Xu, G. Chen, M. Zhang, Y. Zhang, and Z. Lin, "Reusable enclaves for confidential serverless computing," in *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, J. A. Calandrino and C. Troncoso, Eds. USENIX Association, 2023, pp. 4015–4032. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/zhao-shixuan

[30] V. Kjorveziroski and S. Filiposka, "Webassembly orchestration in the context of serverless computing," *J. Netw. Syst. Manag.*, vol. 31, no. 3, p. 62, 2023. [Online]. Available: https://doi.org/10.1007/s10922-023-09753-0

[31] A. Tzenetopoulos, E. Apostolakis, A. Tzomaka, C. Papakostopoulos, K. Stavrakakis, M. Katsaragakis, I. Oroutzoglou, D. Masouros, S. Xydis, and D. Soudris, "Faas and curious: Performance implications of serverless functions on edge computing platforms," in *High Performance Computing: ISC High Performance Digital 2021 International Workshops, Frankfurt Am Main, Germany, June 24 – July 2, 2021, Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 428–438. [Online]. Available: https://doi.org/10.1007/978-3-030-90539-2_29

[32] G. D. Palma, S. Giallorenzo, J. Mauro, M. Trentin, and G. Zavattaro, "Funless: Functions-as-a-service for private edge cloud systems," *CoRR*, vol. abs/2405.21009, 2024. [Online]. Available: https://arxiv.org/abs/2405.21009

[33] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "Faasflow: enable efficient workflow execution for function-as-a-service," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds. ACM, 2022, pp. 782–796. [Online]. Available: https://doi.org/10.1145/3503222.3507717

[34] M. Abdi, S. Ginzburg, X. C. Lin, J. M. Faleiro, G. I. Chaudhry, I. Goiri, R. Bianchini, D. S. Berger, and R. Fonseca, "Palette load balancing: Locality hints for serverless functions," in *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, G. A. D. Luna, L. Querzoni, A. Fedorova, and D. Narayanan, Eds. ACM, 2023, pp. 365–380. [Online]. Available: https://doi.org/10.1145/3552326.3567496

[35] AWS, "Serverless workflow orchestration – aws step functions – amazon web services," url=https://aws.amazon.com/step-functions/.

[36] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Durable functions: semantics for stateful serverless," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–27, 2021. [Online]. Available: https://doi.org/10.1145/3485510

[37] "Nomad," https://www.nomadproject.io/, 2024.

[38] "Nerves project," https://nerves-project.org/, 2024.