Pushing Serverless to the Edge with WebAssembly Runtimes

Philipp Gackstatter, Pantelis A. Frangoudis, and Schahram Dustdar
Distributed Systems Group, TU Wien, Vienna, Austria
Email: philipp.gackstatter@student.tuwien.ac.at, pantelis.frangoudis@dsg.tuwien.ac.at, dustdar@dsg.tuwien.ac.at

2022 22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 978-1-6654-9956-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/CCGRID54584.2022.00023

***Abstract*—Serverless computing has become a popular part of the cloud computing model, thanks to abstracting away infrastructure management and enabling developers to write functions that auto-scale in a polyglot environment, while only paying for the used compute time. While this model is ideal for handling unpredictable and bursty workloads, cold-start latencies of hundreds of milliseconds or more still hinder its support for latency-critical IoT services, and may cancel the latency benefits that come with proximity, when serverless functions are deployed at the edge. Moreover, CPU power and memory limitations which often characterize edge hosts drive latencies even higher. The root of the problem lies in the de facto runtime environments for serverless functions, namely container technologies such as Docker. A radical approach is thus to replace them with a more light-weight alternative. For this purpose, we examine WebAssembly's suitability for use as a serverless container runtime, with a focus on edge computing settings, and present the design and implementation of a WebAssembly-based runtime environment for serverless edge computing. WOW, our prototype for WebAssembly execution in Apache OpenWhisk, reduces cold-start latency by up to 99.5%, can improve on memory consumption by more than 5×, and increases function execution throughput by up to 4.2× on low-end edge computing equipment compared to the standard Docker-based container runtime for various serverless workloads.**

***Index Terms*—Function-as-a-Service, edge computing, serverless, WebAssembly**

I. Introduction

When latency matters and data intensity is high, executing IoT service logic in the cloud is challenged, due to the latter's physical distance and the sheer amount of data devices at the edge of the network generate. Combined with privacy concerns, this calls for pushing services to the edge, for data processing and decision making in-place or nearby. However, applying traditional IaaS or PaaS cloud models to host such services at edge infrastructure can have adverse effects, due to resource limitations that make virtual machine or container allocation and scaling expensive [1]. This further complicates the elastic management of event-driven IoT services, which are often associated with bursty and unpredictable workloads. A model with finer grained resource elasticity is thus required.

One piece of the puzzle towards realizing this goal is serverless (edge) computing [2]. Serverless, commonly in the form of Function-as-a-Service (FaaS), allows developers to execute functions over cloud infrastructure without having to specify how the latter is set up, managed, and auto-scaled. The serverless provider ensures precise per-function provisioning and pure pay-per-use at the function level, following the *scale-to-zero* principle, i.e., de-allocating resources not in use.

To isolate function instances in a multi-tenant environment, serverless frameworks make use of OS-level virtualization through containers–usually of the Docker flavor. When a function is first invoked, its container is provisioned from scratch. This is referred to as *cold start* and can introduce latencies of hundreds of ms or more [3]. For example, median cold start latencies of 250-265 ms and 110-493 ms have been reported [4] for AWS and Google Cloud Platform, respectively. Particularly when operating at low-end edge computing equipment [5]–[7], and in the face of concurrent requests [8] typical of bursty workloads, cold start latencies are driven further up.

At the heart of the problem lies the container runtime and its expensive startup procedure. This issue is partially addressed by keeping containers warm in-between requests. For instance, Apache OpenWhisk,[1] a popular open-source serverless framework, keeps a function's container paused and ready for reuse for 10 minutes, before removing it entirely, while AWS Lambda's cold start policy keeps an instance alive for 5-7 minutes [9]. However, this is a form of over-provisioning and therefore opposed to the scale-to-zero premise.

We believe OS-level virtualization to be unsuitable for serverless edge computing. We thus follow a more radical approach, as has been recently suggested [10]–[12], by replacing the container runtime with an alternative offering more efficient cold starts. A technology that can play this role is WebAssembly (Wasm) [13], a portable, binary instruction format for memory-safe, sandboxed execution. Its portability means that a compiled Wasm function can be executed wherever a runtime exists. Wasm can be compiled with different strategies, some reaching near-native execution speeds to compete with function invocation within a Docker container. Importantly, Wasm functions can be created and destroyed in microseconds.

With this background, we set off to address our key research question: *the viability of WebAssembly in serverless edge computing and the performance benefits it can bring about*. We make the following contributions: ① We design an execution environment for Wasm serverless functions, with the requirements of multi-platform edge execution and ease of integration with existing serverless frameworks in mind (§III). ② We present WOW (§IV), a prototype for executing Wasm workloads in Apache OpenWhisk. WOW is extensible in

[1] https://openwhisk.apache.org/

978-1-6654-9956-9/22/$31.00 ©2022 IEEE
DOI 10.1109/CCGrid54584.2022.00023

terms of the Wasm runtimes it supports, requires minimal modifications to OpenWhisk, and seamlessly integrates Wasm special features such as capability-based access control. WOW is available as open source.[2] ③ We extensively experiment with WOW over different serverless workloads and on diverse edge compute infrastructure (§V). WOW reduces cold start latency by up to 99.5% on small-scale single-board computers (SBC) and to 94% on server-class machines, increases function execution throughput by up to 4.2× and 3×, respectively, and can reduce memory footprint by more than 5× compared to the current practice of using Docker runtimes. These results are particularly important in resource-constrained edge settings: Despite existing limitations (§VII) and still-noticeable cold starts, Wasm runtimes make it feasible to execute serverless workloads on low-end edge hosts with acceptable latency, something impractical with current container runtimes.

II. Preliminaries

A. Serverless workload

Developers use serverless for diverse use cases, creating a mix of CPU and I/O bound function workloads [14]. Shahrad et al. [15] find that 81% of functions on Microsoft Azure are invoked less than once per minute, and 50% execute for less than 1 s. However, those accessed more frequently constitute 99.6% of all invocations. This is corroborated by the cold start times reported by Wang et al. [4]. Eismann et al. [16] analyzed 89 open-source serverless applications, of which 84% have bursty workloads. Thus, these spikes in demand–manifested in concurrent requests, in turn causing up-scaling–ought to be handled well by serverless platforms. Furthermore, 39% of the applications have a high traffic intensity, 47% have a low traffic intensity and 17% utilize scheduled functions. Thus, the latter combined 64% represent on-demand scenarios where the platform is likely to experience cold starts. There is a fundamental trade-off between using memory to keep functions warm and forgoing it, but incurring the additional latency of a cold start. Keeping functions warm burdens the operator while also violating the scale-to-zero principle, therefore *short cold starts help reduce costs*. In light of these findings, a primary goal of our work is making cold starts faster, in turn enabling reduced keep-alive times.

B. WebAssembly

To achieve this goal, we build on WebAssembly as the function execution environment. Wasm was intended to optimize client-side code execution in the browser and act as a compilation target for JavaScript, but is recently considered for use in Serverless [10], [11], [17]–[19]. FaaS developers can write functions in a variety of languages, compile them to Wasm, and execute them in a Wasm runtime. While cold-starting a Wasm function can be very fast, Wasm has been found [20], [21] to execute 10%-50% slower than native code. However, Wasm can be compiled to native code, either by just-in-time (JIT) engines at the time of execution, or ahead-of-time (AoT) by the same JIT engines or AoT compilers. AoT compilation is a technique we exploit to to reduce the time it takes to get a Wasm module ready for execution.

Security-wise, Wasm uses software-based fault isolation techniques to sandbox the executing module. Wasm interacts with the host system via the WebAssembly System Interface (WASI) [22]. A Wasm module cannot directly perform an OS system call due to sandboxing, but imports equivalent WASI functions instead. WASI is binary-compatible and specifies a fine-grained capability-based security model.

III. Architecture of a Wasm-based FaaS platform

We present a high-level view of an architecture that enables integrating Wasm as the runtime environment for executing serverless workloads. Our design and technological choices are driven by specific requirements, with a view to edge execution.

A. High-level view

1) Design requirements: Operators are forced to choose between saving cost and resources or enabling a high quality of service–in particular shorter response times. While this trade-off is inevitable to some degree, cold starts exacerbate it. A new container runtime must alleviate that initial latency, to provide mutual benefits for the serverless user and the operator. The runtime needs to be programming-language agnostic and provide cheap sandboxing to ensure multi-tenant capability with no adverse effects on the cold start latency. These features are readily provided by Wasm. Furthermore, the Wasm runtime environment needs to be easily integratable with existing serverless frameworks and offer as close to native speed as possible, to be a viable alternative to de facto container runtimes such as Docker. Finally, targeting the whole edge-fog-cloud continuum, which is notorious for its host heterogeneity [23]–[25], the introduced runtime should be able to execute on devices with potentially different capabilities and instruction set architectures (at least x86_64 and Arm-based).

2) Generalized serverless architecture and function lifecycle: Popular open-source serverless frameworks, such as OpenFaaS,[3] Apache OpenWhisk and OpenLambda,[4] are characterized by a similar general architecture. Users interact with the system through an API Gateway (acting as a single point of entry and for TLS termination), and a *Controller* implements the main logic regarding resource allocation, authorization, and function scheduling over the underlying compute infrastructure. A user can interact with the system through the gateway and manage functions via CRUD (Create, Read, Update, Delete) operations, set up scheduled triggers or retrieve the results of previous invocations. Upon each function invocation, a load balancer selects one of potentially multiple *Invokers* to execute the function on a host. Each host runs a sandboxing execution engine, such as Docker or a similar container runtime, which runs user functions in an isolated manner.

[2]https://github.com/PhilippGackstatter/wow

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

[4]https://github.com/open-lambda/open-lambda

Serverless frameworks allow developers to write their functions in a variety of languages. Many popular languages need a runtime, such as `node.js`, a Python interpreter or a Java Runtime Environment (JRE). State-of-the-art serverless platforms typically bake these runtimes into container images. To execute a function, the container image with the appropriate runtime is pulled from a container registry. The function's code is retrieved from an internal database. The container is started and the code is injected and prepared for execution. Finally, the function is invoked with the user's parameters and the result is returned to the framework, which relays it to the user.

To seamlessly integrate different runtimes, a common protocol is needed so that the Controller communicates with each of them. Therefore each runtime implements and exposes a number of endpoints. An *instantiation* endpoint creates a new container and returns its address. A request to an *initialization* endpoint is sent to this address, where the container receives the code and prepares the function for execution. Once initialized, an *execution* endpoint can be called with different parameters many times to invoke the function. Serverless platforms use various optimizations of the described flow, such as pre-warming containers or keeping the container alive after invocation, to avoid cold start costs at the expense of resource use. In the absence of invocations and after some threshold time elapses, the container is removed entirely by accessing the container runtime's *termination* endpoint.

3) WebAssembly runtime support: We introduce a lightweight runtime management layer that has a similar role as traditional container runtimes (e.g., Docker), but instead manages the execution of Wasm functions. We adopt the terminology of Apache OpenWhisk, our serverless platform of choice, where the term *action* is used to denote a serverless function, and introduce the following components of this layer:
Executor: This is in charge of the actual execution of a Wasm function on an underlying host. The executor wraps around a Wasm runtime binary (*Wasm container runtime*) targeting the instruction set architecture (ISA) of the host, and exposes function creation, initialization, execution, and termination endpoints. It is the equivalent of a container runtime.
Invoker: Upon receiving a request from the API gateway, the Invoker applies the respective function lifecycle management action. It interacts with an Executor to prepare a function for execution, run it, and relay the results to the user.
Wasm module: This is the actual Wasm code of a serverless function. It is the result of compilation which can happen ahead of time (when a function is submitted by a user) and stored serialized in a Function Store. We use the term *module* as the equivalent of a container image, and the term *instance* to denote a module that is currently run by an Executor.

This serverless design is shown in Fig. 1.

4) Compilation strategies: In serverless platforms, we identify three main steps for code to be executed. Uploading the code to the platform, initializing the execution environment, and running it. The latter two are already part of the execution path; they occur when an execution request is actively waiting for the result to be returned. Thus, during initialization, the

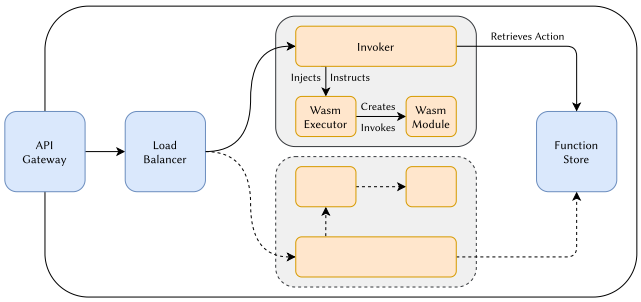


Fig. 1: Invocation flow of a Wasm action. An Invoker *injects* the code into the executor, which *creates* a Wasm module ready for execution. It then *instructs* the executor to *invoke* the module. The result is returned to the Invoker.

action should already be well-optimized for execution, such that as little preparation as possible is needed. Wasm is a binary target format, however, not one that is ready for *fast* execution. It can be interpreted right away–thus a small cold start time–but execution performance will lag behind compared to native code. Compilation to native code is thus essential for good runtime performance, but it also takes time. Given this trade-off, it might seem that JIT compilation should be the best model. However, we measured that to build a simple, WASI-enabled module (1,6 MB) with a JIT compiler takes 35 ms on our test machine (or 40 ms when optimizing for speed). This represents 98% of the entire setup time, including every other part of the runtime. Thus, unsurprisingly, the cold start is defined almost entirely by the compilation phase. For comparison, the cold start time of the `hello-world` Docker image on the same machine is 452 ms. This is roughly in line with the cold start times of AWS Lambda or Google Cloud Function. While the JIT cold start is an order of magnitude faster than the startup times of Docker containers, *it is still on the same order of magnitude as the execution time of some functions.*

In contrast to browsers, which receive JavaScript or Wasm just-in-time, and thus need to compile it in the same manner, a serverless platform takes ownership of a module earlier. When a user uploads a module, the platform has much more time to apply optimizations to the module than when on the execution path, thereby making it ready for a fast startup *and* execution, solving the previously described dilemma. The key is to run the expensive module creation *ahead of the time* of execution. We refer to this as *precompilation*. Once the module is precompiled, we can serialize it and store it in the database. During initialization, all that is left to do is to deserialize the precompiled bytes into an in-memory module; this is a very fast operation. At the time of execution, the module is ready to produce an instance and execution can start immediately.

B. Technology selection

1) Serverless platform: We use Apache OpenWhisk as the framework to implement Wasm support for. OpenWhisk is production-ready, used in the commercial IBM Cloud Functions. It also finds frequent use in serverless research, for example to implement new ideas against cold start latency [8],

as a model of a serverless architecture [10], and to build on top mechanisms to accelerate workflows of interacting functions [26], among others. OpenWhisk also has a *lean* version, suitable for deployment on resource-limited edge devices, such as the Raspberry Pi (RPi) on which we evaluate our system. Furthermore, because OpenWhisk is designed to be easily extensible with new languages, the language runtime protocol is well-refined and straightforward to implement.

2) WebAssembly runtimes: With the rise of Wasm outside the Web browser, a number of standalone runtimes have emerged. Our system supports three of them out of the box: **Wasmtime**[5] is a runtime for JIT-compiled code, available for the x86_64 and aarch64 ISA. It is written in Rust and thus easily embeddable from there. Support for the latest WASI standards and future proposals for Wasm are implemented. **Wasmer**[6] is a mature runtime for x86_64 and aarch64. It offers three different compilers, `singlepass`, `cranelift` and `LLVM`, whose compilation times get slower and execution times get faster, in that order. The runtime has a JIT and a native engine. Support for high-quality AoT compilation with `LLVM` and fast startup times make `wasmer` a promising option. **WebAssembly Micro Runtime (wamr)**[7] supports all compilation strategies. The binary of the AoT runtime is claimed to feature near-native speed at just 50 kB. For execution on edge devices, where memory is scarce, this is advantageous.

Other candidates include Lucet,[8] and WASM3,[9] a Wasm interpreter for embedded devices. The former is currently only available on the x86_64 ISA, an important limitation since at various edge scenarios SBCs of other architectures (e.g., RPi) prevail. The latter supports interpretation only.

IV. WOW: A WebAssembly container runtime for OpenWhisk

A. Approach

Implementing our own Executor allows us to integrate Wasm into OpenWhisk with fewer changes. Furthermore, the more precise control over container management, as well as compilation strategy and execution, are the main arguments in favor of this approach. We therefore introduce our own layer between OpenWhisk and different Wasm runtimes which enable the execution of Wasm modules. This layer implements the parts independent of Wasm runtimes, such as OpenWhisk communication. Thus, each Wasm runtime results in a separate Executor binary and can be used independently of the others. In this architecture, the Docker daemon of vanilla OpenWhisk is replaced, but our design still leverages the existing OpenWhisk interface for container management.

We decided against the alternative to use Kubernetes to manage Wasm modules, because it is incompatible with various aspects of the OpenWhisk execution model, leads to increased Wasm serverless function complexity/code size (Wasm modules should implement themselves the OpenWhisk protocol), and risks cold starts due to Kubernetes pod management operations.

B. Interfacing with OpenWhisk

Following the architecture of Fig. 1, we modify OpenWhisk's Invoker such that it communicates with the Wasm Executor instead of the Docker daemon. Fortunately, the Invoker is already well-separated from the concrete containerization technology through a *Service Provider Interface* (`ContainerFactory`, in OpenWhisk terms), which abstracts the underlying container platform. The layer we introduce implements this interface (`WasmContainerFactory`), and OpenWhisk can be instructed to use it via a configuration property.

When the Invoker receives a request to start/destroy a container, `WasmContainerFactory` relays this call to the respective endpoint of the Wasm executor. The result is returned as a `WasmContainer`, i.e., an object following the OpenWhisk container abstraction. Via this OpenWhisk container interface, the Invoker can access the `/init` and `/run` endpoints it exposes, to initialize and execute a function.

Notably, in OpenWhisk's Docker-based implementation, OpenWhisk only uses the Docker daemon to create a container, but then communicates with the container directly. In our implementation, every request is proxied through the Wasm executor to the container itself. This design presupposes that the Wasm executor can handle a large amount of concurrent requests and is thread-safe. The advantage is that not every Wasm module needs to implement the OpenWhisk protocol, thus leading to more lightweight Wasm modules.

C. Wasm Executor internals

The Executor is written from scratch in Rust. The Rust compiler uses `LLVM` as its backend, so Rust programs can be compiled for any architecture we care about, in particular also aarch64 and wasm32-wasi, the Wasm WASI target. The Executor architecture is shown in Fig. 2 and is detailed below.

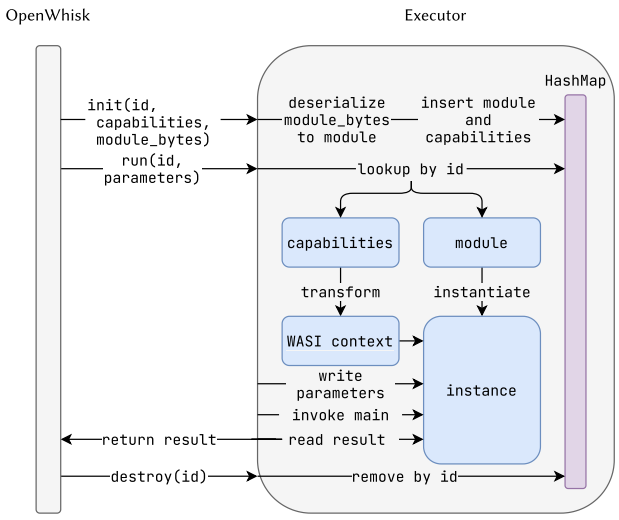


Fig. 2: Overview of the Executor architecture.

[5]https://github.com/bytecodealliance/wasmtime
[6]https://github.com/wasmerio/wasmer
[7]https://github.com/bytecodealliance/wasm-micro-runtime/
[8]https://github.com/bytecodealliance/lucet
[9]https://github.com/wasm3/wasm3

1) Executor entry point: The entry point into our Executor is through HTTP endpoints. We use `tide` [27], an asynchronous by-default web framework ideal for prototyping. One of the implemented Wasm runtimes is selected at Executor compile time. The runtime is used as *state* and is shared among requests. Sharing objects requires them to be thread-safe. This is where Rust's compile-time checks help us ensure this property. Each request receives a copy of the runtime. For this, the runtime wrappers implement a shallow copy method for performance reasons, and our framework ensures thread safety.

To start a container, the `WasmContainer` object in OpenWhisk generates a new UUID, without having to communicate with the Wasm Executor at all. This is possible because there is no setup for a Wasm container, that needs to happen ahead of time. That shows an important aspect of what constitues a Wasm container. In-memory, it is ultimately only represented by the action's code. No external operating system resource is used. It follows that actual work only happens in the endpoints, which are parameterized by the container ID.

`/init` is called once by OpenWhisk per container to initialize the container with the module's code. Although different for each runtime, in general, this endpoint will do any work that can be frontloaded. `/run` is invoked potentially many times, so any work that is not frontloaded would multiply there. Once OpenWhisk decides to destroy the container, it does so through `/destroy`. Of course, `/init` is the potential culprit for the cold start latency, so our focus is on reducing the amount of work this endpoint needs to handle.

Note that unlike Docker-based OpenWhisk, we do not keep instances (containers) in-memory while warm, but only the modules (images), because the cost of instantiating a function from a module, i.e., the overhead on every warm invocation, is negligible (340 μs on average on our test machine to set up a `wasmtime` instance and the parts needed for execution). Moreover, two instances can execute concurrently, even if the module itself is not thread-safe, for instance when re-entrant execution would be unsafe. Finally, implemented this way, it is also safe to execute two requests from different tenants, since instances are isolated from each other. They operate in their own memory space and have their own WASI environment.

2) The WasmRuntime abstraction: The Wasm Executor is generic over the underlying Wasm runtime for reasons of extensibility. To that end, we abstract the common runtime tasks into a trait–essentially an interface in Rust terminology. The state of our Executor can be any object that implements this trait. For each Wasm runtime, we create a wrapper implementing the `WasmRuntime` trait. All wrappers work in similar ways and differ only in internal details.

a) Initialization: The Executor decodes the `base64` string given by OpenWhisk and unzips it, before calling the `initialize` method with the result. This initializes the container identified by the given ID. All wrappers store the code they are given in a thread-safe HashMap. In contrast to the idiomatic Rust APIs of the `wasmtime` and `wasmer` runtimes, where memory- and thread-safety properties are encoded in the type system and checked by the compiler, we used Rust bindings to embed `wamr`, itself written in C and not providing thread safety, thus being more challenging to integrate.

b) Execution: All used runtimes support WASI. WASI follows a capability-based security model, which allows for fine-grained control over what the module has access to. Hence, building a so-called WASI context means setting up those capabilities. That might include writing parameters to `stdin`, receiving logs from `stdout` or setting environment variables and `argv` arguments. By default, the module also has access to WASI APIs like `random_get` for high-quality randomness or `clock_time_get` to access various clocks. WASI also controls on a per-file basis what directories the module has access to. Specifically, the module needs a preopened file descriptor in order to access files. Our wrappers therefore have to open the files and pass them to the module. However, what files a module should have access to is not definable in the OpenWhisk protocol, since it does not make use of Docker's host-to-guest mappings, so we need a way for users to specify these capabilities. OpenWhisk actions can have optional annotations attached to them, which are simple key-value pairs. With some minimal code changes to OpenWhisk's container abstraction, these annotations are then passed to the Executor during initialization. On the Executor side, they are passed to each runtime wrapper, which can then act upon these data. It would be easily possible for a service provider to implement policies on top of this mechanism.

The `run` method executes the module associated with the given container ID and with the given parameters as input. As required by OpenWhisk, the parameter is an object in JavaScript Object Notation (JSON). In summary, `run` looks up the previously stored module or its bytes by the container ID, creates a WASI context with the associated capabilities and potential imports, passes the parameters and calls the module.

c) Cleanup: The `destroy` function removes the Wasm container, which simply means freeing the memory taken up by the module. This is fast and thread-safe.

D. Serverless function development

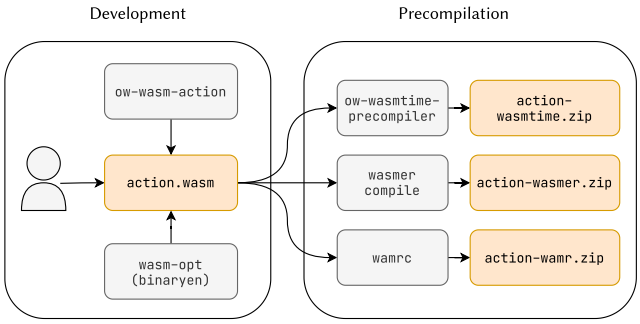


Fig. 3: Serverless function creation workflow.

Developing a serverless function for WOW takes place in two phases (Fig. 3). First, after a function has been implemented, it is compiled to Wasm and an optimization pass is run on the produced code, since not all runtimes do

so themselves. This makes use of the `binaryen`[10] toolchain as a Wasm-to-Wasm optimizer. Second, depending on what runtime is compiled into the Executor, the corresponding precompiler is used to a produce code ready for execution on the target ISA. For `wasmtime`, we have written `ow-wasmtime-precompiler`, a simple wrapper around `wasmtime`'s API, that creates a module from Wasm bytes, in turn kicking off the internal compilation process. `wamrc` and `wasmer compile` can be used for the others. Finally, the resulting precompiled binary is zipped and is ready to be uploaded to OpenWhisk.

A point worth noting regarding passing parameters to a module is that Wasm currently only supports numeric data types natively. Supporting the exchange of higher-level ones, such as strings, is an ongoing process with WebAssembly Interface Types.[11] To work around the problem, actions written for our Executor can use our `ow-wasm-action` library, which allows exchanging arbitrary byte buffers between the runtime wrapper and the Wasm module. Conveniently, a developer only has to implement a function which receives and returns JSON objects. The internals of passing parameters are completely abstracted from action developers, who only need a single line of code to be included.

V. Evaluation

We evaluate WOW experimentally by comparing it to vanilla OpenWhisk. Our purpose is to quantify the performance benefits of Wasm runtimes for serverless function execution at the edge and beyond, and study the performance characteristics of different Wasm runtimes vs. Docker, the current practice in serverless.

A. Methodology

1) Workload types: A serverless platform ought to handle well both CPU- and network I/O-bound workloads. We present experiments with both workload types in isolation, as well as realistic mixes drawing from the literature (see § II-A).

We write test actions in Rust and compile them to wasm32-wasi, and aarch64 or x86_64 respectively, depending on whether we test a Wasm or Docker Executor and which ISA we test on. In order to execute the native binary in OpenWhisk, we use its "black box" feature. It is implemented by the `dockerskeleton` image, which lets us execute any action that adheres to a JSON-in, JSON-out protocol. We follow this approach instead of the official Rust runtime support of OpenWhisk, as the latter requires compiling the function during initialization, massively contributing to cold start latency for Docker. This makes for a more fair comparison between both container runtime types, and renders our results more generalizable to other serverless platforms.

a) CPU-bound functions: For this workload, we repeatedly hash a byte string in a loop using the `blake3`[12] algorithm. Hashing is CPU-bound and free from system calls, so it is a good candidate for this workload type. We choose the number of iterations such that the completion takes roughly 100 ms in a native binary on the respective hardware, in order to have a non-trivial amount of work per invocation–significantly more than what OpenWhisk needs for its internal scheduling.

b) I/O-bound functions: For this workload, we want to measure the effect of a blocking operation, such as an HTTP request. However, networking support in WASI is work in progress. Instead, we simulated the perceived effects of an HTTP request with a 300 ms sleep system call. The latter native host function is supplied to the Wasm module by the Executor via an import.

2) Hardware classes and configuration: Rausch et al. [25] overview different edge computing scenarios and argue that there is significant heterogeneity in edge infrastructure and capabilities. E.g., SBCs can be considered typical hosts for edge computing workloads in smart city scenarios, such as urban sensing, while telco-driven deployments following the Multi-access Edge Computing model [28] feature more powerful server-class hosts in edge data centers. For a more complete view of Wasm-powered serverless performance, we run our experiments both on a RPi Model 3B as an exemplary SBC, and an x86_64 server-grade host. The RPi has 1 GB of RAM and it runs the 64-bit Raspberry Pi OS. The server has 8 GB of RAM, an Intel Xeon E3-1231 v3 CPU (3.40 GHz) with 4 physical cores and 8 logical threads, and a Samsung SSD 850 EVO, where Docker images are stored and loaded from. It runs Ubuntu 20.04.2 LTS. Due to difficulties in deploying OpenWhisk on the RPi with Kubernetes and ansible (default), we use its standalone (*lean*) Java version for evaluation, which is tailored to resource-constrained devices. Since we are not benchmarking OpenWhisk itself, but rather the underlying container technology, we do not see this as threat to validity.

3) Evaluation metrics and measurement methodology: Our metrics of interest are latency and function execution throughput. To measure latency, we utilize OpenWhisk's *activation record*, which is a collection of data resulting from each action invocation. User-perceived latency is composed of `waitTime` (accounts for OpenWhisk internal overheads and for provisioning a container; the latter only applies to Docker), `initTime` (function initialization), action execution, and the overheads to receive a request and return the result. We define cold start latency as `waitTime + initTime`. Measured this way, the cold start time is simply the time between OpenWhisk receiving the request and the container being ready for execution, independent of the underlying container runtime. We devise the following experiments:

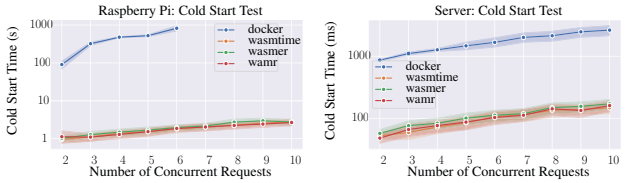
1) To measure cold start times, we first configure OpenWhisk's deallocation time as 10 s. We can then send requests at intervals exceeding 10 s to always trigger cold starts. To measure the effect of concurrency on cold starts, we send $i = 1, ..., N$ concurrent requests. For each i, we wait for OpenWhisk to destroy all containers before continuing. That results in exactly i concurrent cold starts per iteration.

2) We run concurrent requests under various workloads consisting of either CPU- or I/O-bound actions, or mixes thereof.

[10] https://github.com/WebAssembly/binaryen

[11] https://hacks.mozilla.org/2019/08/webassembly-interface-types/

[12] https://crates.io/crates/blake3



(a) (b)

Fig. 4: Cold start times (a) on a RPi and (b) on a server-class host for Docker and Wasm-based container runtimes. On a server-class host, Docker cold starts take 869-2873 ms; Wasm executors reduce this to 47-180 ms. The same behavior shows up on a RPi, but values are orders of magnitude higher.

For the latter, we also simulate request patterns following the findings of [15] about Microsoft Azure.

3) We measure the code size of actions and the amount of memory they consume while being held in-memory, ready for execution.

We used the latest available stable release of OpenWhisk, either the vanilla one, or the standalone version, which we extended with support for Wasm. The `wsk` command line tool was used to create test actions. A separate host connected over 1G Ethernet was used to send requests to the test machine.

B. Cold start performance

We first evaluate cold start latencies in the face of concurrency. We create N CPU-bound actions with the same code. On the RPi, we increase the swap memory to 1024 MiB to be able to test up to 6 concurrent cold starts with Docker; CPU and memory limitations did not allow us to test beyond that point. This is reflected in Fig. 4a, which shows the Docker-based serverless platform to be very slow to cold-start, having orders of magnitude higher latencies than any of the Wasm executors. Even at 2 concurrent requests, Wasm Executors already take $\sim$1 s to cold-start. Wasm runtimes are similar in their startup performance, particularly `wasmtime` and `wamr`, even though they are written in different languages and use different compilation strategies. *The Wasm executors have, on average, less than 0.5% the cold start time of Docker.*

On an x86_64 server-class machine (Fig. 4b), while cold-starts are, as expected, significantly faster, the relative performance among runtimes is similar. Docker cold start slatencies suffer under rising concurrency and vary more. *The average reduction in cold start time compared to Docker is 94% for `wasmtime` and `wamr` and 93% for `wasmer`.*

These results give a holistic picture of the performance of both runtime types in the OpenWhisk context. We then turn our attention to how Wasm runtimes compare. Fig. 5 shows the cold start latencies for the same experiment, but only reporting the duration of a call to the `/init` endpoint of the executor (`initTime`). This excludes the time taken due to OpenWhisk specifics and isolates the overhead introduced *only* by the Executor. Thus, it allows for a more general Wasm container runtime comparison beyond OpenWhisk. Independent

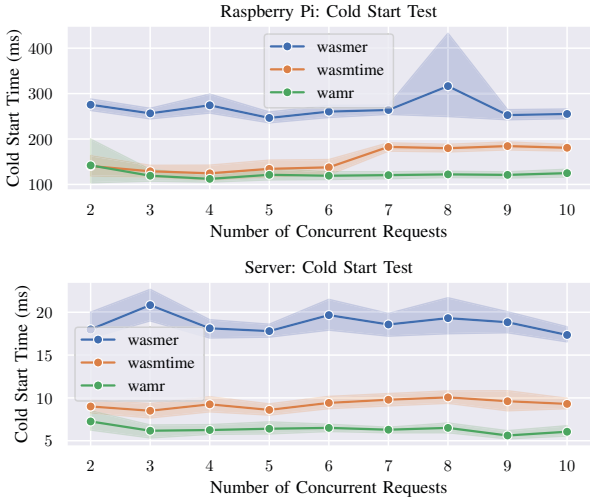


Fig. 5: Cold start latencies for the three Wasm runtimes exluding OpenWhisk internal overheads (only the time spent at the /init endpoint of the executor is reported). Mean latency values are shown with 95% confidence intervals.

of the hardware, `wasmer` is about 2$\times$ slower than `wasmtime` and 3$\times$ slower than `wamr`. The `wasmer` Executor has an inherently higher cost for deserialization and features slightly more performance variability. *Overall, average pure cold start times for Wasm Executors are within 112-274 ms on a RPi.*

C. Function execution throughput

Our second set of experiments aims to evaluate the function execution throughput of OpenWhisk for the different runtimes in question. We experiment with different workload types.

1) Mixed workload: We set the concurrency limit parameter of OpenWhisk to 10, such that 10 actions can be executed concurrently in the same container. We run one iteration of this test before starting the measurements to pre-warm containers and exclude cold starts from our results. The mixed workload was made up of an equal amount of I/O- and CPU-bound actions, and was generated using Apache JMeter.[13] We gradually increased the workload until saturation, and found that for such a function mix, Wasm runtimes consistently outperformed Docker both on a RPi and on a server-class host. The gains of each Wasm runtime over Docker are shown in Table I. In absolute terms, Docker could handle $\sim$6.5 requests/s on a RPi and $\sim$20 requests/s on the server-class host in our tests. The gains of Wasm are more pronounced on the RPi, indicating that Docker is less suitable for resource-constrained edge devices than for server-grade hardware. Among our Wasm executors, `wasmer` fares the best, because we configured it to use the `LLVM` compiler toolchain, which produces high-quality native code. While `wamr` also uses `LLVM` for AoT compilation, its API does not allow us to thread-safely initialize the module once and run it multiple times, forcing us to initialize it from the raw bytes on every `run` call, which affects performance.

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

TABLE I: Average throughput gain of Wasm executors over Docker, in a mixed workload scenario.

Runtime	Raspberry Pi	Server
`wasmer`	4.2×	3.0×
`wasmtime`	3.8×	2.2×
`wamr`	2.4×	1.6×

2) I/O-bound workload: To study the ability of each runtime to handle concurrent network requests, we emulate an I/O-bound workload as described in Section V-A1, and execute tests on the server-grade host. As in our previous experiment, we pre-warm functions to eliminate cold-start effects. The `dockerskeleton` we use in vanilla OpenWhisk performs poorly for I/O-bound workloads. We posit that this is a limitation of the action proxy used in `dockerskeleton`. We configure our setup to handle a maximum of 12 concurrent requests (4 containers; concurrency limit set to 3). The throughput for Wasm runtimes peaked at approximately 40 requests/s, while for Docker at less than 15 requests/s. A higher concurrency limit could increase throughput, but would not change the results qualitatively. Since blocking takes the same amount of time in every Wasm runtime, this performance is mainly determined by the Executor itself and its threading model. Other than OpenWhisk limitations, performance rests on the number of threads that can be spawned on the system.

3) CPU-bound workload: Pure CPU-bound workloads are where Docker outperforms Wasm. In an experiment with the same configuration as the previous one, but for the `hash` serverless function, we found that on average, the Wasm executors achieve only 39% (`wamr`), 57% (`wasmtime`) and 88% (`wasmer`) of the throughput of a Docker runtime. This shows that for a steady stream of highly concurrent CPU-bound requests, Docker is the best option, particularly if the container is likely to be reused many times. This comes perhaps unsurprisingly, given that the executing code is Rust compiled to a native x86_64 binary. There is hardly a way to generate faster code. One could argue, though, that a fully CPU-bound Wasm module reaching 88% of the performance of a native binary in Docker, but with much reduced startup costs, is an acceptable compromise.

As a sidenote, using native code in serverless functions is not particularly popular. In different and probably more realistic settings, performance would vary. To put this into perspective, we ran an experiment to demonstrate the potential improvements of Wasm over today's prevalent serverless runtime: `node.js`. We wrote a CPU-intensive function (sieve of Eratosthenes) in AssemblyScript, a strict variant of TypeScript (typed JavaScript; a popular option to write code that compiles to JavaScript, also supported in various serverless SDKs). This compiles both to Wasm and to JavaScript. We then executed it in a `wasmtime` and a `node.js` OpenWhisk runtime. `Wasmtime` achieved more than 3× the throughput of `node.js`.

D. Latency distribution for non-uniform workload mixes

We perform an experiment on a server-class host with a workload mix characteristic of real-world commercial serverless platforms, in line with [15] but at a smaller scale. In our case, 90% of the requests are for two functions, and each of the remaining 10% concerns one of 60 functions selected uniformly at random. Half of the functions are CPU-bound. Fig. 6 shows the Empirical Cumulative Distribution Function (ECDF) of response time.

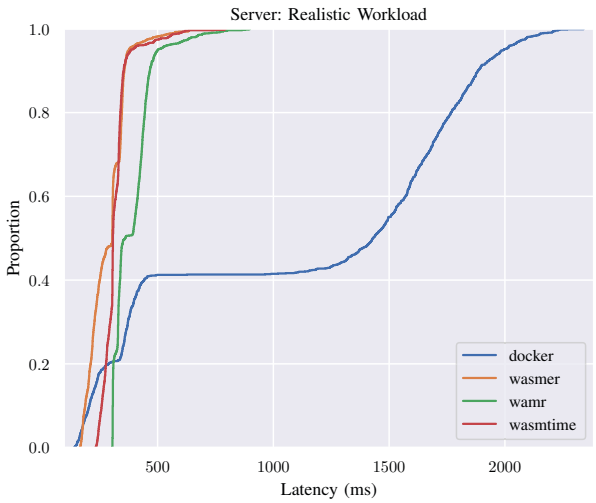


Fig. 6: Latency ECDF of an experiment designed based on serverless usage data from Microsoft Azure [15]. The y-axis shows the proportion of total requests that were finished with the latency given on the x-axis, or less.

For Docker, around 40% of the requests completed in less than 500 ms (warm starts), but 60% of the requests were only finished in less than 1500 ms. The pure function execution times are always less than 300 ms, because we configured them so, while we know from Fig. 4b that the average cold start time of Docker for 4 concurrent requests, as in this experiment, is 1269 ms. Once cold starts are occurring, they have a detrimental effect on the overall system performance as indicated by the increasingly higher latencies beyond the plateau. Only 20% of invocations handled by the Docker runtime finish faster than a Wasm executor. These correspond to CPU-bound warm starts. Cold-starts, on the other hand, do not manifest themselves clearly in Wasm executor curves.

E. Memory use

We run our cold start test again and measure the amount of memory for each container once the actions have finished execution, but before OpenWhisk removes them. At this point, Docker containers have been paused by OpenWhisk. For Wasm executors, the amount of consumed memory is critically affected by the size of the Wasm module (the container *is* the Wasm module in memory, plus some supporting data structures). As Table II shows, Wasm can help keep more containers warm, which can be important to reduce latencies when serverless functions execute in resource-constrained environments. For our minimal hash example function (compiled either to Wasm or, in the case of Docker, to a native binary), 3.5×-7.6× as many functions can be kept alive in the same amount

of memory by Wasm executors. Note that this comparison works in favor of Docker: At measurement time, the function invocation has finished so the corresponding function code is not in memory; the reported values are pure overheads of the `dockerskeleton` image (which is minimal) and the action proxy service that it implements to expose `/init` and `/run` endpoints. Other Docker images used in practice (e.g., node.js) could need more memory.

TABLE II: Memory usage per warm container.

Runtime	Memory consumption (MiB)	Gain vs. Docker
`Docker`	22.28	
`wasmer`	4.02	5.5×
`wasmtime`	6.34	3.5×
`wamr`	2.93	7.6×

VI. Related Work

There are two main approaches to reducing cold starts:[14] (i) introducing optimizations to reduce the likelihood of cold starts in the first place (e.g., selective pre-warming), or (ii) making cold-starting cheap, e.g., via Wasm runtimes.

A. Complementary approaches

Pre-warming: Lin and Glikson [29] implement a solution where a request is served by a pre-warmed Kubernetes pod from a pool. In OpenWhisk, the platform operator needs to configure the pre-warming system by specifying which containers are pre-created, potentially depending on the anticipated load type (e.g., primarily JavaScript functions) [30]. More sophisticated function caching policies are possible [31]. Pre-warming a sufficient amount of containers may require a significant amount of memory. This is especially problematic at the edge. Our approach is shown to improve on that.

Pre-creation: Mohan et al. [8] argue that setting up a Docker container's network namespace accounts for more than 90% of the startup time. Since the kernel uses a single global lock for this task [32], performance declines under concurrency. They thus propose to pre-create a pool of *pause containers*, whose initialization is paused after network namespace creation. A pause container from the pool is attached to a Docker container initialized to execute a function, thus sharing the respective namespace and avoiding the latter's creation at cold-start.

Prediction: Shahrad et al. [15] propose a policy that learns function invocation frequencies and adaptively decides when to pre-warm a function and how long to keep it warm. Such decisions may be driven by AI-based mechanisms, such as reinforcement learning [33], [34]. While pre-warming is less important when cold starts are cheap, having a keep-alive time based on the function's invocation pattern still reduces memory consumption compared to a fixed policy. Thus, combining such predictive methods with a Wasm-based runtime is promising.

Client-centric approaches: Bermbach et al. [35] show that client-side (i.e., outside the FaaS platform) middleware co-deployed with a composition of functions can proactively invoke their cold starts exploiting knowledge of the composition structure. Such approaches can also work in cross-cloud settings, which are becoming increasingly relevant [36].

[14]Notably, the two approaches can be used in conjunction, and this is an avenue for future work.

B. WebAssembly in serverless computing

Execution in node.js: Wasm's potential for serverless was studied by Hall and Ramachandran [10]. They use the V8 engine's Wasm support within `node.js` as their basis, creating a new V8 *context* per request to load and execute Wasm code. They find Wasm to reach 56% of Docker's throughput for a CPU-bound workload, while our precompiled Wasm reached 88% with `wasmer`. Recall that `wasmtime`, our JIT-configured runtime reached 57% of throughput for this workload type. This confirms that Wasm execution in `node.js` is not the best option for pure speed, due to its JIT compilation strategy. Our AoT approach is expected to reduce cold start times compared with their on-demand one, particularly on resource-constrained edge devices. Additionally, we provide a full-fledged Wasm-based environment for OpenWhisk.

Cloudflare Workers: *Workers* [17] also use Google's V8 engine, but execute user functions in JavaScript–or other languages via Wasm support–directly in a V8 *isolate*. This eliminates the costly startup of a `node.js` process, but still needs to parse and compile function code before execution, which is what our approach eliminates.

Fastly's Lucet: Fastly allows to run Wasm code in its *lucet* runtime [18]. The lucet compiler translates Wasm to native code, after which it can be executed in the runtime. We did not leverage it due to security concerns and platform/ISA support limitations, but applied the idea of precompiling to native code AoT. This payed off: we measured module instantiation times of 340 μs to set up a `wasmtime` instance from a module.

FAASM: FAASM [11] is a serverless runtime using Wasm. One or more functions are executed in a `faaslet`, which uses Wasm's software fault-isolation to restrict memory access to its own address space. FAASM targets a challenge which is significant but rather orthogonal to our work, namely state sharing across functions. It achieves fast cold starts by initializing a `faaslet` ahead-of-time as a snapshot of a function's stack, heap function table, stack pointer and data. This is generated at function-creation time. This is similar to how we use Wasm modules as our templates and produce instances on every invocation for isolation.

Sledge: Gadepalli et al. [19] present one of the few thorough *edge-centric* treatises of Wasm-powered serverless. They propose Sledge, a Wasm runtime with its own compiler, extensively evaluated on edge hosts and scenarios. Sledge focuses on function execution on *single-host* servers (all components of the serverless framework are built to run on the same edge host). While it would be interesting performance-wise to integrate the Sledge Wasm runtime in WOW, its current lack of WASI support and single-host design could make it challenging.

VII. Discussion

A. Wasm suitability for serverless edge execution

Revisiting our research question, we remark that Wasm emerges as a highly promising enabler for serverless edge

computing. This boils down to a number of points.

Performance: Wasm runtimes make cold-starting cheap, without significantly sacrificing on warm-start performance. Wasm facilitates serverless on low-end edge hosts, where cold starts of Docker-based solutions are too slow to be practical.

Language Support: C, C++, and Rust already have excellent support for Wasm as a compilation target, and many others are considered production-ready. Serverless operators thus have the option to polish their Wasm support instead of maintaining multiple (Docker) container images with different language runtimes in different versions and for different ISAs.

Scale-to-zero: While Wasm does not get us all the way to the ideal of scale-to-zero, it gets us much closer. With appropriate keep-alive policies [15], [31], rarely invoked functions may be cold-started with negligible cost, while frequently accessed ones can be kept warm with minimal memory overhead.

B. Limitations and open challenges

Wasm features: Some important Wasm features are proposed but not yet fully implemented. These include interface types, networking support in WASI, multi-threading and atomics or a garbage collector (GC), which will make it easier for GC'ed languages to be compiled to Wasm.

Stateful functions: Our Executor is stateless; each function is executed in a new instance. Some functions, however, may wish to cache data from an external service in memory to improve latency. A fully stateless system does not allow for such optimizations, while caching instances rather than modules and invoking them repeatedly would.

Performance on edge SBCs: While we have seen great leaps in performance on a RPi, we can still witness cold starts of beyond 100 ms. This is faster than current FaaS offerings, but can still challenge some latency-critical applications at the edge. Complementary techniques used in conjunction with our approach can drive startup latency further down.

VIII. Conclusion

In order to enjoy the elasticity, flexibility and cost benefits of the serverless model, and at the same time harness the potential of edge computing, which is crucial for emerging IoT services, performance inefficiencies of serverless platforms need to be tackled. In this paper, we provided solutions to this problem by means of a serverless design powered by WebAssembly runtimes as the underlying execution environment. By adopting a more light-weight runtime environment compared to the standard use of containerization technology, we achieve multi-fold improvements in terms of cold start latency, function processing throughput, and memory consumption, and significantly enhance the processing of serverless workloads on inexpensive and resource-constrained edge compute infrastructure.

References

[1] L. Baresi and D. Filgueira Mendonça, "Towards a serverless platform for edge computing," in *Proc. IEEE ICFC*, 2019.

[2] M. S. Aslanpour *et al.*, "Serverless edge computing: Vision and challenges," in *Proc. ACSW*, 2021.

[3] J. Manner *et al.*, "Cold start influencing factors in function as a service," in *Proc. IEEE/ACM UCC Companion*, 2018.

[4] L. Wang *et al.*, "Peeking behind the curtains of serverless platforms," in *Proc. USENIX ATC*, 2018.

[5] A. Tzenetopoulos *et al.*, "FaaS and Curious: Performance Implications of Serverless Functions on Edge Computing Platforms," in *Proc. ISC Workshops*, 2021.

[6] T. Rausch *et al.*, "Optimized container scheduling for data-intensive serverless edge computing," *Future Gener. Comput. Syst.*, vol. 114, pp. 259–271, 2021.

[7] T. Pfandzelter and D. Bermbach, "tinyFaaS: A Lightweight FaaS Platform for Edge Environments," in *Proc. IEEE ICFC*, 2020.

[8] A. Mohan *et al.*, "Agile cold starts for scalable serverless," in *Proc. USENIX HotCloud*, 2019.

[9] M. Shilkov. When does cold start happen on aws lambda? [Online]. Available: https://mikhail.io/serverless/coldstarts/aws/intervals/

[10] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proc. ACM/IEEE IoTDI*, 2019.

[11] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *Proc. USENIX ATC*, 2020.

[12] P. K. Gadepalli *et al.*, "Challenges and opportunities for efficient serverless computing at the edge," in *Proc. SRDS*, 2019.

[13] W3C. WebAssembly. [Online]. Available: https://webassembly.org/

[14] P. Leitner *et al.*, "A mixed-method empirical study of function-as-a-service software development in industrial practice," *Journal of Systems and Software*, vol. 149, pp. 340–359, 2019.

[15] M. Shahrad *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX ATC*, 2020.

[16] S. Eismann *et al.*, "Serverless applications: Why, when, and how?" *IEEE Software*, vol. 38, no. 1, pp. 32–39, 2021.

[17] Cloudflare. How workers works. [Online]. Available: https://developers.cloudflare.com/workers/learning/how-workers-works

[18] fastly. Announcing Lucet: Fastly's native WebAssembly compiler and runtime. [Online]. Available: https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime

[19] P. K. Gadepalli *et al.*, "Sledge: A Serverless-First, Light-Weight Wasm Runtime for the Edge," in *Proc. ACM/IFIP Middleware*, 2020.

[20] A. Haas *et al.*, "Bringing the Web up to Speed with WebAssembly," in *Proc. ACM PLDI*, 2017.

[21] A. Jangda *et al.*, "Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code," in *Proc. USENIX ATC*, 2019.

[22] L. Clark. Standardizing WASI: A system interface to run WebAssembly outside the web. [Online]. Available: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

[23] D. Kimovski *et al.*, "Cloud, Fog or Edge: Where to Compute?" *IEEE Internet Computing*, vol. 25, no. 4, 2021.

[24] B. Varghese *et al.*, "A survey on edge performance benchmarking," *ACM Comput. Surv.*, vol. 54, no. 3, Apr. 2021.

[25] T. Rausch *et al.*, "Synthesizing plausible infrastructure configurations for evaluating edge computing systems," in *Proc. USENIX HotEdge*, 2020.

[26] J. J. Kotni *et al.*, "Faastlane: Accelerating function-as-a-service workflows," in *Proc. USENIX ATC*, 2021.

[27] A. Turon *et al.* Tide. [Online]. Available: https://crates.io/crates/tide

[28] ETSI, "Group Specification MEC 003; Multi-access Edge Computing (MEC); Framework and Reference Architecture (V2.2.1)," Dec. 2020.

[29] P. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool-based approach," *CoRR*, vol. abs/1903.12221, 2019. [Online]. Available: http://arxiv.org/abs/1903.12221

[30] M. Thömmes. Squeezing the milliseconds: How to make serverless platforms blazing fast! [Online]. Available: https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0

[31] A. Fuerst and P. Sharma, "FaasCache: keeping serverless computing alive with greedy-dual caching," in *Proc. ACM ASPLOS*, 2021.

[32] E. Oakes *et al.*, "SOCK: Rapid task provisioning with serverless-optimized containers," in *Proc. USENIX ATC*, 2018.

[33] S. Agarwal *et al.*, "A reinforcement learning approach to reduce serverless function cold start frequency," in *Proc. IEEE/ACM CCGrid*, 2021.

[34] L. Schuler *et al.*, "AI-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments," in *Proc. IEEE/ACM CCGrid*, 2021.

[35] D. Bermbach *et al.*, "Using application knowledge to reduce cold starts in FaaS services," in *Proc. ACM/SIGAPP SAC*, 2020.

[36] A. F. Baarzi *et al.*, "On merits and viability of multi-cloud serverless," in *Proc. ACM SoCC*, 2021.