# CWASI: A WebAssembly Runtime Shim for Inter-function Communication in the Serverless Edge-Cloud Continuum

Cynthia Marcelino
c.marcelino@dsg.tuwien.ac.at
TU Wien
Vienna, Austria

Stefan Nastic
snastic@dsg.tuwien.ac.at
TU Wien
Vienna, Austria

## ABSTRACT

Serverless Computing brings advantages to the Edge-Cloud continuum, like simplified programming and infrastructure management. In composed workflows, where serverless functions need to exchange data constantly, serverless platforms rely on remote services such as object storage and key-value stores as a common approach to exchange data. In WebAssembly, functions leverage WebAssembly System Interface to connect to the network and exchange data via remote services. As a consequence, co-located serverless functions need remote services to exchange data, increasing latency and adding network overhead. To mitigate this problem, in this paper, we introduce CWASI: a WebAssembly OCI-compliant runtime shim that determines the best inter-function data exchange approach based on the serverless function locality. CWASI introduces a three-mode communication model for the Serverless Edge-Cloud continuum. This communication model enables CWASI Shim to optimize inter-function communication for co-located functions by leveraging the function host mechanisms. Experimental results show that CWASI reduces the communication latency between the co-located serverless functions by up to 95% and increases the communication throughput by up to 30x.

## CCS CONCEPTS

• **Software and its engineering** → **Abstraction, modeling and modularity**; **Message passing**; • **Computer systems organization** → **Cloud computing**.

## KEYWORDS

WebAssemly, Inter-function, Edge-Cloud Continuum, Serverless, Shim

**ACM Reference Format:**
Cynthia Marcelino and Stefan Nastic. 2023. CWASI: A WebAssembly Runtime Shim for Inter-function Communication in the Serverless Edge-Cloud Continuum. In *The Eighth ACM/IEEE Symposium on Edge Computing (SEC '23), December 6–9, 2023, Wilmington, DE, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3583740.3626611


*SEC '23, December 6–9, 2023, Wilmington, DE, USA*

ACM ISBN 979-8-4007-0123-8/23/12.
https://doi.org/10.1145/3583740.3626611

## 1 INTRODUCTION

The Edge-Cloud continuum leverages many benefits from the Serverless Computing paradigm. Serverless Computing simplifies application development by removing the complexities of infrastructure management from the developer [1, 2, 3]. Recently, the integration of WebAssemly (Wasm) with Serverless Computing enabled enhanced portability, improved performance, and broader language support in deploying serverless functions. Furthermore, in an environment composed of resource-constrained devices such as the Edge-Cloud continuum, Wasm offers many advantages due to its near-native execution speed, security, and reduced cold start. In Wasm, large container images are replaced with small binary files executed in isolated sandboxes. Wasm creates a secure sandbox typically without host access, increasing security and privacy, which leads to additional protection for co-located guest applications [3, 4, 5]. Despite these advantages, Wasm's constrained host access introduces additional challenges to the complex serverless inter-function communication.

**Data exchange between serverless functions.** The increased usage of serverless functions compositions exposed limitations regarding data exchange between serverless functions [2, 6, 7]. Currently, there are two main approaches enabling data exchange between serverless functions:

(a) *Remote third-party services:* Most of the current serverless platforms are function-locality agnostic. They leverage remote services to exchange data between the functions providing flexibility and scalability. Such remote services used for serverless data exchange can be classified as follows: (i) *In-memory* solutions such as key-value stores (KVS) [8, 9] or remote far memory solution [10] provide low latency. Nevertheless, they still rely on third-party services, creating data, memory, and network overhead, making them inadequate for the Edge-Cloud continuum, where resources can be limited. (ii) *Storage-based* solutions are most common approach to exchange data between functions [11]. However, they increase latency by pushing functions to repeatedly download and upload the data. Additionally, they increase costs with remote storage while limiting usage to a single Cloud provider [12, 13]. In Wasm, *remote services* usage became possible after the release of WASI, which introduced POSIX-like calls such as network access [14]. This enables the Wasm binaries to connect to remote services via the host network interfaces. Although remote services enable data exchange between serverless functions, for co-located functions they add unnecessary network traffic, create data and resource overhead, duplicate data serialization on the source and target, and consequently increase latency. Unfortunately, with these approaches, co-located functions cannot benefit from the function proximity since the data exchange is done via the remote services [15]. Furthermore, remote services lead to additional costs in the typical "pay-per-use" model.

(b) *Inter-function communication:* Other approaches enable serverless functions to leverage direct data exchange to decrease latency and avoid extra remote services [6, 16]. Two common approaches for inter-function communication are: (i) *Message Queues* such as SAND [17] leverage publish/subscribe paradigm for inter-function communication. In Wasm, WASI libraries provide network communication enabling message queue communication [14]. Even though message queues enable point-to-point inter-function communication, they still rely on external message brokers. This creates network and resource overhead and dependency on external solutions. (ii) *Shared memory* approaches [18, 19] exploit sharing of the same memory region to enable low-latency data exchange between the serverless functions. For coordination, such approaches use pipes [20], threads [21], or custom software isolation [22, 23]. Although shared memory provides low latency communication, resource sharing between two different functions reduces the isolation offered by containerization [24]. Additionally, all the functions must be started simultaneously, as the shared memory address space must be allocated before the processes start, increasing the memory footprint. In Wasm, *shared-memory*-based communication is typically enabled by statically linking the modules at Wasm function startup. The static link creates a shared memory region in the Wasm Virtual Machine (VM) which makes it accessible between the modules. Once host runtimes explicitly link the modules, Wasm modules can also be reused [25]. Nevertheless, static linking is still unavailable in the current wasm-based Edge-Cloud infrastructure as it requires the modules to be explicitly statically linked via the host function. Current state-of-the-art wasm-based shims enable single Wasm module execution at a time.

**Contributions**. In this paper, we introduce CWASI, a runtime shim that facilitates inter-function communication between wasm-based serverless functions. CWASI is a container runtime shim that leverages Wasm runtimes to provide isolation and security. CWASI optimizes inter-function communication for co-located serverless functions by introducing mechanisms that select the best inter-function communication approach. The main contributions of this paper include:

- *A novel model for serverless inter-function communication* that leverages function locality to enable three-mode communication. This model enables co-located functions to leverage the local host mechanisms to optimize inter-function communication, consequently reducing their dependency on external services and network connections;
- *A WebAssembly Container Runtime Shim* that enables the three communication modes proposed by our inter-function communication model to optimize data exchange between serverless functions. By following the three-mode communication model, CWASI presents a reduction of up to 95% in latency and an increase in throughput by up to 30 times.

**Outline**. The paper has eight sections. Section 2 describes the motivating scenario and our research challenges. Section 3 discusses related work and limitations. Section 4 gives an overview of the CWASI Communication Model and its Shim Architecture. Section 5 describes the mechanisms introduced in CWASI and their usage. Section 6 shows the implementation details. Section 7 describes the experiments and evaluation, and Section 8 concludes with final discussion and future work.

# 2 MOTIVATION

## 2.1 Illustrative Scenario

To better motivate the research challenges, we present a realistic real-time video analytic use case for vehicle path reconstruction. A serverless workflow for vehicle path reconstruction typically involves the deployment of cameras at intersections and along the road. The cameras detect the vehicle by using object detection algorithms and then, serverless functions are employed to identify the vehicle's location in its path. Our workflow features four serverless functions partially executed on the Edge and partially executed on the Cloud. The use case is based on AWS use cases [26] and scientific researches [27, 28, 29]. To decrease the communication latency, large real-time video streams are processed on edge nodes responsible for extracting image frames and other tasks such as labeling and anonymization. In contrast, tasks that require greater computing resources, such as more powerful object detection models, are performed in the Cloud.

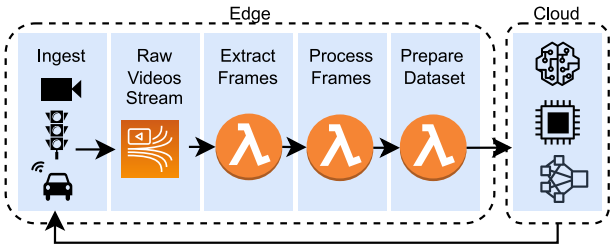



**Figure 1: Serverless Workflow for Image Processing for Vehicle Path Reconstruction (simplified)**

In Fig. 1, the *Ingest* stage represents the real-time videos that the cameras collect and transmit to edge nodes via a streaming framework. *Raw Videos Stream* stage triggers the serverless functions, which are responsible for *Extract Frames* from the real-time videos. To decrease the latency, the real-time videos are processed in small chunks where each *Extract Frames* function instance is responsible for processing a small part of the video. Once *Process Frames* functions have finished labeling and anonymization, they again place the results from *Process Frames* in the storage. In the next stage, the function *Prepare Dataset* retrieves the images from the storage and places the results in the storage again. Further, the workflow continues in the cloud, where the function retrieves the prepared dataset results from *Prepare Dataset* for more resource-intense tasks such as model training for object detection. At the end of our workflow, the trained model is returned to the edge nodes, which can act upon the newly trained model, such as recreating a vehicle path.

CWASI shim detects when *Extract Frames*, *Process Frames* and *Prepare Dataset* functions are running on the same host. Thus, CWASI skips remote storage and applies co-located function communication optimization to exchange data for functions running on the same host. CWASI simplifies the workflow, decreases latency, and reduces financial costs with remote storage services.

## 2.2 Research Challenges

Current serverless platforms have limited support for inter-function communication. In wasm-based serverless approaches, the challenges are even more significant for co-located inter-function communication, given that WASI only allows limited access to the host. We identify the following main research challenges:

**RC-1: How can we enable efficient inter-function communication in wasm-based serverless platforms?** Inter-function communication is crucial for real-time scenarios where edge devices must quickly react upon scenario changes. In our scenario, inter-function communication latency directly affects the reaction time of the path reconstruction which can influence the vehicles' trajectory. In the given scenario, efficiency and latency are critical factors for a reliable workflow. However, current state-of-the-art serverless platforms have remote services as a common approach for exchanging data. This is a burden in the Edge-Cloud environment for composed workflows, which constantly exchange data. Serverless functions must repeatedly download and upload data, increasing resource footprint, network, and data access overhead which leads to a complexity increase of inter-function communication [2, 3, 30].

**RC-2: Can we utilize Wasm static-linking to optimize inter-function communication?** Wasm static-linking enables multiple modules to access the same memory region. Each module has a dedicated linear memory space. However, the modules can access other memory spaces within the same Wasm VM when statically linked, enabling shared memory and Wasm module reuse. To statically link the modules, every Wasm binary must be explicitly added in the Wasm VM by the host runtime before starting it. WebAssembly static linking is still unavailable in state-of-the-art Wasm shims such as RunWasi WasmEdge and Wasmtime [31, 25]. The lack of Wasm static linking exposes a challenge in achieving optimal communication between modules in the same serverless workflow.

**RC-3: Can we exploit function locality to optimize the communication for co-located Wasm serverless functions?** Co-located Serverless functions such as *Extract Frames* and *Process Frames* can profit from the function proximity to decrease communication latency. Function locality awareness enables us to leverage the host mechanisms, avoiding common challenges introduced by network inter-function communication, such as data duplication and redundant serialization on the source and target functions, besides resource and network overhead [22, 30, 3].

# 3 RELATED WORK

## 3.1 Serverless Platforms Models and Shims

CWASI is a WebAssembly runtime shim that interacts with container managers such as *containerd* to manage containers' process lifecycle. Fig. 2 shows how CWASI fits into the existing Edge-Cloud stack. Fig. 2a shows the standard container-based serverless platforms such as OpenWhisk, where every function is packed in a container, and a serverless management framework such as request forwarding is shared among all containers [32, 33]. Container-based serverless offer strong isolation as every serverless function is packaged on its container with dedicated libraries and language runtime. Nevertheless, large container images with a single serverless function lead to a memory footprint overhead and high startup latency. Hence, it is inefficient for the Edge where resources are limited and low latency required [3, 21, 2, 34].

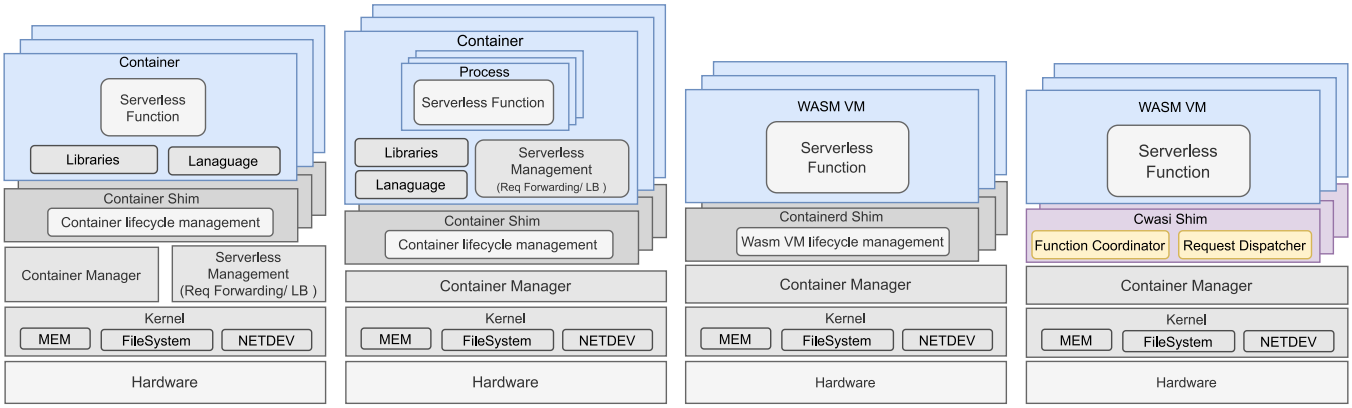
In process-based serverless stack, in Fig. 2b, serverless frameworks such as OpenFaas introduces additional process in the standard container. Serverless functions share container resources such as libraries and language runtime with side-car serverless management processes [35, 36, 16]. For example, in OpenFaas, every function has an additional watchdog process acting as a reverse proxy [30, 37]. The serverless function and the watchdog are isolated processes but share the same container resources. Although process-based serverless platforms decrease the overhead compared to container-based platforms (Fig. 2a), they still rely on containerization, which leads to resource overhead such as memory and cold start issues [3, 22, 2].

Fig. 2c shows how container managers and runtime shim enables Wasm in the current Edge-Cloud infrastructure. Two key features contribute to adopting Wasm outside the web browser: WASI and Host Runtime [38, 14, 39, 40]. WASI provides access to the host machine and, consequently, to the network. Host runtime creates the Wasm VM by statically loading and executing the Wasm binary file enabling Wasm module interaction via imports and exports [14, 41]. Wasm enables near-native speed and decreases serverless cold start issues significantly [42, 43]. As Wasm binaries run on a specific sandbox with restricted host access, it presents limitations for inter-function communication.

We present CWASI approach in Fig. 2d. A serverless runtime shim that leverages Wasm to provide isolation and security while enabling optimized inter-function communication. CWASI follows Open Container Initiative (OCI) specification to integrate Wasm in the Edge-Cloud continuum [44]. Furthermore, CWASI introduces a novel approach to identify and select the best inter-function communication approach available for every serverless function based on the functions' location.

## 3.2 Sandboxes

The state-of-the-art Edge-Cloud platforms leverage the concept of sandboxes to provide isolation between the serverless functions. In the current container-based serverless platforms this is achieved with Linux *namespaces* and *cgroups* [3]. Solutions such as SOCK [45] and Faasm [22] leverage additional sandboxes to isolate functions, *worker*, and *faaslet*, respectively. Hence, functions in the same sandbox can safely share resources. While SOCK leverages IPC, Faasm uses distributed shared memory [23]; thus, co-located functions can benefit from proximity for low-latency inter-function communication. Although additional sandboxes decrease challenges such as cold start and inter-function latency, they add complexity and resource overhead with the additional sandboxes. SAND [17] provides different isolation approaches based on the workflow. Containers isolate different workflows, while processes in the same container isolate functions within the same workflow. SAND enforces more robust isolation for different workflows and leverages container sharing in the same workflow to mitigate cold start issues. Additionally, shared libraries within the workflow must only be loaded once per workflow. Further, SAND's workflow container sharing decreases memory footprint as new functions are isolated by processes instead of isolated containers. Nevertheless, serverless functions are bound to SAND serverless




**(a) Container-based Serverless** **(b) Process-based Serverless** **(c) WebAssembly in the Cloud** **(d) CWASI**

**Figure 2: Overview of Edge-Cloud Serverless Platforms**

framework to have these benefits decreasing portability and flexibility. Sledge [30] leverages the Wasm sandbox mechanism to provide isolation, enabling untrusted modules to be safely executed on the host machine. As Sledge does not use standard container isolation, it is not OCI compliant, which means applications developed for Sledge are limited to a single Wasm runtime. CWASI follows OCI specifications to provide isolation and safety in addition to the Wasm sandbox. OCI specification enables compatibility, flexibility, and scalability within the state-of-the-art container manager and orchestration frameworks. Furthermore, CWASI selects the best communication approach based on function location and it optimizes co-located inter-function communication.

## 3.3 Third-party Services

Third-party services are the most common approach for inter-function data exchange, most specifically remote storage [11]. SAND [17] introduces a hierarchical message bus to provide low latency with a global and a local bus. SAND optimizes co-located inter-function communication by introducing a local message bus responsible for exchanging data between functions on the same host. Nevertheless, SAND's message bus leverages Apache Kafka, which means SAND's approach relies on third-party services to exchange data even for co-located functions leading to unnecessary network traffic and infrastructure overhead since additional systems require additional infrastructure and operational effort. SONIC [46] offers three different approaches for inter-function communication: (i) host storage where co-located functions leverage local storage to exchange data; (ii) direct passing where functions in different hosts exchange data by sending to the next function a data reference, e.g., IP. and file path; (iii) remote storage by using third-party services such as S3 and MinIO. SONIC provides low latency and flexibility by selecting the most suitable approach based on the functions and data location. CloudBurst [9] relies on a local cache on each function host to allow low latency access to frequent data from a remote KVS Anna [8]. Although Anna is low latency and highly scalable KVS, it proposes data exchange via remote third-party service. Regardless of storage or KVS, SONIC and CloudBurst still rely on third-party services to exchange data leading to additional network, resource, and data overhead which leads to duplicated data serialization. CWASI introduces an optimized co-located inter-function communication that leverages IPC to exchange data directly through the host kernel memory buffer, decreasing latency and increasing throughput significantly. Additionally, CWASI enables Wasm static-linking to enable modules reuse and shared memory between Wasm modules decreasing latency.

## 3.4 Co-located Inter-function Communication

Co-located functions that rely on third-party services to exchange data do not profit from the function's placement to exchange data. SAND [17] creates a local bus for co-located inter-function communication. Nevertheless, it relies on a third-party service to transfer the data, i.e., Apache Kafka. ExCamera [47] proposes a long-lived rendezvous server that receives and forwards requests between source and target workers. As inter-function relies on an external component, it adds unnecessary network requests similar to the third-party service approaches. Nigthcore [20] uses shared memory to exchange data and Linux pipes as inter-process communication to notify when data is available for reading or writing in the memory region. Nightcore provides microseconds latency but requires functions to include its runtime library, limiting portability. POCKET [48] enables shared applications with cross-clients by creating an IPC channel with shared memory for low latency and high isolation. Floki [15] leverages sync pipes for co-located inter-function communication and TCP sockets for remote communication. Floki provides low-latency communication by skipping multiple network communication layers and connecting directly to the POSIX layer. Nevertheless, Floki relies on an additional component forwarding agent deployed in the host namespace to create and connect to the TCP socket and to forward data between two functions. As Floki's forwarding agent is an extra component deployed via orchestration tools such as Kubernetes, it means that to be scalable; the forwarding agent needs to be separately scaled in addition to the function, which causes resource overhead in data-intensive scenarios. CWASI introduces a container runtime shim to provide optimized co-located inter-function communication via a local buffer. Since CWASI is a shim, it lives along the function process, avoiding dependencies on additional components. Moreover, CWASI enables seamless Wasm static-linking between modules.

Wasm-based serverless workflows with multiple modules run on a single Wasm VM, decreasing latency and memory footprint while increasing Wasm modules' reusability.

# 4 CWASI MODEL AND ARCHITECTURE OVERVIEW

## 4.1 CWASI Model Overview

CWASI introduces a novel model for efficient inter-function communication between serverless functions. The model exploits function locality, i.e., functions co-located on the same host. Fig. 3 shows the CWASI inter-function three-mode communication model featuring three modes: Function Embedding, Local Buffer, and Networked Buffer. Function Embedding communication model proposes one sandbox for trusted serverless functions. Local Buffer optimizes data exchange through the host resources, while Networked Buffer uses state-of-the-art mechanisms to communicate remote serverless functions. CWASI communication model integrates automatic inter-function communication selection and autonomous provision. CWASI enables inter-function communication by identifying, selecting, and provisioning the most suitable communication mode for each function. Consequently, it allows functions to have an optimized data exchange without external intervention. Next, we discuss the main model abstractions in more detail.

*Function Embedding.* This communication mode groups trusted functions into one sandbox, allowing them to share the same resources. Thus, these functions can directly call each other via volatile memory such as shared memory, heap memory, and registers, leading to high-speed data access. Therefore, this communication mode is the most efficient. Function Embedding relies on CWASI to embed different functions in one sandbox. Thus, it decreases the function isolation. Nevertheless, functions in the same application namespace are considered trusted and do not need strong isolation [17, 38]. In CWASI shim, we leverage Wasm static-linking to embed tightly coupled and fully trusted Wasm functions into one single Wasm VM.

*Local Buffer.* CWASI leverages the host mechanisms to create a Local Buffer that enables co-located functions to exchange data. Hence, co-located functions can communicate without external services or network communication. This communication mode keeps different functions in its dedicated sandbox, enforcing a more robust isolation than Function Embedding. Therefore, the Local Buffer communication mode enables CWASI to execute functions that require higher trust, e.g., functions from different namespaces. During function startup, CWASI detects functions on the same host and creates a Local Buffer for each function.

*Networked Buffer.* Networked Buffer communication extends CWASI capabilities beyond co-located functions, allowing inter-function communication in distributed environments. Network communication is the standard communication between serverless functions. Thus, it provides a simplified deployment without any additional software required. Nevertheless, it increases latency, throughput, and network overhead.

*Locality awareness and model selection.* CWASI inter-function communication model improves data exchange with a mode selection that is transparent to the functions. To achieve this, the communication model must determine the function locality. CWASI model decides between Function Embedding and Local buffer for co-located functions based on the function trust level. For instance, functions in the same namespace are trusted and, therefore, eligible for Function Embedding. In contrast, functions in different namespaces are not trusted thus, only eligible for Local Buffer communication mode. Additionally, functions can give hints via deployment annotations to allow the CWASI model to choose one specific communication mode. Once the CWASI model has selected the best communication mode, it automatically provisions it.

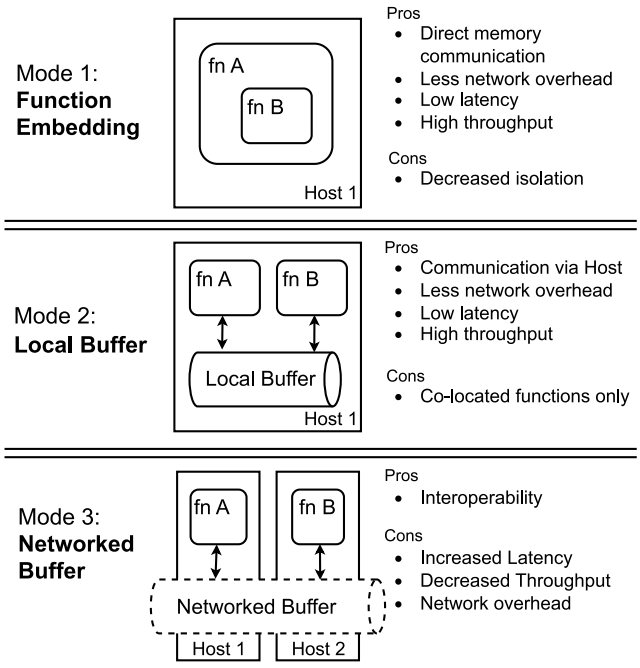



**Figure 3: CWASI Inter-function Communication Model**

## 4.2 CWASI Shim Architecture Overview

CWASI is a container runtime shim that mediates the communication of the container manager with the container runtime. For isolation, it relies on *cgroups* and *namespaces* on the host machine. Additionally, CWASI leverages WasmEdge runtime to create an isolated Wasm VM. Wasm deny-by-default mode provides security by only allowing explicitly requested host access via WASI [22, 14]. CWASI proposes a novel design while maintaining interoperability with standard container managers and Wasm shims, which means CWASI runs alongside existing Wasm shims, e.g., RunWasi shims.

Fig. 4 shows how CWASI interacts with wasm-based serverless functions during startup and function runtime to enable three-mode inter-function communication. We label functions as primary and secondary according to the workflow execution order. For example, given a workflow composed of fnA and fnB, where fnA invokes fnB. In this example, fnA is a primary, while fnB is a secondary function. During the function start, CWASI decides whether to the first, second, or third mode of the CWASI communication model. At

function runtime, CWASI fnA acts as a forward proxy identifying the best inter-function communication option and forwarding the requests. Whereas CWASI fnB acts as a reverse proxy receiving requests and initiating fnB function startup.

Fig. 4 shows CWASI core components: *Function Coordinator* and *Request Dispatcher*. These core components implement the following features: (i) *Function (Fn) Lifecycle*; (ii) *Function Embedding (FE) Discovery*; (iii) *Local Buffer (LB) Receiver*; (iv) *Network (N) Receiver*; (v) *Inter-Function Communication (IFC) Selection*; and (vi) *Local Buffer (LB) Sender*.

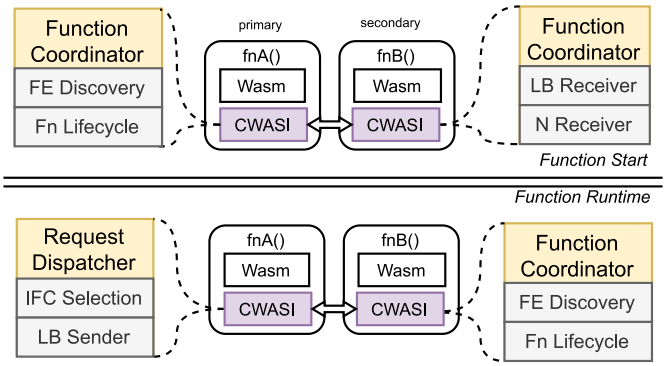



**Figure 4: CWASI Shim Architecture Overview**

*Function Coordinator:* It is a core component responsible to start and maintain the function. This component is an enabler for the three-mode communication. During function start, the function coordinator starts *Function Embedding Discovery* for the first communication mode, *Local Buffer Receiver* for the second, and *Network Receiver* for the third, as displayed in Fig. 4. In addition to the communication-related tasks, the function coordinator manages the *Wasm Lifecycle*. For the task decision-making process, the function coordinator relies on the OCI bundle annotations, as described in Algorithm 1.

In Algorithm 1, let $O$ be the OCI spec which consists of arguments $A_i$, annotations $N_j$, where $i$ and $j$ are indexes that represent the elements in the set $A_i$ and $N_j$, a bundle path $B$, and remaining fields represented by $F$. Let $S$ be a string identifier. If the identifier $S$ exists in the set of annotations $N$, then Algorithm 1 starts a *Network Receiver* with the function name, where the function name is the first element $A_0$ of the argument set $A$. Further, it creates a *Local Buffer Receiver* for function name $A_0$ and bundle path $B$. If $S$ does not exist in $N$, then the manager writes Wasm binary file to WAT, loads the discovered Wasm function modules into Wasm VM function and finally triggers *Wasm Lifecycle* which will start the Wasm VM function with the set of arguments $A$.

(i) *Function Lifecycle:* This feature is responsible for function lifecycle methods such as `new`, `start`, `wait`, `delete`, `kill`. This feature creates and executes Wasm VM and Linux processes. Once the function has finished, it terminates every process related to the specific function. Additionally, it connects directly with the container manager to give updates on the container status.

(ii) *Function Embedding Discovery:* This feature discovers whether tightly coupled additional modules are present on the machine. If such a module is found, it embeds these additional modules in the same Wasm VM function. Hence, during runtime these Wasm modules share the same memory stack, allowing them to communicate directly in the first mode of the CWASI communication model.

**Algorithm 1** Function Coordinator

**Input:** O = ($A_i, N_j, B, F$) where $i, j \in \mathbb{N}_0$ : oci spec
1: S: string primary/secondary identifier
2: **if** $\exists S \in N$ **then**
3: $network\_receiver(A_0)$
4: $kernelbuffer\_receiver(A_0, B)$
5: **else**
6: $wat \leftarrow wasmprinter(B)$
7: $wasm\_modules \leftarrow function\_embedding\_discovery(wat)$
8: $embedd\_modules\_in\_function(wasm\_modules)$
9: $wasm\_lifecycle(A)$
10: **end if**

(iii) *Local Buffer Receiver:* It is a feature that creates a Local Buffer Receiver based on the functions OCI annotations. CWASI shim creates this after identifying whether the serverless function is primary or secondary, e.g., if *fnA* connects to *fnB*, *fnA* does not need a Local Buffer Receiver but *fnB* does. Therefore, we label *fnB* as secondary, and based on the annotations on the OCI spec, the shim can decide whether to create a Local Buffer Receiver and a Network Receiver for this function or not.

(iv) *Network Receiver:* This feature enables the secondary function to receive messages via network communication. In CWASI, we enable network communication with message queues. During the function start, the function coordinator creates a dedicated queue for each function which will during runtime accept incoming messages via network.

*Request Dispatcher:* This core component identifies during runtime which communication model is more efficient for the task at hand, according to the three modes from the communication model presented in Section 4.

(v) *Inter-Function Communication (IFC) selection:* Triggered by *Request Dispatcher* component, this feature identifies during runtime whether running serverless functions are co-located or remotely. CWASI Shim leverages the container manager to learn which functions run on the same host. Further, either it selects a target function on the same host, or it sends the request via the Networked Buffer if the function is placed on a different host.

Algorithm 2 shows how CWASI determines whether the functions run locally or remotely during runtime. In Algorithm 2, $FT$ is the input function target type $FT$ e.g. *fnB*, and container manager running path $RP$, while local buffer receiver path $SP$ is the output. Algorithm 2 iterates every function path $F_P$ of OCI spec file $F$ in running path $RP$. In each interaction, the function reads the OCI config for file path $F_P$. Let $C$ be an object that is composed of a set of arguments $A$ and remaining fields $RF$ obtained from $read\_oci\_config$. If there exists target function $FT$ in argument set $A$ then, set local buffer receiver path $SP$ is equal to the function path $F_P$ plus $S$, where $S$ is a string local buffer receiver suffix.

(vi) *Local Buffer Sender:* After identifying a serverless function on the same host, the *Request Dispatcher* component triggers this feature during runtime. The Local Buffer Sender enables the shim to

**Algorithm 2** IFC Selection

**Input:** $FT$, $RP$: function type and running path
**Output:** $SP$: local buffer receiver path
1: **for all** $F_P \in F, \forall F \in RP$ **do**
2: $C = (A, RF)$, where $C \leftarrow read\_oci\_config(F_P)$
3: **if** $\exists FT \in A$ **then**
4: **return** $SP \leftarrow F_P + S$
5: **end if**
6: **end for**

communicate with a Local Buffer Receiver from another serverless function shim on the same host synchronously. It is responsible for halting the shim until a response from the Local Buffer Receiver arrives.

CWASI inter-function communication model displayed Fig. 3, and its architecture shim, in Fig. 4, address RC-1. First, we present a model for inter-function communication. Further, we describe the CWASI architecture that enables this communication model.

# 5 CWASI THREE-MODE INTER-FUNCTION COMMUNICATION

CWASI introduces three key mechanisms for efficient and low-latency data exchange for serverless workflows in the Edge-Cloud continuum. These mechanisms enable each mode from the communication model presented in Section 4. CWASI facilitates function embedding by statically linking Wasm functions from the same namespace. In the second mode, CWASI leverages Unix Sockets to enable local buffer communication for functions in different namespaces. In the third mode, CWASI enables networked buffer communication via state-of-the-art mechanisms.

## 5.1 Function Embedding Communication Mode

For function embedding, CWASI enables Wasm functions static linking by analyzing the WAT file. Once the shim knows every necessary import for a specific function, it leverages the container manager snapshot to identify if any required import exists in the host [25]. This mechanism is executed before the `start` function part of the *Wasm Lifecycle* feature of the *Function Coordinator* component. Whenever the Wasm functions start, the shim is aware of every statically linked function enabling a seamless Wasm function integration with low latency communication.

To achieve that, before starting *fnA.wasm*, CWASI reads the *fnA.wasm* in WAT and searches for imports. If such import exists on the host, then *fn-utils.wasm* can be loaded into the same Wasm VM. CWASI statically links *fn-utils.wasm* with *fnA.wasm* Wasm VM during VM instantiation. Static links enable *fnA.wasm* to reuse *fn-utils.wasm* function without creating a second Wasm VM. Additionally, during execution time, *fnA.wasm* and *fn-utils.wasm* share the same Wasm VM memory address facilitating the data exchange between the Wasm binaries [25]. Once statically linked, *fnA.wasm* leverages FFI mechanisms to call method `get_image_metata da(image_file)` in *fn-utils.wasm*. In this specific situation, the communication remains within the same Wasm VM. *fnA.wasm* can access the memory from *fn-utils.wasm* to exchange data enabling the communication between the two functions as one function. Algorithm 3 shows how the *Function Embedding Discovery* detects additional Wasm functions import in the WAT file to enable Wasm static linking. Further, the *Function Embedding Discovery* looks for the bundles that match the WAT imports in the container manager snapshot, i.e., *containerd* snapshot, and returns them to the *Function Coordinator*. If the imported function exists in the snapshot path, the *Function Coordinator* loads these functions into the Wasm VM before starting it, as described in Algorithm 1.

In Algorithm 3, let the WAT file text be input $W$ and $B$, the result set containing all bundles' matches encountered by the Algorithm 3. For every file path $F_P$ where $F_P$ belongs to a single file object $O$. $O$ is part of a set of files $F$ in the container manager snapshot. $P$ is a set of import pattern matches in the WAT file $W$. If there exists an element $P$ such that file path $F_P$ equals $P$, then add $F_P$ to the result set $B$. The result set $B$ is returned to the *Function Coordinator*, which statically links the functions in the Wasm VM. Algorithm 1 receives input $VM$ API provided by the Wasm runtime and a set $M$ with the functions returned from Algorithm 3. Line 9 starts the Wasm VM with every function $m$ in set $M$ to the Wasm $VM$. Once every function is registered, they can seamlessly communicate as one application.

**Algorithm 3** Function Embedding Discovery

**Input:** $W$: WAT file text
**Output:** $B$: bundles path set
1: $B \leftarrow \emptyset$
2: $F \leftarrow$ files in container manager snapshot
3: **for all** $F_P(F_P \in O, \forall O \in F)$ **do**
4: **if** $\exists P(F_P = P \wedge P \in W)$ **then**
5: $B \cup F_P$
6: **end if**
7: **end for**

By enabling Function Embedding via Wasm static linking, CWASI addresses the research challenge RC-2, described in Section 2. It enables multiple Wasm functions to share the same address space, facilitating more efficient communication since the data can be transferred directly from the memory. Although loading multiple Wasm functions into a single Wasm VM directly affects the Wasm VM isolation, CWASI considers these functions to be fully trusted code that belongs to the same serverless function. Thus, these Wasm functions do not require to be executed in separate Wasm VMs.

## 5.2 Local Buffer Communication Mode

CWASI reduces the network traffic by forwarding the local traffic via a local buffer instead of using the network for local communications. By enabling local buffer communication, CWASI addresses the research challenge RC-3, described in Section 2.

Fig. 5 shows that *fnA* and *fnB* run on their own isolated Wasm VM, and each of them has its own CWASI shim instance. Separated Wasm VMs are necessary to keep isolation between the functions. To enable functions to exchange data via a local buffer, we use a Foreign Function Interface (FFI) mechanism, where the Wasm binary executes a function outside of its Wasm VM but on its own CWASI shim process. First ① *fnA* shim starts, it loads the Wasm function

binary file. The binary file path is present on the OCI specification created by the container manager. Then *fnA* shim executes *fnA* function; ② *FnA* function triggers Request Dispatcher on *FnA* shim which is executed with a pointer reference and data length from *fnA* input data; ③ The Request Dispatcher on *fnA* shim looks for running *fnB* in the container manager, and in positive cases connects to *fnB* socket and sends *fnA* function input; ④ *FnB* shim accepts *fnA* connection request, creates and executes Wasm *fnB* function; ⑤ *FnB* function is executed and its result returned to *fnB* shim; ⑥ *FnB* shim receives the function result and responds to *fnA* shim. Afterward, it starts its shutdown, including closing the server socket; ⑦ *FnA* shim receives the input from *fnB* shim and writes the result into the *fnA* function memory; ⑧ *FnA* read the memory with pointer and size returned from Request Dispatcher and resumes its execution; ⑨ Once *fnA* has finished, *fnA* shim returns the output to the container manager and starts its shutdown.

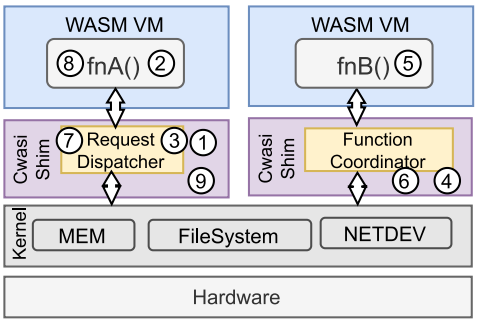



**Figure 5: Local Buffer Inter-function Communication**

CWASI provides the Request Dispatcher to enable data exchange between functions during runtime, as shown in Fig. 5. Request Dispatcher is a host function on the shim that enables host runtimes to communicate with Wasm functions via imports and exports. The Wasm function triggers the Request Dispatcher via an FFI that connects the Wasm function to the host function on the shim. As shown in Algorithm 4, the *Request Dispatcher* receives a memory API as input. The memory API gives access to Wasm VM, enabling *Request Dispatcher* to read and write directly into Wasm VM memory. Once the *Request Dispatcher* reads the input data from the source serverless function, it serializes the input and extracts the source and target function to create an inter-function communication pair. With the container manager running path from the OCI configuration file, the *Request Dispatcher* triggers the *ifc_selection*, shown in Algorithm 2, to look for the target function.

In Algorithm 4, let memory API $M$ and input values set $V_i$ to be the input. Let the output be a result string reference $R$, where $R$ is composed by pointer $R_P$ and string length $R_L$. Let function type $FT$ and function source payload $PL$ be part of the string read via memory API $M$. Algorithm 4 reads $FT$ and $PL$ via $M$ with the first two elements from input $V$, pointer $V_0$ and length $V_1$. $V_0$ and $V_1$ compose a reference to a string in the Wasm VM. $RP$ is the container manager running path statically accessible to the shim. $SP$ is the result from $ifc\_selection$ which receives target function type $FT$ and running path $RP$ as input. In cases where socket path $SP$ is not empty, Algorithm 4 connects to the target socket path $SP$ and sends function source payload $PL$. When socket path $SP$ is empty, the function publishes payload $PL$ into the target function type queue $FT$. Let the output of *connect_socket* and *publish_queue* be a string with a pointer reference and string length. Then, Algorithm 4 writes into the Wasm VM memory $M$ the result string $R$ using result pointer $R_P$ and result string length $R_L$. Finally, the function returns $R$ containing results pointer reference $R_P$ and length $R_L$.

**Algorithm 4** Request Dispatcher

**Input:** $M, V_i$ where $i \in \mathbb{N}_0$: VM memory API and input set
**Output:** $R = (R_P, R_L)$ : Result pointer and length

```
FT, PL ∈ M.read(V_0,V_1)
RP ← current container manager running path
SP ← ifc_selection(FT, RP)
if SP ≠ ∅ then
    R ← connect_socket(SP, PL)
else
    R ← publish_queue(FT, PL)
end if
M.write(R_P, R_L)
return R
```

## 5.3 Networked Buffer Communication Mode

On the third mode, CWASI enables inter-function communication by leveraging existing state-of-the-art publish/subscribe mechanisms instead of relying on typical storage solutions. Direct inter-function communication leads to significant latency and throughput improvement when compared to solutions that rely on remote services such as remote storage, KVS, and databases [17].

# 6 CWASI IMPLEMENTATION

CWASI is published as an open-source shim part of the Polaris SLO Cloud. Polaris itself is part of the Linux Foundation Centaurus project. It is implemented in Rust and currently supports WasmEdge runtime. source code is available on GitHub. It is implemented in Rust and currently supports WasmEdge runtime. CWASI source code is available on GitHub[1]. We are influenced by RunWasi, which introduces runtime shims for the state-of-the-art Wasm runtimes WasmEdge and Wasmtime. For this implementation, we connect the container manager *containerd* using the protobuf provided by RunWasi crates. CWASI adopts lifecycle-related methods from RunWasi, meaning they connect directly to *containerd* via protobuf. In the Wasm context, the shim is a *host runtime* that interacts with Wasm binaries via imports and exports. Currently, CWASI supports WasmEdge and leverages WasmEdge SDK [49] to interact with Wasm binaries' memory space to exchange data between different modules. We have chosen WasmEdge runtime for the variety of its available features and community support.

*Inter-function communication.* For local buffer communication mode, CWASI leverages IPC – Unix Domain Socket (UDS) for inter-function data exchange on the same host to enable bi-directional data exchange between the functions. Unlike other IPC communication mechanisms such as Unix pipe, UDS allows the communication

[1]https://github.com/polaris-slo-cloud/containerd-shim-cwasi

between two non-related processes, i.e., processes without direct communication or co-dependencies to exchange data[50]. For co-located inter-function communication between two shims, we leverage UDS, creating a temporary file with OCI bundle name and `.sock` suffix in the container manager snapshot, e.g., `/var/run/containerd/io.containerd.runtime.v2.task/mycontainer.sock`. The shim shuts down every socket server and removes the temporary socket file when the process finishes. Alternatively, in case of errors, the container manager sends a kill signal to shim, which cleans up the container process resources, shutdown the socket, and removes the socket file. We leverage the rust `UnixListener` to create a socket server type `AF_UNIX` and `UnixStream` to create a socket client from rust module `std::os::unix::net`[2]. For remote inter-function communication, we leverage Redis Pub/Sub[3].

*Complex data transfer.* Currently WebAssembly only supports numbers as data types, i.e., integers and float of 32-bit and 64-bit each. Therefore transferring complex data such as strings is a challenging task [51]. As CWASI is a host runtime with access to the Wasm VM memory space, we leverage WasmEdge APIs to transfer complex data to the shim such as string by sending a pointer reference along with the data length. Further, the shim reads the byte array from memory and converts it to a string. The shim does the reverse action to enable Wasm modules to receive strings during runtime. It writes the byte array into the Wasm VM memory and returns the pointer and length to the Wasm module. Finally, the Wasm modules can retrieve the byte array to a string. Libraries such as wasm-bindgen and wasmedge-bindgen attempt to facilitate a high-level complex data transfer interaction between Wasm modules. Nevertheless, wasm-bindgen cannot be applied for data transfer between the Wasm module and host function until the current moment of writing this paper.

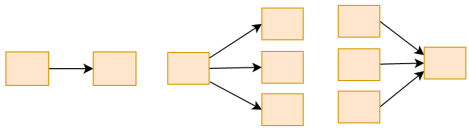
# 7 EXPERIMENTS

## 7.1 Overview

We design experiments to evaluate our scenario application using Serverless workflow's most important invocation patterns, including *Sequential*, *Fan-out*, and *Fan-in* (Fig. 6), as outlined in [52]. Furthermore, the experiments aim to measure the performance of the main contributions of this paper, listed in Section 1.

Fig. 6a shows an example of a *Sequential* workflow when only one instance of *Extract Frames*, *Process Frames*, and *Prepare Dataset* is created, so each function is called sequentially. A *Fan-out* workflow, shown in Fig. 6b, happens when one function triggers multiple parallel functions. In our scenario, that occurs when the resulting frame from *Extrac Frames* stage triggers multiple *Process Frames* e.g., one function for labeling and another function for anonymization. Finally, we have a *Fan-in*, shown in Fig. 6c, when multiple parallel functions trigger one single function. In our scenario, that happens when multiple instances of *Process Frames* functions trigger one function *Prepare Dataset*. We describe the results of evaluating these workflow composition patterns in Section 7.3, Section 7.4, and Section 7.5, respectively.

We compare CWASI to the baselines Runwasi WasmEdge shim [31] and a container solution OpenFaas [37]. CWASI Shim focuses on



**(a) Sequential** **(b) Fan-out** **(c) Fan-in**

**Figure 6: Experimental Workflows**

inter-function communication. Therefore, we measure the specific time frame between two serverless functions, when the shim sends the message until the subsequent shim receives the message. We use an HTTP client and server for WasmEdge for inter-function communication. For OpenFaas, we use its standard API Gateway, which is also an HTTP server client solution.

One of the main characteristics of Serverless Computing is the function composition between small stateless short-lived functions. As stateless functions, inter-function data exchange is crucial for serverless computing [53]. We must ensure that functions can communicate on a large scale with low latency without adding extra resource overhead on the multi-tenant environment from the Edge-Cloud continuum. Hence, we perform experiments to show performance, scalability, and potential drawbacks in resource usage. For this purpose, we collect the following metrics:

*Latency.* This metric shows the exact execution time when *fnA* shim sends the message until *fnB* shim receives the message. We use seconds as the unit metric for the latency experiments.

*Throughput.* With this metric, we want to examine if the optimization performed does not affect the scalability of serverless functions under high load. We use requests per second to indicate the throughput. In cases where the executions are lower than one second, we extrapolate, e.g., if we send ten requests that take less than one second, we extrapolate the throughput by considering the rate of execution over a one-second timeframe.

*Resource Usage.* As our co-located inter-function leverages the host resources, we measure the resource usage of the host machine to show that CWASI shim performs similarly or at maximum small increases compared to the baseline metrics. We display the CPU usage in % and the RAM usage in Kb.

## 7.2 Experiment Setup

To evaluate CWASI shim, we execute the designed workflows with CWASI and, in the baselines WasmEdge and OpenFaas. Our Serverless functions for co-located experiments are performed in a single machine Intel NUC with i5 2.9 GHz 8GB RAM with Ubuntu 22.04 LTS. This machine executes all serverless functions. We perform OpenFaas experiments with Openfaas faasd [37]. As container manager, we use containerd and its client cli tool [4] to execute the serverless functions. Additionally, we use shell scripts to start multiple functions for every baseline. To avoid biased results, we repeated the experiments ten times and collected the average result.

[2]https://doc.rust-lang.org/std/os/unix/net/index.html
[3]https://redis.com/solutions/use-cases/messaging

[4]https://github.com/projectatomic/containerd/blob/master/docs/cli.md

## 7.3 Sequential Workflow

In this experiment, we perform synchronous requests between two serverless functions fnA and fnB as shown in Fig. 6a. We increase the input size during the experiment to evaluate the solution with different loads. Fig. 7 shows overall latency, throughput, and resource results while Table 1 shows precise measurements extracted from the overall results.

Fig. 7a displays the input data size in MB on the $x$-axis and the latency on the $y$-axis. Over input size increase, CWASI shows from 5 to 143 ms, WasmEdge shows from 48 ms to 4.4 s, while OpenFaas shows from 24 ms to 368 ms. The latency results show CWASI decreases up to 79% and up to 89% of the latency when compared to OpenFaas and WasmEdge, respectively. The expected linear increase shows application stability in all three cases with consistent performance.

Fig. 7b shows the throughput of the systems. $x$ axis shows the input data size and the $y$ axis shows how many requests the systems can handle per second. The results show a linear throughput decrease over the input size increase, which means the complexity is at maximum linear. According to Table 1, CWASI shows a throughput from 266 to 6.9, WasmEdge from 15 to 0.17, and OpenFaas from 41 to 2.7 requests per second. The results show CWASI increases the throughput up to 5x when compared to OpenFaas and up to 14x compared to WasmEdge.

Fig. 7c shows RAM and CPU usage for CWASI, OpenFaas, and WasmEdge. We selected three input sizes representing the small, mid, and large input sizes with latency and throughput experiments. On the left side of Fig. 7c, we notice a higher CPU usage from CWASI and WasmEdge. As OpenFaas only includes the scale to zero feature in the pro bundle and we perform the experiments with the community version, our experiments with OpenFaas have two functions running constantly. CWASI and WasmEdge had different functions and processes for each request in this experiment. The slight difference in the experiment design does not affect the latency and throughput but leads to lower CPU usage as the functions are reused in OpenFaas. The right side of Fig. 7c shows RAM usage. We observe a decrease of up to 30% usage in CWASI compared to WasmEdge and up to 15% compared to OpenFaas.

**Table 1: Sequential Result**

| Sequential Workflow | | | | |
|---|---|---|---|---|
| | Latency sec | | Throughput Req/sec | |
| | 2MB | 100M | 2MB | 100MB |
| CWASI | 0.0051 | 0.1436 | 266.4535 | 6.9569 |
| WasmEdge | 0.048 | 4.4324 | 15.8323 | 0.1732 |
| OpenFaas | 0.0248 | 0.3681 | 41.7240 | 2.7556 |

## 7.4 Fan-out Workflow

In this experiment, we perform fan-out requests. As parallel requests are a regular part of real-world use cases, as described in Section 2, the goal of this experiment is to analyze the performance of CWASI shim compared to the baselines WasmEdge and OpenFaas when executing parallel requests. This experiment comprises one function that branches out and executes multiple parallel requests. Since WASI does not support multi-threading [54], we leverage wasmedge [49] libraries such as hyper-wasi and tokio-wasi to execute Wasm asynchronous requests. Fig. 6b exemplifies the design of the fan-out experiments. We use a fixed input size of 2MB and increase the level of multiple parallel requests in the fan-out starting from 10 to 500 fan-out requests.

Fig. 8 shows latency, throughput, and resource usage results. In the latency results Fig. 8a, we have the multiple parallel executions in axis $y$ and the duration in seconds in axis $x$. We observe stable horizontal lines with slight degrees of change for every three frameworks. Table 2 shows a sample of metrics collected during this experiment. CWASI shows around 6 ms, WasmEdge shows 195 ms, and OpenFaas 16 ms. Although none of the frameworks display significant changes over the axis $x$, CWASI shows up to 62% lower latency execution when compared to OpenFaas and up 95% compared to WasmEdge.

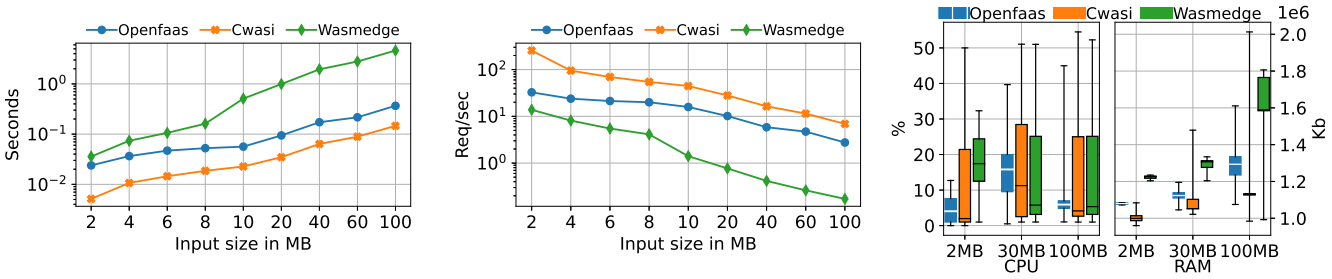
Similar to Fig. 8a latency results, we see horizontal lines representing the measured frameworks in the throughput results shown in Fig. 8b. On axis $x$, we have the fan-out degree, while on axis $y$ requests per second. The expected stable throughput for every framework indicates no performance issues under high load. According to Table 2, CWASI shows around 3 ms, WasmEdge 115 ms, and OpenFaas 7 ms. Hence, CWASI shows up to 1.3x higher throughput compared to OpenFaas and up to 37x compared to WasmEdge.

Fig. 8c shows CPU and RAM usage for the fan-out experiment. On the left of Fig. 8c, we have execution metrics samples on the axis $x$ and CPU percentage usage on the axis $y$. Overall, we observe a slight CPU usage increase between 10 and 500 executions where CWASI displays peaks of 100% usage. On the right side of Fig. 8c, we observe an increase in RAM usage from CWASI compared to OpenFaas and WasmEdge. The low resource usage is because WasmEdge and OpenFaas rely on the HTTP client and server where we could reuse the same HTTP server. At the same time, CWASI explicitly creates a new process for every new function, leading to higher CPU usage for CWASI. The processing creating increase reflects on the CPU usage peaks and RAM usage. The resource usage increase reflects latency and throughput results shown in Fig. 8 and in Fig. 8b, respectively.

## 7.5 Fan-in Workflow

In these experiments, we aim to measure CWASI scalability with multiple requests in fan-in workflows as shown in Section 7.5. As CWASI assumes every request is a new function and consequently a new process, we have limitations to creating a fan-in workflow by simply calling the same function over and over. In this scenario, if a particular function is called ten times, CWASI creates ten different functions, which means fan-in is the same and ten different sequential workflows. To overcome this experiment limitation, we use the fan-out experiments and calculate the responses, revealing a fan-in workflow. Using the same set of experiments means the resource results shown in Fig. 8c corresponds to fan-out and fan-in experiments. Thus, in this subsection, we address latency and throughput.

Fig. 9a shows the latency results for the fan-in experiments. On axis $x$, we have the number of executions during fan-in, as presented in Fig. 6c, while on axis $y$, we have the latency in seconds. As in previous experiments, every three measures framework displays horizontal lines, which means the performance does not decrease with the given executions. The slight variation in CWASI results



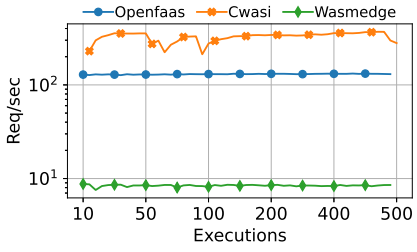

**(a) Latency** **(b) Throughput** **(c) Resource**

**Figure 7: Sequential Results**


**(a) Latency** **(b) Throughput** **(c) Resource**

**Figure 8: Fan-out Results**


**(a) Latency**




**(b) Throughput**

**Figure 9: Fan-in Results**

**Table 2: Fan-out and Fan-in Sample Results**

| **Fan-out Workflow** | | | | |
|---|---|---|---|---|
| | **Latency sec** | | **Throughput Req/sec** | |
| | 10 Exec | 100 Exec | 10 Exec | 100 Exec |
| CWASI | 0.0045 | 0.0066 | 203.7327 | 211.2039 |
| WasmEdge | 0.1931 | 0.1950 | 4.3781 | 4.9118 |
| OpenFaas | 0.0441 | 0.0169 | 59.8344 | 61.2589 |
| **Fan-in Workflow** | | | | |
| CWASI | 0.0029 | 0.0032 | 298.9536 | 314.2019 |
| WasmEdge | 0.1161 | 0.1170 | 8.6752 | 8.5336 |
| OpenFaas | 0.0075 | 0.0079 | 127.3925 | 130.8297 |

in orange is due to resource usage already discussed in Section 7.4. According to Fig. 9a and Table 2, CWASI shows around 3 ms while OpenFaas performs around 8 ms and WasmEdge shortly over 110 ms. Overall, CWASI shows up to 60% latency decrease compared to OpenFaas and up to 95% when compared to WasmEdge.

In Fig. 9b, we see the throughput performance in the fan-in experiments. As expected from the fan-in latency results, the throughput shows three horizontal lines with slight variation in CWASI, in orange, due to the resource usage, where WasmEdge and OpenFaas remain constant throughout the experiment. On axis $x$, we have the executions; on axis $y$, we have the throughput in requests per second. CWASI displays around 300 requests per second while OpenFaas shows 130 and WasmEdge 8 requests per second. CWASI has 1.3x higher throughput than OpenFaas and over 30x higher throughput when compared to WasmEdge.

# 8 CONCLUSION & FUTURE WORK

We presented in this paper an interoperable inter-function communication model for the Serverless Edge-Cloud Continuum with three modes: Function Embedding, Local Buffer, and Networked Buffer. Additionally, we introduce CWASI, a WebAssembly runtime shim that implements the three-mode communication model to enable optimized inter-function communication. CWASI Shim leverages the function locality to identify and select the best inter-function communication mode.

We evaluated CWASI by running Serverless workflows among the most relevant serverless invocation patterns sequence, fan-out, and fan-in using latency, throughput, and resource usage as the metric baselines. The experiments show that CWASI provides up to 95% lower latency and up to 30x higher throughput compared to the start-of-the-art container runtimes for serverless platforms. CWASI shows significant improvement in serverless workflows that have high co-located inter-function communication.

Currently, CWASI only supports WasmEdge runtime. In the future, we plan to address this by extending CWASI for other runtimes to enable portability and flexibility. Additionally, CWASI relies on particular OCI specifications, i.e., annotations. To overcome this limitation, we plan to introduce programming models to identify functions of the same workflow and translate this composition into OCI specifications, such as arguments and annotations. We envision programming models identifying whether a serverless function is better suited for standard containers or WebAssembly and creating a specific deployment hint for each serverless function. Furthermore, we envision CWASI as an enabler for a *BaaSless* Serverless Computing where the framework will manage the backend-as-a-service infrastructure, further simplifying application development. Our future programming models will enable CWASI to identify necessary backend-as-a-service for a specific workflow. Once the BaaS' is identified, CWASI will proactively start and manage every BaaS necessary to run this particular workflow.

# ACKNOWLEDGMENTS

This research received funding from the EU's Horizon Europe Research and Innovation Program under Grant Agreement No. 101070186. EU website for TEADAL: https://teadal.eu.

# REFERENCES

[1] Luciano Baresi and Danilo Filgueira Mendonça. 2019. Towards a serverless platform for edge computing. In *2019 IEEE International Conference on Fog Computing (ICFC)*, 1–10. DOI: 10.1109/ICFC.2019.00008.

[2] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless computing: one step forward, two steps back. (2018). arXiv: 1812.03651 [cs.DC].

[3] Phani Kishore Gadepalli, Gregor Peach, Ludmila Cherkasova, Rob Aitken, and Gabriel Parmer. 2019. Challenges and opportunities for efficient serverless computing at the edge. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, 261–2615. DOI: 10.1109/SRDS47363.2019.00036.

[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), 185–200. DOI: 10.1145/3140587.3062363.

[5] Philipp Gackstatter, Pantelis Frangoudis, and Schahram Dustdar. 2022. Pushing serverless to the edge with webassembly runtimes. In (May 2022), 140–149. DOI: 10.1109/CCGrid54584.2022.00023.

[6] Stefan Nastic, Philipp Raith, Alireza Furutanpey, Thomas Pusztai, and Schahram Dustdar. 2022. A serverless computing fabric for edge & cloud. In *2022 IEEE 4th International Conference on Cognitive Machine Intelligence (CogMI)*, 1–12. DOI: 10.1109/CogMI56440.2022.00011.

[7] Sasko Ristov, Stefan Pedratscher, and Thomas Fahringer. 2023. *xAFCL*Xafcl: run scalable function choreographies across multiple faas systems. *IEEE Transactions on Services Computing*, 16, 1, 711–723. DOI: 10.1109/TSC.2021.3128137.

[8] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. 2018. Anna: a kvs for any scale. *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 401–412.

[9] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M. Faleiro, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: stateful functions-as-a-service. *Proc. VLDB Endow.*, 13, 2438–2452.

[10] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. 2022. Jiffy: elastic far-memory for stateful serverless analytics. In *Proceedings of the Seventeenth European Conference on Computer Systems* (EuroSys '22). Association for Computing Machinery, Rennes, France, 697–713. ISBN: 9781450391627. DOI: 10.1145/3492321.3527539.

[11] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup. 2021. Serverless applications: why, when, and how? *IEEE Software*, 38, 01, (Jan. 2021), 32–39. DOI: 10.1109/MS.2020.3023302.

[12] Thanasis G. Papaioannou, Nicolas Bonvin, and Karl Aberer. 2012. Scalia: an adaptive scheme for efficient multi-cloud storage. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 1–10. DOI: 10.1109/SC.2012.101.

[13] Pengwei Wang, Caihui Zhao, Yi Wei, Dong Wang, and Zhaohui Zhang. 2020. An adaptive data placement architecture in multicloud environments. *Sci. Program.*, 2020, 1704258:1–1704258:12.

[14] Bytecode Alliance. 2023. Wasi. https://wasi.dev.

[15] Anna Maria Nestorov, Josep Lluís Berral, Claudia Misale, Chen Wang, David Carrera, and Alaa Youssef. 2022. Floki: a proactive data forwarding system for direct inter-function communication for serverless workflows. In *Proceedings of the Eighth International Workshop on Container Technologies and Container Clouds* (WoC '22). Association for Computing Machinery, Quebec, Quebec City, Canada, 13–18. ISBN: 9781450399296. DOI: 10.1145/3565384.3565890.

[16] Philipp Raith, Stefan Nastic, and Schahram Dustdar. 2023. Serverless edge computing—where we are and what lies ahead. *IEEE Internet Computing*, 27, 3, 50–64. DOI: 10.1109/MIC.2023.3260939.

[17] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: towards High-Performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, (July 2018), 923–935. ISBN: 978-1-939133-01-4. https://www.usenix.org/conference/atc18/presentation/akkus.

[18] Marcin Copik, Alexandru Calotoiu, Rodrigo Bruno, Roman Böhringer, and Torsten Hoefler. 2022. Process-as-a-Service: FaaSt Stateful Computing with Optimized Data Planes. Tech. rep. (Jan. 2022).

[19] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. 2019. FreeFlow: software-based virtual RDMA networking for containerized clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, (Feb. 2019), 113–126. ISBN: 978-1-931971-49-2. https://www.usenix.org/conference/nsdi19/presentation/kim.

[20] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '21). Association for Computing Machinery, Virtual, USA, 152–166. ISBN: 9781450383172. DOI: 10.1145/3445814.3446701.

[21] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: accelerating Function-as-a-Service workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, (July 2021), 805–820. ISBN: 978-1-939133-23-6. https://www.usenix.org/conference/atc21/presentation/kotni.

[22] Simon Shillaker and Peter Pietzuch. 2020. Faasm: lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, (July 2020), 419–433. ISBN: 978-1-939133-14-4. https://www.usenix.org/conference/atc20/presentation/shillaker.

[23] Simon Shillaker, Carlos Segarra, Eleftheria Mappoura, Mayeul Fournial, Lluis Vilanova, and Peter Pietzuch. 2023. Faabric: fine-grained distribution of scientific workloads in the cloud. *arXiv preprint arXiv:2302.11358*.

[24] Qiang Su, Chuanwen Wang, Zhixiong Niu, Ran Shu, Peng Cheng, Yongqiang Xiong, Dongsu Han, Chun Jason Xue, and Hong Xu. 2022. Pipedevice: a hardware-software co-design approach to intra-host container communication. In *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies* (CoNEXT '22). Association for Computing Machinery, Roma, Italy, 126–139. ISBN: 9781450395083. DOI: 10.1145/3555050.3569118.

[25] Linux Foundation. 2021. How and why to link webassembly modules. https://training.linuxfoundation.org/blog/how-and-why-to-link-webassembly-modules/.

[26] AWS Amazon. 2021. Field notes: building an automated image processing and model training pipeline for autonomous driving. https://aws.amazon.com/blogs/architecture/field-notes-building-an-automated-image-processing-and-model-training-pipeline-for-autonomous-driving/.

[27] Anurag Ghosh, Srinivasan Iyengar, Stephen Lee, Anuj Rathore, and Venkata N Padmanabhan. 2023. React: streaming video analytics on the edge with asynchronous cloud support. In *Proceedings of the 8th ACM/IEEE Conference on Internet of Things Design and Implementation* (IoTDI '23). Association for Computing Machinery, San Antonio, TX, USA, 222–235. ISBN: 9798400700378. DOI: 10.1145/3576842.3582385.

[28] Miao Zhang, Fangxin Wang, Yifei Zhu, Jiangchuan Liu, and Bo Li. 2021. Serverless empowered video analytics for ubiquitous networked cameras. *IEEE Network*, 35, 6, 186–193. DOI: 10.1109/MNET.101.2000668.

[29] Yiding Wang, Weiyan Wang, Junxue Zhang, Junchen Jiang, and Kai Chen. 2019. Bridging the Edge-Cloud barrier for real-time advanced vision analytics. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA, (July 2019). https://www.usenix.org/conference/hotcloud19/presentation/wang.

[30] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: a serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference* (Middleware '20). Association for Computing Machinery, Delft, Netherlands, 265–279. ISBN: 9781450381536. DOI: 10.1145/3423211.3425680.

[31] Containerd. 2023. Runwasi. https://github.com/containerd/runwasi.

[32] Garrett McGrath and Paul R. Brenner. 2017. Serverless computing: design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 405–410. DOI: 10.1109/ICDCSW.2017.36.

[33] Karim Djemame, Matthew Parker, and Daniel Datsev. 2020. Open-source serverless architectures: an evaluation of apache openwhisk. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, 329–335. DOI: 10.1109/UCC48980.2020.00052.

[34] Rakesh Kumar and B Thangaraju. 2020. Performance analysis between runc and kata container runtime. In *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*. IEEE, 1–4.

[35] Junfeng Li, Sameer G. Kulkarni, K. K. Ramakrishnan, and Dan Li. 2019. Understanding open source serverless platforms: design considerations and performance. In *Proceedings of the 5th International Workshop on Serverless Computing* (WOSC '19). Association for Computing Machinery, Davis, CA, USA, 37–42. ISBN: 9781450370387. DOI: 10.1145/3366623.3368139.

[36] Alessandro Randazzo and Ilenia Tinnirello. 2019. Kata containers: an emerging architecture for enabling mec services in fast and secure way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE, 209–214.

[37] OpenFaaS. 2023. Serverless functions, made simple. https://www.openfaas.com.

[38] Benedikt Spies and Markus Mock. 2021. An evaluation of webassembly in non-web environments. In *2021 XLVII Latin American Computing Conference (CLEI)*, 1–10. DOI: 10.1109/CLEI53233.2021.9640153.

[39] Bytecode Alliance. 2023. Wasmtime. https://wasmtime.dev.

[40] Salim S. Salim, Andy Nisbet, and Mikel Luján. 2020. Trufflewasm: a webassembly interpreter on graalvm. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (VEE '20). Association for Computing Machinery, Lausanne, Switzerland, 88–100. ISBN: 9781450375542. DOI: 10.1145/3381052.3381325.

[41] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. 2019. Weakening webassembly. *Proc. ACM Program. Lang.*, 3, OOPSLA, Article 133, (Oct. 2019), 28 pages. DOI: 10.1145/3360559.

[42] Vojdan Kjorveziroski, Sonja Filiposka, and Anastas Mishev. 2022. Evaluating webassembly for orchestrated deployment of serverless functions. In *2022 30th Telecommunications Forum (TELFOR)*, 1–4. DOI: 10.1109/TELFOR56187.2022.9983733.

[43] Ju Long, Hung-Ying Tai, Shen-Ta Hsieh, and Michael Juntao Yuan. 2021. A lightweight design for serverless function as a service. *IEEE Software*, 38, 1, 75–80. DOI: 10.1109/MS.2020.3028991.

[44] The Linux Foundation. 2023. Open container initiative runtime specification. https://github.com/opencontainers/runtime-spec/blob/main/spec.md.

[45] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, (July 2018), 57–70. ISBN: 978-1-931971-44-7. https://www.usenix.org/conference/atc18/presentation/oakes.

[46] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, (July 2021), 285–301. ISBN: 978-1-939133-23-6. https://www.usenix.org/conference/atc21/presentation/mahgoub.

[47] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, (Mar. 2017), 363–376. ISBN: 978-1-931971-37-9. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[48] Misun Park, Ketan Bhardwaj, and Ada Gavrilovska. 2023. Pocket: ml serving from the edge. In *Proceedings of the Eighteenth European Conference on Computer Systems* (EuroSys '23). Association for Computing Machinery, Rome, Italy, 46–62. ISBN: 9781450394871. DOI: 10.1145/3552326.3587459.

[49] Wasmedge. 2023. Wasmedge. https://wasmedge.org.

[50] Michael Kerrisk. 2010. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. (1st ed.). No Starch Press, USA. ISBN: 1593272200.

[51] W3C Community Group. 2023. Webassembly specification. https://webassembly.github.io/spec/core/index.html.

[52] Eric Jonas et al. 2019. Cloud programming simplified: a berkeley view on serverless computing. (2019). arXiv: 1902.03383 [cs.OS].

[53] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (SoCC '20). Association for Computing Machinery, Virtual Event, USA, 30–44. ISBN: 9781450381376. DOI: 10.1145/3419111.3421280.

[54] WebAssembly. 2020. Multi-threading and atomics. https://github.com/WebAssembly/WASI/issues/296.