Building Reliable Distributed Edge-Cloud Applications with WebAssembly

Franz-Josef Grosch
Corporate Research, Robert Bosch GmbH
Renningen, Germany
Franz-Josef.Grosch@de.bosch.com

Dakshina Dasari
Corporate Research, Robert Bosch GmbH
Renningen, Germany
Dakshina.Dasari@de.bosch.com

Nuno Pereira
School of Engineering of the Polytechnic of Porto (ISEP-IPP)
Porto, Portugal
nap@isep.ipp.pt

Anthony Rowe
Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, USA
agr@andrew.cmu.edu

***Abstract*—WebAssembly has emerged as a high-performance, cross-platform, and polyglot software sandbox, not only for Web applications but also for cloud-native software components. In this paper, we position WebAssembly (or Wasm for short) as an important enabling technology for designing distributed applications across the Edge-Cloud continuum that are characterized by specific concerns like safety, timeliness, and reliability. We provide insights into its monitoring capabilities and propose that lightweight virtualization is a promising tool to address different challenges in designing, deploying, and managing distributed Edge-Cloud applications and enable Edge-Cloud orchestration.**

***Index Terms*—WebAssembly, Edge computing, Reliable Distributed Systems, Cloud Native Applications**

I. Introduction

We are entering a new era in which Edge devices are more capable, with advanced compute, networking, sensing and learning capabilities, enabling the emergence of exciting applications. Examples include autonomous vehicles that can offload functionality to the roadside infrastructure, flexible production lines with software-defined manufacturing and advanced robotics, or augmented reality systems in continuous interactions with the physical environment [1]. There are good reasons to anticipate a paradigm shift in designing and deploying distributed systems at the Edge. In these systems, applications are deployed across an Edge-Cloud continuum (sometimes we simply refer these as Edge-Cloud applications) spanning constrained devices distributed in the physical world, smart devices for end-users, edge devices for network access, edge servers for computation in on-premise or regional locations, and Cloud data centers across the globe. As a result, applications often include various devices, hardware architectures, and operating systems. They also have an extensive collection of distributed software architectures (like n-tier, publish-subscribe, microservices), and are developed, managed and operated using various programming languages and tools. There is a pressing need for methods and tools that address the resulting complexity and achieve a vision of a truly distributed application deployment and management across the Edge-Cloud continuum to realize a given functionality.

The work leading to this paper was funded by the Bundesministerium fr Wirtschaft und Klimaschutz (BMWK), in the research project SDM4FZI, under the grant number 13IK001A.

WebAssembly (or Wasm for short) has emerged as a high-performance, cross-platform, and polyglot software sandbox, not only for Web applications but also for cloud-native software components. The number of popular tools adopting Wasm is already increasing. In recent years, we saw key products like Zoom, Google Meet, Google Earth, and of course all modern web browsers adopting Wasm. Similarly, Cloudflare Workers for serverless computing, AutoCAD's web application, eBays barcode scanner, and the Unity gaming engine are examples of the growing popularity and adoption of Wasm. However, in this paper, we position Wasm as an essential enabling technology to design distributed applications across the Edge-Cloud continuum with a focus on concerns like safety, timeliness, and reliability. These concerns are technically challenging to address when one operates outside the purview of a controlled data centre-like environment, which is a characteristic of Cloud frameworks. Edge frameworks also must consider heterogeneous infrastructure components as a first-order concern to support effective application/workload deployment over diverse platforms. We propose that Wasm can help address many difficulties in designing, deploying, and managing reliable distributed Edge-Cloud applications, coupled with Edge-first resource monitoring and management.

II. What is WebAssembly aka Wasm?

Wasm is a binary instruction format for a stack-based virtual machine. It is a safe, fast, and portable low-level bytecode format that is designed for efficient validation and compilation, and safe execution with low to no overhead. Wasm is an abstraction over modern hardware, making it independent of language, hardware, and platform and applicable far beyond just the Web. It has been designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications. Wasm is the first industrial language that has been designed with a formal semantics from the start, utilizing formal methods that have matured in programming language research over the last four decades.

Wasm development started with low-level languages like C, C++, Rust and Go but has now rapidly expanded in scope. Currently, backends for high-level languages like Java, CSharp, or Kotlin and functional languages follow. The long-term goal is to also compile dynamic languages like Python or Ruby directly to Wasm. Currently they remain interpreted by their runtimes compiled to Wasm. Wasm execution started with support by all major browser engines that enable JavaScript to run Wasm programs, which in turn interact with their environment via the browser.

Another way of executing Wasm are host runtimes as shown in Figure 1, that either interpret or compile just-in -time or ahead-of -time and run Wasm programs. The host runtime acts as an interface between the Wasm programs (called Wasm modules) and the rest of the system. Because Wasm programs execute completely sandboxed, programs need to be given the capabilities to access their environment via suitable functions and global data supplied by their host. With the introduction of the portable Wasm System Interface (WASI) that provides platform-independent, non-Web, and system-oriented APIs, the use of operating-system-like features is simplified and standardized.

To realize reliable distributed applications in the Edge-Cloud context, the underlying ecosystem needs to meet certain requirements: The components constituting a distributed application need to be fast to ensure scalability and real-time requirements. They must be portable across operating systems and hardware architectures, open to be embeddable and have customizable interfaces to external environments, safe to use libraries from different, sometimes untrusted sources, and polyglot to combine components implemented in different, appropriate programming languages. We now enlist the properties of Wasm and compare it against these basic requirements.

Wasm is fast: In all evaluations, compiled Wasm is shown to execute within the range of factor 1 to 2 compared to natively compiled code [10]. More important, its performance

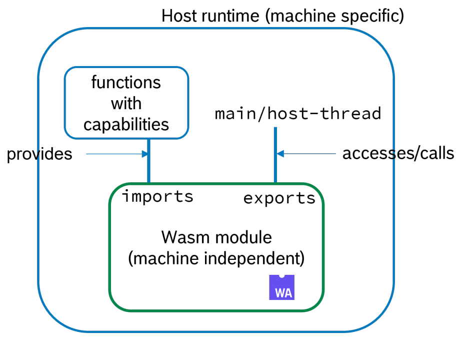
Host runtime (machine specific)
functions with capabilities
main/host-thread
provides
accesses/calls
imports
exports
Wasm module (machine independent)
WA

Fig. 1. Interaction between the Wasm module and the Host Runtime

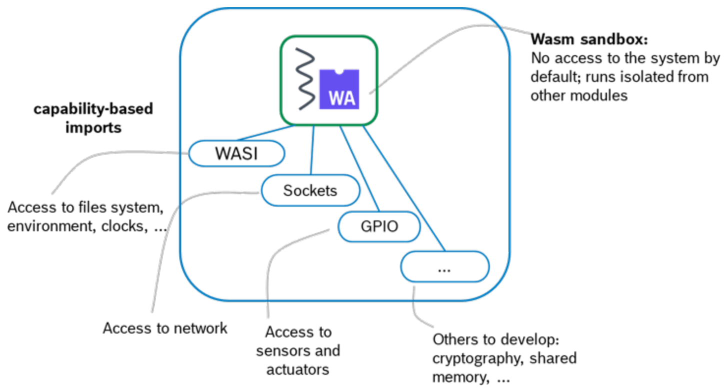
Wasm sandbox:
No access to the system by default; runs isolated from other modules
capability-based imports
WA
WASI
Sockets
GPIO
...
Access to files system, environment, clocks, ...
Access to network
Access to sensors and actuators
Others to develop: cryptography, shared memory, ...

Fig. 2. Access using a capability based model

is consistently fast and predictable. Wasm is close enough to machine code, performant code compiles into Wasm as expected, compilation and optimization happen before execution, and garbage collection is not necessary. This makes even just-in-time compiled or interpreted Wasm suitable for applications with real-time requirements.

Wasm is safe: Sandboxing provides software-based fault isolation in terms of memory safety and control-flow integrity. Wasm prevents accesses to memory outside its sandbox by restricting loads and stores to linear memory and ensures that branches/jumps in the code only go to intended addresses. It prevents buffer overflows with the help of a separate data stack and overriding of function pointers by calling function references only from a table of checked functions. WASI's capability-based security extends Wasm's sandboxing to include I/O. For example if a user invokes a function that needs to access a file, you have to pass in a file descriptor, which has permissions attached to it. By default access is denied [12]. So unlike user-based access control, the program must request access to a particular resource to the runtime. This could be for the file itself, or for a directory that contains the file. In this way, Wasm isolates the system from buggy or malicious effects of untrusted code and untrusted inputs.

Wasm is deterministic: Inspite of the portability claimed by different languages, the effects of hardware variance seep in during execution in different corner cases such as out-of-range shifts, integer divide by zero, overflow or underflow in floating point conversion, and alignment. Wasm provides deterministic semantics to all of these across all hardware with only minimal execution overhead [4]. Thus, a program executes identically on any platform and leaves the sandbox in a (functional) deterministic state after every instruction. Only the float representation for Not a Number (NaN) is hardware-specific which requires normalization when migrating the code and state of a sandbox from one platform to another. Regression tests and redundant safety computations can be run on any platform with identical results.

Wasm is lightweight: It has a compact representation, designed for fast decoding, validation, and compilation. Its safety features allow applications to be run as so-called nano-

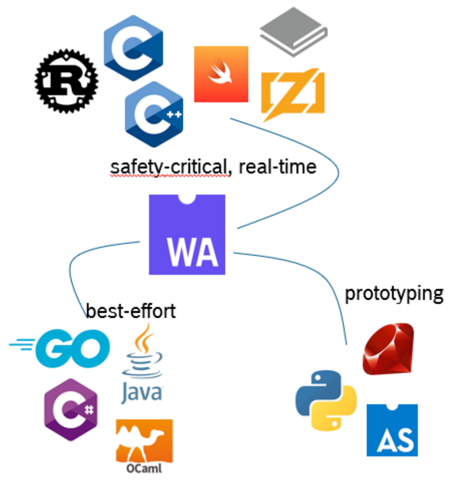

Fig. 3. Many languages can be compiled to Wasm

processes, which combine the isolation of processes with the isolation of containers. Nano-processes allow a very fast cold start and are promising for online migration also on resource-constrained nodes.

Wasm is polyglot: A Wasm binary is called a module. Modules could be compiled from various languages and linked via imports and exports. This allows to combine existing applications, e.g. written in C, with functions written in a different language, like Rust. It opens up several avenues including the compilation of safety-critical real-time applications from low-level languages, or best-effort applications from high-level languages. It also provides the abilitiy to run prototyped applications from dynamic languages and to link applications from components written in different languages. Figure 3 depicts the polyglot nature of Wasm.

Wasm is open: It can be used in different host environments. Several proposals are in progress, which extend WASI with APIs for neural networks, crypto, the filesystem, clocks, entropy sources, and more. Wasm already supports reference types, multi-value returns, bulk data operations, and vector instructions. Soon it will get additional features like tail calls, exceptions, stack switching, multi-threading, and garbage-collection. Compiling domain-specific languages or real-time runtimes directly to Wasm will be easier and efficient.

Wasm is provably correct: Wasm is the first industrial-strength language or Virtual Machine (VM) that has been designed with a formal semantics from the start [4]. It has a complete formalization of static and dynamic semantics, that is machine verified. Furthermore, it does not expose any undefined behavior, preventing bugs that too easily turn into security and safety incidents. To prevent airy wishes for language extensions, Wasm has a rigid proposal process that requires a textual and a formal specification, a test suite, and implementations for the reference interpreter and two production engines. These properties facilitate provably correct compilers and raise the bar for future industrial programming language design, especially important for the safety-critical domain.

III. Edge-Cloud Wasm Framework

Given the properties above, Wasm is a potential technology for building safety-critical, real-time applications. Table I presents a comparison of support technology for Edge-Cloud applications and Figure 4 shows how Wasm as a Virtualization technology, can be used to further ease the development of Edge-Cloud applications. Wasm runtimes can provide isolation comparable to other technologies, while allowing for much smaller system call times, memory footprint and cold startup times. As discussed later in this section, Wasm modules are easy to snapshot and migrate, and do not require to copy the state of an entire VM. While lightweight VM implementations exist [7], these still do not run on smaller resource-constrained devices. Wasm is one of the few well supported technologies that spans Cloud, Edge and Device. With adequate resource management and orchestration, distributed applications can be designed and managed across a network that spans the Edge-Cloud continuum.

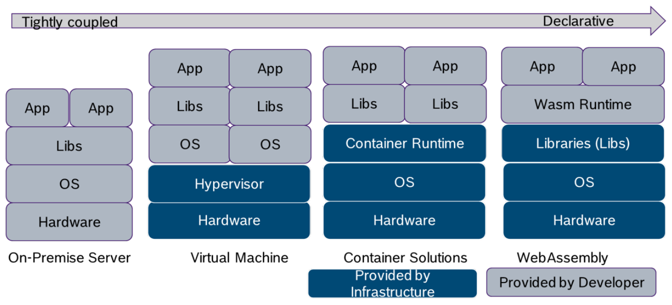

Fig. 4. Evolution: Moving away from Tightly Coupled architectures to more loosely coupled architectures with declarative specifications. [6]

A. *Wasm Orchestration*

Orchestration is an essential component to help manage distributed applications. In the Cloud, frameworks such as Kubernetes are becoming extremely popular. It is natural to think it can be extended to the Edge, and there are efforts in that direction [11]. However, Cloud frameworks tend not to translate well because many assumptions do not hold at the Edge, where systems often interact with the surrounding environment via sensors and actuators. Other efforts extend ideas from the Cloud, particularly serverless computing, to the Edge [3]. These can be appealing when applications are a good fit for this model. Nevertheless, many Edge applications are composed of stateful components that progress through different modes and keep track of the surrounding environment. In both cases, safety, timeliness, and reliability are a concern. Management frameworks for such systems are still immature.

We believe Wasm will play a major role in the future of Edge-Cloud systems and complement Containers in the same way Containers complemented the erstwhile VMs. We imagine distributed applications to be composed of several (single

	WASM	**Containers**	**Virtual Machines (VMs)**
Isolation	Good	Good	Good
System Call Overhead	Good (function call)	Poor (int / kernel)	Poor (int / kernel / HAL)
Portability	Good	Good	Good
Memory Efficiency	Good (KB)	Medium (MB)	Poor (GB)
Cold Startup Time	Good (10us)	Medium (100ms)	Poor (1sec)
Live Migration	Good (seconds)	Poor (n/a)	Medium (minutes)
Legacy Support	Medium	Medium	Good
Targets (Server, PC/Mobile, Embedded)	Cloud, Edge, Device	Cloud, Edge	Cloud, Edge

TABLE I
EDGE-CLOUD SUPPORT TECHNOLOGY.

threaded) modules as shown in Figure 5 that communicate over well-defined channels. Wasm modules run in isolation, in their own separate memory, and are given access only to the specific resources, controlled and monitored by the runtime. In this section, we discuss the characteristics of Wasm that make it a critical enabler of Edge-Cloud orchestration.

Platform Neutrality: As opposed to the very popular Containers, which need to be built for a specific operating system and a specific architecture, Wasm modules are platform-independent. They only need a runtime compiled for the particular architecture. Wasm bytecode format is platform-neutral, and therefore one can compile a web assembly module (from different languages) and then execute it on any operating system and architecture with a Wasm runtime. Since the runtime is responsible for handling how to interact with the platform (file, network interfaces, etc.,), this decoupling makes the module portable, and the module itself does not need to adhere to or handle any specific platform-specific information.

In the bigger picture, this will further ease and accelerate the DevOps process since the developer does not need to build different ”containers” for different environments. A case in point is that Amazon Prime Video [9], a major content delivery service, needs to stream content and push updates to more than 8,000 device types, such as gaming consoles, TVs, set-top boxes, and USB-powered streaming sticks. To avoid needing to develop a separate native release for each of the devices, Amazon uses Wasm (instead of JavaScript) to enable efficient updates while still maintaining performance.

Unlike Cloud computing data centers, characterized by huge clusters consisting of uniform nodes, many heterogeneous devices ranging from small microcontrollers and smartphones to huge servers can act as Edge devices (think of a factory floor in the industrial production use case) and in such a scenario, a platform-independent sandboxing mechanism like Wasm can be highly valuable, in order to deal with the heterogeneity across devices.

Resource-constrained Devices: Edge devices are often resource-constrained and incapable of hosting heavy-weight container environments and yet, we need a sandboxing solution to isolate different applications - even in these scenarios, Wasm is considered as a potential solution as a lightweight sandboxing solution. Wasm runtimes like the Web Assembly MicroRuntime (WAMR) [2] have been specifically designed for resource-constrained devices. WAMR features an Ahead-of-Time (AoT) compiler for near-native speeds, and to enable a smaller runtime footprint. AoT compilation of Wasm is an important feature to support resource-constrained devices, which requires that an external node, such as a gateway, is capable of performing this transformation before sending the program to the end device.

Fine-grained Monitoring: Wasm can enable cross-platform fine-grained monitoring of resource usage both by code instrumentation and by the runtime as it can monitor module access to external resources such as network. This is a particular relevant featire to enable better resource allocation and real-time properties as we discuss later in Section III-B2

Live Migration: While VMs can perform snapshots/restore across a network, and these are limited to similar architectures. Due to Wasm’s memory model, performing a snapshot of the program state becomes easy across heterogeneous devices. With well-defined access to external resources (also facilitated by Wasm), we can ensure that the program can safely resume on a different node.

The capability to perform live migration across the network enables a significantly more agile runtime orchestration that adapts to changing resource usage and mobility. Coupled with their compact size, live migration enables deployment and distribution scenarios without paying an extra price for bandwidth or speed.

Enabling High Density and Multi-Tenancy: Wasm modules are very compact, as opposed to Containers. Containers present a self-contained filesystem to the application by packaging together with the operating system, the required libraries and dependencies for running the application, and so on –This can really bloat the size of the image, and especially for simple applications, this overhead can be really high. In comparison, the idea behind Wasm is not to virtualize the operating system but only the application (process). As a result, the resulting size of Wasm modules is much smaller (in MBs) as compared to Docker containers (in GBs).

Furthermore, many applications deployed on the Edge-Cloud are event-driven and short-lived and therefore, the cost of tearing down and bringing up containers can be very high. In comparison, Wasm with its faster startup times can thus enable dense deployments [3].

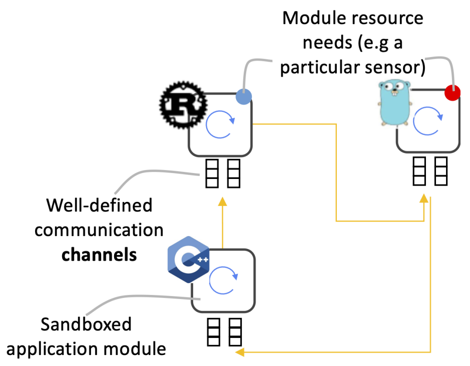
Module resource needs (e.g a particular sensor)
Well-defined communication channels
Sandboxed application module

Fig. 5. Applications are composed of multiple sandboxed modules that communicate with each other over well-defined channels. Each module may have been compiled from a different programming language. The application specification includes the resources it may use, including network, sensors or accelerators.

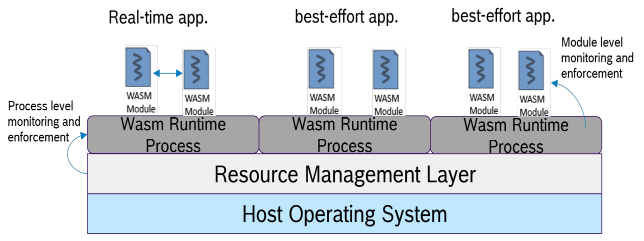
Real-time app.
best-effort app.
best-effort app.
Module level monitoring and enforcement
Process level monitoring and enforcement
WASM Module
Wasm Runtime Process
Resource Management Layer
Host Operating System

Fig. 6. Resource Monitoring and Enforcement to provide differentiated QoS to different applications

B. Wasm to Deploy Real-time Applications?

One of the key questions is whether Wasm can be used to deploy real-time applications. This needs that applications must have predictable behaviour and bounded execution time. Another related aspect is can mixed criticality applications be realized using Wasm on the same platform. For this the underlying Wasm runtime must be capable of providing differentiated services to different applications. This in turn translates to the ability to enforce resource consumption limits to applications deployed as Wasm modules.

Wasm sandboxing allows tight control over the external resources used by the application, such as network or accelerators. A runtime can monitor such usage to enforce policies, and optimize resource usage. When multiple WASM runtimes co-exist an additional resource manager can be used to monitor and manage resource consumption across different runtimes as depicted in Figure 6.

1) Using OS supplied Process-Level Resource Mechanisms: As mentioned earlier, a Wasm runtime is essentially a process that can host multiple Wasm modules where each module is a thread. Different modules may therefore communicate with each other over the runtime. Resource enforcements at the process level are offered by some operating system primitives. For example, the Linux OS offers scheduling primitives like SCHED_DEADLINE to limit the processor consumption to a given process, or SCHED_FIFO which works in conjunction with Linux priorities. Another such mechanism is *control groups* which allows to control resource allocation to a group of processes.

Then, one possible way to realize deploying applications of different criticalities on the same platform, is to assign each application (best-effort, or real-time) to a seperate runtime, as seen in Figure 6 and have multiple runtimes on each node, and enforce resource limits on the runtime level using available process-level resource enforcement mechanisms offered by the operating system. While such a possibility is feasible for single board-computers and Edge servers, smaller resource-constrained devices, that are for example hosted on microcontrollers may not have the capability to host multiple runtimes. For such platforms best-effort and real-time modules will be hosted on the same runtime and therefore we need finer grain control at the module level.

2) Using WASM Monitoring Mechanisms: Many Wasm runtimes (like WasmEdge, Wasmer) provide mechanisms to instrument a Wasm module and also enforce certain resource limits, termed ”Metering” in the Wasm terminology. Instrumentation code is injected into Wasm modules by enabling different metering options (for example to monitor instruction count, or execution time, memory consumption and so on). The metering measures the computation (bytecode instructions executed) for a given module in units of ”Gas” and serves as a representation of the computational time. Given a set of operations (branch, memory or computation, etc.) and a corresponding set of costs for each operation Wasm allows to deterministically run computation for any number of cost units by summing up the costs on the execution of each operation [5]. Furthermore limits can be set to control how long the Wasm code runs. This is also known as ”gas metering”.

With Wasm-Instrument [13], for example, the body of each function is divided into metered blocks, and the calls to charge gas are inserted at the beginning of every such block of code. A metered block is defined so that, unless there is a exception, either all or none of the instructions are executed.

A practical use-case is when Wasm is used for blockchains and smart contracts, which are executed in virtual environments. For example, Ethereum, Gas is the measurement unit for executing operations in the virtual environment, that is called Ethereum Virtual Machine (EVM). Wood et al. [23] present a table with the gas requirements per operation. The amount of gas increases as the complexity of operations in a smart contract increases. For example, an ethereum (ETH) transaction to another agent costs 21000 gas, whereas the deployment of a new contract costs at least 32000 gas to create a contract account.

Metering Memory: The linear memory of Wasm is analogous to the virtual memory provided to processes running on traditional operating systems. It represents a sandboxed region of memory on which the Wasm module may operate on. The runtime stores the static data accessed by the module in an indexed region and manages the stack memory and the dynamic memory required by the application during execution.

Each linear memory section declares an initial memory size (which may be subsequently increased (using the operation grow_memory) and an optional maximum memory size. In order to monitor the memory consumed by the module, the cost of pre-allocated memory is accounted for before instantiating the module. Similarly metering code accounts for the dynamic memory requested by monitoring the grow_memory calls.

To summarize, a combination of traditional resource management mechanisms provided by the OS and fine grain module level monitoring by the Wasm runtime, effective resource management can be realized, which paves the way for differential QoS provisions and allowed applications with different criticalities to co-exist together.

IV. Discussions and Conclusions

Taking into account the promises that Wasm offers in the Edge-Cloud front, the Cloud Native Computing Foundation (CNCF) is a proponent of Wasm in cloud-native infrastructure and therefore hosts several Wasm related projects and initiatives. In the future, we envision a co-existence of different runtime environments (Dockers, Wasm, VMs) to serve a wide range of application requirements. Wasm is still in its development stages (adding support for multithreading, garbage collection, etc.,), however the ecosystem around Wasm is flourishing. We believe that Wasm with its properties of platform and language independence, safe sandboxing and functional determinism will pave the way towards designing reliable real-time distributed applications.

In this work, we presented the key characteristics of Wasm which make it particularly suited for hosting Edge-Cloud applications and as building blocks of orchestration frameworks. We also discussed some inbuilt monitoring and resource enforcement mechanisms which enable the Wasm runtime to provide differentiated QoS to different modules.

References

[1] N. Pereira, A. Rowe, M. W. Farb, I. Liang, E. Lu and E. Riebling, "ARENA: The Augmented Reality Edge Networking Architecture," 2021 IEEE International Symposium on Mixed and Augmented Reality (ISMAR), 2021, pp. 479-488, doi:10.1109/ISMAR52148.2021.00065.

[2] WebAssembly Micro Runtime https://github.com/bytecodealliance/wasm-micro-runtime

[3] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: a Serverless-first, Lightweight Wasm Runtime for the Edge. In Proceedings of the 21st International Middleware Conference (Middleware '20). Association for Computing Machinery, New York, NY, USA, 265279. https://doi.org/10.1145/3423211.3425680

[4] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. SIGPLAN Not. 52, 6 (June 2017), 185200. https://doi.org/10.1145/3140587.3062363

[5] Determining Ewasm Gas Costs, https://github.com/ewasm/design/blob/master/determining_wasm_gas_costs.md

[6] WebAssembly + Cloud Native: A Better Together Story https://cosmonic.com/blog/

[7] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: lightweight virtualization for serverless applications. Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation. USENIX Association, USA, 419434. https://docs.rs/wasm-instrument/latest/wasm_instrument/gas_metering/fn.inject.html

[8] Ethereum Gas Metering https://openethereum.github.io/WebAssembly-GasMetering

[9] How Prime Video updates its app for more than 8,000 device types, https://www.amazon.science/blog/how-prime-video-updates-its-app-for-more-than-8-000-device-types

[10] Yutian Yan, Tengfei Tu, Lijian Zhao, Yuchen Zhou, and Weihang Wang. 2021. Understanding the performance of WebAssembly applications. In Proceedings of the 21st ACM Internet Measurement Conference and Association for Computing Machinery, New York, NY, USA, 533549. https://doi.org/10.1145/3487552.3487827

[11] Kjorveziroski, V., Filiposka, S. Kubernetes distributions for the edge: serverless performance evaluation. J Supercomput (2022). https://doi.org/10.1007/s11227-022-04430-6

[12] Standardizing WASI: A system interface to run WebAssembly outside the web https://hacks.mozilla.org/2019/03/standardizing-wasi-a-WebAssembly-system-interface/

[13] Instrument and transform wasm modules, https://rustrepo.com/repo/paritytech-wasm-instrument