Informatics

SIWA: Wasm Serverless Actors for the Edge-Cloud Continuum

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Jack Shahhoud, BSc.
Matrikelnummer 01631081

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Asst. Prof. Dr. Stefan Nastic
Mitwirkung: Dipl.-Ing. Cynthia Marcelino, BSc.

Wien, 27. August 2024

Jack Shahhoud

Stefan Nastic

Informatics

SIWA: Wasm Serverless Actors for the Edge-Cloud Continuum

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Jack Shahhoud, BSc.
Registration Number 01631081

to the Faculty of Informatics

at the TU Wien

Advisor: Asst. Prof. Dr. Stefan Nastic
Assistance: Dipl.-Ing. Cynthia Marcelino, BSc.

Vienna, August 27, 2024

Jack Shahhoud	Stefan Nastic

Erklärung zur Verfassung der Arbeit

Jack Shahhoud, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 27. August 2024

Jack Shahhoud

Acknowledgements

I want to thank God for helping me in this long and challenging journey.

I want to thank Professor Dustdar and the advisors for the opportunity to work on this thesis and for the guidance and support.

Kurzfassung

Serverless Computing ist ein Computerparadigma, das effizientes Infrastrukturmanagement und elastische Skalierbarkeit bietet. Serverless Functions werden je nach Bedarf hoch- oder herunterskaliert, was bedeutet, dass Functions nicht direkt adressierbar sind und auf plattformgesteuerte Aufrufe angewiesen sind. Darüber hinaus erfordert seine Serverless Natur, dass Funktionen externe Dienste wie Objektspeicher und Key Value Store (KVS) nutzen, um Daten auszutauschen. Serverless Actors sind entstanden, um diese Probleme zu lösen. Dennoch verlassen sich die aktuellen Serverless Actors auf den hochmodernen Serverless LifeCycles und event-trigger Aufrufe und zwingen die Actors daher, Remotedienste zu nutzen, um ihren Status zu verwalten und Daten auszutauschen.

Um diese Probleme zu lösen, stellen wir in diesem Artikel ein neuartiges Serverless LifeCycle Model vor, das die Wiederverwendung von Actors ermöglicht, sodass Actors ihren Status zwischen Ausführungen beibehalten und zugewiesene Ressourcen in anderen Zuständen wiederverwenden können. Darüber hinaus schlagen wir ein neuartiges Serverless Aufrufmodell vor, das es Serverless Actors ermöglicht, das Verhalten zukünftiger Nachrichten zu beeinflussen. Eine dedizierte verteilte Middleware wird verwendet, um die Kommunikation Zwischen Actors zu ermöglichen und neue Anfragen zu verarbeiten.

Abschließend präsentieren wir SIWA, ein leicht WebAssembly Serverless Actor Platform, das es Serverless Functions ermöglicht, sich wie Actors zu verhalten. SIWA-Actors haben eine eindeutige Adresse, die eine direkte Kommunikation über die SIWA Distributed Middleware ermöglicht. SIWA nutzt WebAssembly, um Actors im Vergleich zu anderen Virtualisierungsmethoden eine leichtgewichtige Sandbox-Isolierung zu bieten, wodurch sie für das Edge-Cloud-Kontinuum geeignet sind, wo Rechenleistung und Ressourcen begrenzt sind. Viele Sprachen unterstützen WebAssembly als Kompilierungsziel und bieten dem Entwickler so mehr Optionen zur Implementierung gewünschter Funktionen. Versuchsergebnisse zeigen, dass SIWA die Datenaustauschlatenz um bis zu 92% optimiert und den Durchsatz im Vergleich zu OpenFaaS und Spin um das bis zu 10× erhöht.

Abstract

Serverless Computing is a computing paradigm that provides efficient infrastructure management and elastic scalability. Serverless functions scale up or down based on demand, which means that functions are not directly addressable and rely on platform-managed invocation. Moreover, its stateless nature requires functions to leverage external services, such as object storage and Key Value Store (KVS), to exchange data. Serverless actors have emerged to address these issues. Nevertheless, the current Serverless actors rely on the state-of-the-art Serverless Lifecycle and event-trigger invocation, thus forcing actors to leverage remote services to manage their state and exchange data. Functions run in a dedicated container or virual machine to provide isolation. This increases the consumption of resources and raises the configuration complexity.

To address these issues, in this paper, we introduce a novel Serverless LifeCycle Model that allows actors to be reused, enabling actors to maintain their state between executions and reuse allocated resources in other states. Additionally, we propose a novel Serverless Invocation Model that enables serverless actors to influence the behaviour of future messages using the state of the actor. A dedicated distributed middleware is used to enable actor communication and for processing new requests.

Finally, we present SIWA, a lightweight WebAssembly Serverless Actor platform that enables Serverless functions to behave as actors. SIWA actor has a unique address that enables direct communication via SIWA Distributed Middleware. SIWA leverages WebAssembly to provide the actors with lightweight sandbox isolation compared to other virtualization methods, making them suitable for the Edge-Cloud Continuum, where computational power and resources are limited. Functions are executed in the actor using WebAssembly Virtual Machine. Many languages support WebAssembly as a compilation target, providing the developer with more options to implement desired functions. Experimental results show that SIWA optimises the data exchange latency by up to 92% and increases the throughput by up to 10x compared to OpenFaaS and Spin.

Contents

CHAPTER 1

Introduction

Serverless computing is a paradigm that offers infrastructure management and elastic scaling. In serverless computing, stateless functions respond to an event trigger. Due to the serverless stateless design, functions leverage external services such as object storage, message brokers, and Key-Value Stores (KVS) to exchange data, pushing data race management to the applications [JSS$^+$19, RND23, NRF$^+$22]. Additionally, functions are not directly accessible; they are accessible via platform ingresses such as API Gateway and Load Balancer [BFM19, WCJL23, HFG$^+$18].

Serverless Actors [Akk, CDTV24, BGJ$^+$21, BPSAP$^+$19, MNW$^+$18, HKO21, SP20] have emerged addressing these issues, thus enabling direct communication, state persistence, and concurrency management, which is crucial for serverless functions. Nevertheless, existing serverless actor approaches still rely on state-of-the-art serverless design to enable actors.

1.1 Problem Statement

1.1.1 Actors

Actors [Agh86, Hal12] are isolated entities that can [CDTV24, BGJ$^+$21, Hal12, POPL18, SCH24]:

1. Create other actors
2. Directly communicate with other actors
3. Influence the behavior or state for the next received message

On the other hand, some properties of Serverless functions are [NRF$^+$22, BFM19, WCJL23, HFG$^+$18]:

- Stateless
- Non-addressable
- Event-triggered

Current existing serverless actor approaches [Akk, BGJ+21, MNW+18, SP20, HKO21] leverage the state-of-the-art serverless design characteristics such as lifecycle [Mun19] and event-trigger invocation [JSS+19, WCJL23, KHA+23]. The serverless lifecycle facilitates the creation and management of actors, while event-triggered messages enable distributed and decoupled communication between serverless actors.

1.1.2 Serverless Function LifeCylce

In the current serverless function lifecycle [Mun19], serverless platforms typically launch one stateless function instance per request that either succeeds or fails while its state is typically stored in remote services. Existing approaches that enable functions to preserve state include:

1. *Programming Models* [BGJ+21, CDTV24, BGK+11] that abstract the function state handling from the developer and leverage external services to store it. Such programming models provide frameworks and libraries that automatically manage state persistence. While programming models simplify state management, they might introduce latency overhead due to dependencies on external storage systems.

2. *Sidecars* [Akk, JW21a] systems that act as proxies and manage state interactions transparently, thus ensuring that state consistency and storage are handled outside the serverless function lifecycle, thereby reducing the function's overhead. Despite their benefits, sidecars run alongside the function, consuming additional CPU and memory resources, which impacts the overall resource usage and might become a challenge at Edge-Cloud Continuum.

3. *Custom Sandboxes* [SP20, CCB+22] ensure that functions can access and modify shared states in a controlled manner, providing isolation and, at the same time, enabling efficient state management. Although custom sandboxes might be lightweight, they are not interoperable with the current state-of-the-art platforms, limiting their usage on different serverless platforms such as AWS Lambda, Azure, and GCF. PaaS [CCB+22] still relies on containers to run functions, which can influence the latency of execution upon start.

4. Some *Serverless Platforms* [HKO21, SWL+20] integrate native support for stateful computing, allowing functions to maintain state across multiple invocations via dedicated or state-of-the-art storage mechanisms. They still rely on external storage systems.

However, these approaches depend on the lifecycle of serverless functions that succeed or fail. To fully utilize actor potential, the lifecycle must ensure actor reuse while preserving its state. Consequently, actors maintain states between executions, avoiding unnecessary state propagation. They also rely on external storage to import/export the state, affecting the latency.

1.1.3 Message Exchange

In the existing event-triggered invocation design, serverless platforms typically execute functions in response to events. This asynchronous message processing enables high concurrency and scalability, as each function can operate independently. However, as functions are triggered by events and therefore not addressable, they rely on remote services to exchange data. To enable direct communication between functions, current approaches leverage

1. *Remote services* such as object storage [S3], KVS [WFLH18, KWS+18, BPSAP+19] and cache [WZM+20, RCG+21, HFC+23, MTA+23], are commonly used to enable direct communication between serverless functions. However, remote services might increase latency by up to 95% compared to direct communication [KNGB21].

2. *Direct Communication* such as message queues [ACR+18], TCP-punch hole [CBCH23], shared-memory [SSM+23, MN23, JW21b], disk storage [MSM+21], and local cache [MBN+21, ACR+18, SWL+20] allows serverless functions to leverage host mechanisms to exchange data, decreasing latency. Nevertheless, these approaches decrease the function isolation [MN23, SWN+22]

To ensure concurrency in distributed environments, actors only process a single message at a time. However, they may influence future messages behaviour. For example, if one actor is unavailable, it can reject or keep the next message waiting for processing. The current approach using middleware with event-triggered invocation ensures the current message is delivered, but it does not allow actors to influence future messages. To ensure that actors can effectively influence the behaviour of future messages, the event trigger invocation mechanism must guarantee that the actor's future processing requirements can be met. The actor's current LifeCycle state determines the behaviour of future messages.

1.2 Motivation

1.2.1 Illustrative Scenario

To better motivate the research challenges, we present a use case for real-time video analytics that focuses on detecting fire emergencies in smart cities. To achieve this, cameras and sensors are strategically positioned throughout the city to detect fire patterns. A serverless workflow is employed to identify and respond to fire emergencies.

Our workflow utilizes five serverless functions, partially executed on the Edge and partially executed on the cloud. To reduce communication latency in our workflow, some tasks are executed at the edge, close to the data source. Edge tasks are responsible for processing large real-time video streams, extracting image frames, simple object detection, and triggering immediate local alerts for emergencies. On the other hand, tasks that require more powerful computing resources, such as more complex object detection and model training, are carried out in the Cloud. Our motivating scenario is inspired by a Serverless Workflow for real-time environmental monitoring [ERGC24, SCC+23, MTZ23]

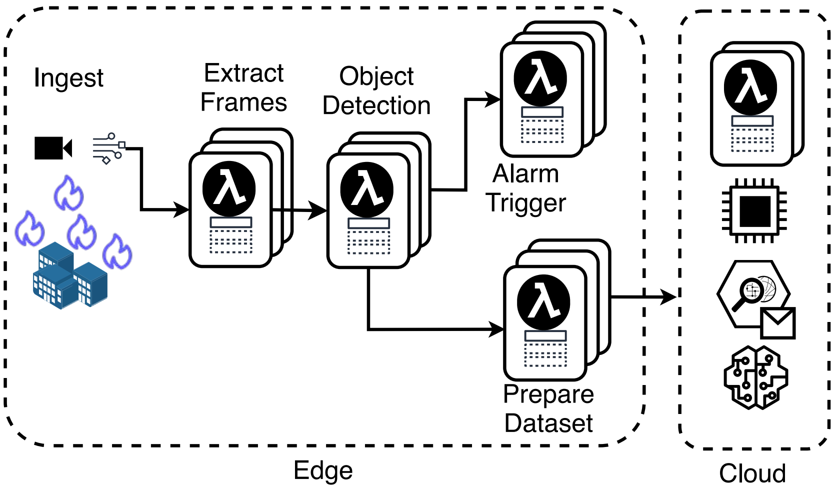


Figure 1.1: Simplified Serverless Workflow for Fire Detection for Smart Cities

In 1.1, in the *Ingest* stage, real-time videos captured by cameras are transmitted to edge nodes via a streaming framework, where serverless functions responsible for *Extract Frames* are activated to process the video data in small chunks, effectively reducing latency. Each *Extract Frames* function processes a video segment, ensuring swift data handling. Upon completing their tasks, these functions directly pass the processed frames to the *Object Detection* functions, who analyse them to identify specific fire patterns such as smoke and flames. Following the detection process, the *Object Detection* functions communicate directly with the *Alert Trigger* functions, who evaluate the data to decide whether to trigger local emergency responses.

Concurrently, *Object Detection* functions send data to *Prepare Dataset* functions for data preprocessing. Finally, the processed data is transmitted to the cloud, where more resource-intensive tasks are performed, such as training machine learning models to enhance fire detection. SIWA enables this workflow execution to minimise latency by facilitating actors communication directly instead of relying on storage services. Thus ensuring that messages are processed without data race conditions between other actors.

Additionally, it optimises edge resources as each actor is reused instead of creating one instance per invocation on conventional state-of-the-art serverless platforms. Finally, it enables actors to define the behaviour of the next message; actors can choose to keep the next message in the queue waiting for processing or reject it completely. Another actor then processes the message. Hence, it enables stateful actors to define the message sequence they will process.

1.3 Contributions

Current state-of-the-art approaches that enable serverless actors enable direct communication and state preservation while leveraging the existing serverless lifecycle and event-triggered invocation. As a result, multiple actor instances are created, and they leverage external services to exchange data and store their state. Current state-of-the-art serverless functions rely on containers and virtual machines for isolation and execution. This leads to a slow start, resource overhead, and regular updates and security checks. Serverless platforms do not support all languages, and some languages are slower in the serverless context. This leads to being restricted to a set of languages to run functions.

To address these issues, we propose novel serverless lifecycle, invocation models, and distributed middleware that enable actors to be reused, facilitate serverless actors to influence the behaviour of future messages, and allow communication between actors. We rely on WebAssembly for isolation and function execution instead of containers and virtual machines. WebAssembly offers improvements in performance without weakening isolation or security. Many languages also support WebAssembly as a compilation target, which makes it easier for developers to implement their desired functions. Finally, we present SIWA, Serverless Independent WebAssembly Actor based serverless platform that executes serverless functions as actors.

The main contributions of this paper include:

- *LCM: A novel SIWA Serverless Lifecycle Model* that natively executes serverless functions as actors. It allows serverless actors to be reused while enabling actors to preserve their state between multiple executions, thereby reducing the number of repeated actors instantiating for multiple requests. Each state is responsible for a specific task, such as registering an actor or running a function. Actor LifeCycle switches between states until the actor is no longer needed. The actor waits for new execution requests without wasting resources. When a new request is received, the actor switches to the execution state, reusing the allocated resources, such as the WebAssembly VM, making the execution faster and less resource-consuming.

- SIM: *A novel SIWA Serverless Invocation Model* that facilitates serverless actors to influence the behaviour of future messages. SIM enables busy actors to either reject future messages or keep them waiting to process them as soon as the actor becomes available. The Invocation Model relies on the Actor LifeCycle. The current actor state defines the behaviour of the incoming messages. For example, an actor in an execution state cannot accept new execution requests. The actor informs the distributed middleware of its availability to accept new execution requests.

- *SIWA: A WebAssembly Serverless Actor Platform* that leverages Wasm to provide lightweight isolation. The SIWA architecture leverages the LCM model to enable serverless functions to execute as actors. The user can compile their code into a WebAssembly binary and run the binary in the actor using the Wasm VM.
- *SIWA Middleware:* Furthermore, SIWA introduces its dedicated message middleware, which enables direct communication and leverages SIM to enable actors to influence the behaviour of future messages. The distributed middleware supports the actor's characteristic of being addressable. Functions running on different hosts can communicate through the distributed middleware using the unique ID of the actor.

1.4 Research Questions

The actor model's ability to deliver messages across distributed systems aligns with the dynamic and decentralised Serverless at Edge-Cloud Continuum [Hal12]. However, the current Serverless design limits its full potential [BPP$^+$19, POPL18, SCH24]. Therefore, we identify the following research challenges to enable the full capabilities of Serverless Actors in the Edge-Cloud Continuum.

RQ-1: How to enable serverless actors to be reused in the Edge-Cloud Continuum?

The current serverless function lifecycle supports either succeeded or failed states, which leads to platforms creating multiple instances for handling different function executions. Current approaches for serverless actors preserve their state in remote storage and load the previous state into the new instance.
Virtually, the new actor instance has the previous state, but physically, it is a new process on the host. Due to the current serverless lifecycle design limitation, every request is a new actor, which requires actors to leverage external services to maintain their state.

By enabling actors to be reused, the actor can preserve their state between executions, avoiding external services to persist their state, potentially reducing the number of created instances and preserving resources and costs at the edge [Mun19, HKO21, CDTV24]. Existing allocated resources, such as Wasm VM, are reused, leading to increased performance and decreased resource consumption.
The distributed middleware is aware if the actor is available to receive new requests and be reused, since the actor informs the middleware of its own availability. The actor exists for a specific duration, and the duration is renewed after each function execution. Idle actors stop running after this duration.

RQ-2: How can we enable direct communication between actors while allowing them to influence the behaviour of future messages?

Direct communication among serverless actors requires addressability. By enabling direct message exchanges between actors, they avoid using external services to exchange data, thus reducing latency and network overhead.
Nevertheless, the state-of-the-art event-triggered serverless function invocation enables single message delivery, which means the platform cannot decide which function executes the message. To enable actors to influence future messages, the event-triggering middleware must:

- Forward to the actor for processing
- Enables actors to keep the message in the middleware until the actor becomes available again.
- Forward to another actor in case of rejection by the existing actor

By enabling actors to influence the message processing behaviour, actors can optimise state management by keeping state between executions, thus decreasing latency and network traffic overhead, crucial for enhancing performance in sensitive edge environments [ACR+18]. Each actor has a unique ID assigned. The actors can communicate directly using their IDs through the distributed middleware.

RQ-3: How to provide lightweight isolation while enabling the full potential of serverless actors in the Edge-Cloud Continuum?

Isolation is critical to ensuring that failures by one actor do not impact others. Wasm provides a secure, sandboxed environment that reduces the overhead associated with traditional container-based isolation methods. This lightweight isolation allows serverless actors to execute with minimal latency and resource consumption, which is crucial for the Edge-Cloud Continuum. Furthermore, actors can profit from the reduced cold starts, decreasing their startup time. [GFD22, MPFS22].
In the following chapters, we discuss how WebAssembly provides this isolation independent of the compiled code source. Other platforms require pre-packages and platform-specific libraries that WebAssembly does not require, making it even easier for the developer to focus on the functionality. Each actor contains a dedicated Wasm VM to execute the provided WebAssembly binary file. This VM is initialised once, and due to Actor LifeCycle, it can be reused again.

1.5 Methodology

The aim of the thesis is to investigate the effect of using actors with extended functionality and dedicated distributed communication middleware on the FaaS. We start our research by studying papers dealing with serverless computing, specifically FaaS. We focus on state-of-the-art FaaS that use actors to execute their functions. We also check papers that utilise distributed middlewares for communication. We design and implement our system, going through different requirements and technologies. We measure the performance of our system by applying different experiments and comparing the results with other state-of-the-art frameworks.

1.6 Structure

In Chapter 2, we explore the history and background of related topics.

Chapter 3 goes through the process of designing our system, taking different requirements into consideration.

In Chapter 4, we discuss the implementation and the different aspects of SIWA. We go through the components of the system and how they work together.

Chapter 5 demonstrates the performance of SIWA by applying different experiments and comparing the results with other frameworks.

In Chapter 6, we mention relevant work that also deals with similar aspects, such as actors and distributed communication.

Chapter 7 summarises our work by mentioning revising the research questions, discussing future work, and mentioning the main contributions of our work.

CHAPTER 2
Background

In this chapter, we cover different terms and methodologies related to serverless computing, WebAssembly and actors and have an overview of them.

2.1 Edge Cloud Continuum

Organizations can run and host their web-based businesses and websites by themselves. Hosting and maintaining them became more challenging, especially in terms of security and scalability. Organizations started seeking different approaches to hosting and running their businesses in order to save costs and time.

Cloud computing started emerging and sparked interest from different parties as an alternative to traditional self-hosted servers. There are several advantages that make Cloud Computing attractive [AMF^{+}09, JSS^{+}19]:

- The aspect of on demand computing resources.
- No up-front obligation by the users.
- Ability to pay only for used resources in short terms.
- Scalability depending on the usage leads to reduced cost and avoiding large data centers.
- Simple and clear operation and configuration due to resource virtualization.
- Increase hardware utilization by coupling different workloads from several organizations.

Companies started offering computing resources using virtualization. Amazon EC2 [1], Microsoft Azure Virtual Machine [2], and Google Compute Engine [3] provide low-level, lightweight virtual machines that can be configured. Users configure the Virtual Machine (VM) that run in the cloud, where they can define their own custom required functionality.
This increased the encouragement for switching from locally hosted servers to cloud-based solutions. An important point that makes this approach more appealing is that the hardware is made available from the providers, and the user is responsible for defining the required computational power and resources.

Creating and maintaining VMs is not simple. Users sometimes want to execute only a certain task, such as uploading a file, sending a notification, or storing data in a database. For these tasks, users need to setup a VM and customize it according to its functionality. But creating a VM for simple tasks also requires a significant amount of time and cost, particularly when the base structure, such as the operating system and the language, is common between functionalities and needs to be configured again. Also, there are other challenges, such as scalability and security [JSS+19]. For these reasons, Function as a Service came up.

2.2 Serverless Computing

2.2.1 Function as a Service

Function as a Service (FaaS) or serverless computing is very popular in the cloud computing world. FaaS means that an application is deconstructed into several function-level microservices. They are short-lived, event-driven, and provide function isolation [LGC+22]. They are not only suitable for cloud platforms, but they can also run on edge devices. There are multiple cloud providers that support FaaS, such as Amazon Lambda [4] and Azure functions [5].

Taking the asynchronous functions shown in Figure 2.1 as an example, these functions are invoked by events or by sending API queries. After validating the queries, the functions are invoked in a new sandbox (Cold Start) or reused in an existing sandbox (Warm Start). The functions run individually and are isolated in a sandbox represented as a virtual machine (VM) or as a container. They can be executed on-demand and scaled horizontally

[1] https://aws.amazon.com/ec2/
[2] https://azure.microsoft.com/products/virtual-machines/
[3] https://cloud.google.com/products/compute
[4] https://aws.amazon.com/lambda/
[5] https://azure.microsoft.com/en-us/products/functions/

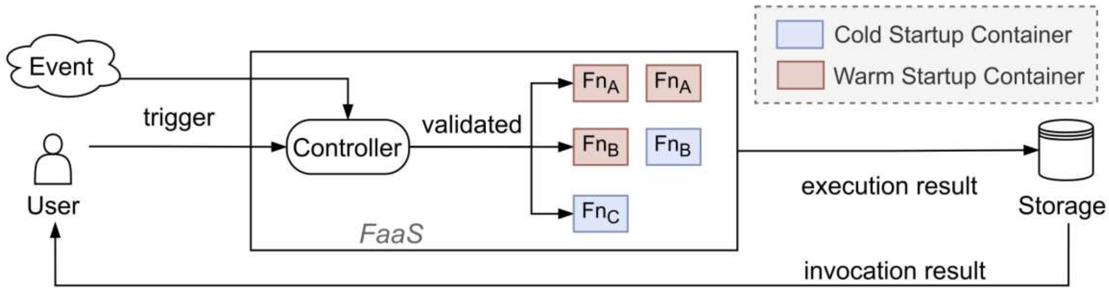


Figure 2.1: Asynchronous invocation in serverless computing [LGC+22].

depending on the workload [LGC+22]. Functions should support the following features in order to be identified as FaaS [LGC+22]:

- Auto-scaling: The functions should be able to scale horizontally and vertically. When the number of instances scales to zero, accessing the function next time leads into cold start. This means initializing the function from the start, introducing a slower start, and affecting the server response.
- Flexible Scheduling: The server schedules functions according to resource consumption. The servers are distributed into multiple regions, and according to the workload, the functions can be scheduled to run in different regions.
- Event-driven: The functions can be triggered by different events, message queue updates, or updates in a storage service.
- Transparent development: Users are not responsible for maintaining the underlying physical host, the providers are responsible for providing function isolation, sandboxing, execution environments, computational power, and many other resources. They also provide various DevOps tools to the developers.
- Pay-as-you-go: The user pays only for resources that the function actually used. This is possible by sharing CPU, network and disk and other resources among functions. The user does not need anymore to buy dedicated servers.

2.2.2 Virtualization

In this section, we go through various virtualization techniques that are used for running FaaS.

FireCraker

FireCraker [ABI+20] is a MircoVM that relies on Linux Kernel KVM [KKL+07] virtualization. The idea behind it is to rely on components that are part of Linux instead of creating new ones. FireCraker has been used in AWS Lambda since 2018.

gVisor

gVisor [YZCH+19] is a container based sandbox and implements an Open Container Initiative (OCI) runtime called runsc. Its main focus is to provide more security by restricting the number of syscalls to the kernel. gVisor is being used in Google Cloud Functions [6].

Hyper-V

Hyper-V [Mic] uses containers and microVMs. Each container instance runs in an isolated MicroVM with a dedicated kernel, providing isolation between containers. Microsoft Azure Functions [7] use Hyper-V.

Unikernel

Unikernel[MMR+13] is a standalone kernel that is compile-time specialized. The goal is to reduce image size, improve cost, security and efficiency.

2.2.3 Challenges

The mentioned sandboxes face multiple challenges:

- Cold Starts: Every time a function needs to be executed, a Virtual Machine or a container starts up. This introduces overheads and slower starts [SP20].
  Hyper-V uses containers and microVMs, offering strong isolation, but this virtualization method makes the start-up time slow. gVisor controls the syscalls in the kernel, making applications that use a large number of syscalls, slower. FireCraker is using a strong isolated microVM, but still, the initialization for mircoVM requires a long time in different cases, such as starting JVM.
- Runtime: These virtualization techniques require language runtime stacks, including operating system libraries and packages [SP20]. Unikernel is not flexible once built. This makes it challenging for developers to implement their applications.
- Security: Containers can introduce multiple security challenges such as Meltdown [LSG+18] and Spectre [KHF+19].
- Memory Footprint: Containers have a relative fast start up, but they introduce large memory footprint, effecting scalability [SP20].

Solving these problems can not only save costs but also optimize functions, reduce resource consumption and deliver faster executions, especially when running on edge devices.

[6]https://cloud.google.com/functions
[7]https://azure.microsoft.com/en-us/products/functions

2.3 Edge Computing

Cloud computing is becoming more popular, and numerous devices, such as mobile phones, cameras, and sensors, are capturing data and sending it to the cloud for computation purposes. But these devices started gathering more and more data that needed to be processed. Transferring large amounts of data is very challenging since the bandwidth and quality of the network play an important role, and the bandwidth is not increasing in proportion to the sent data. Furthermore, sending data across the network increases the latency, affecting performance. For the reasons mentioned, there is a tendency to make the computations directly on edge devices instead of sending them to the cloud. This is still challenging since the computation power and resources available on the edge are less than on a cloud server. Tasks running on the edge should consume as few resources as possible and still perform almost equally to those in the cloud. For these reasons, frameworks running on the edge should be lightweight and efficiently use resources.

2.4 Cold Start

In this section, we mention how some platforms try to reduce the cold start:

2.4.1 OpenFaas

OpenFaas is a serverless platform that relies on Kubernetes [8] and Docker containers to deploy and run functions. OpenFaaS tries to improve the cold start by considering the use cases of a cold start [Ell]:

- Initialization steps require creating the sandbox from scratch and fetching the required libraries and packages from the start.
- Horizontal scaling, depending on the workload required to create new replicas.

OpenFaaS tries to resolve the cold start by making sure that at least one replica of the function exists, assuming that the workload and number of functions are less than the workload that larger companies have to deal with.
In this way, the cold start eliminates completely, but it is also possible to scale the number of functions to zero by activating the cold start option in the configuration.

[8] https://kubernetes.io/

2.4.2 Preprediction

There are several methods to predict the execution of a serverless function. Xu et al. [XZG+19] predict the invocation time of the functions to prewarm and initialize containers by taking advantage of Long Short-Term Memory networks to analyze dependence relationships relying on historical traces.

2.5 WebAssembly

WebAssmebly (WASM) is a binary code format that is presented as a language. Since Javascript can have inconsistent performance and other challenges, WebAssembly came up as a solution for them. It focuses on safe, fast, portable code that can run not only on the browser web but also outside the browser. Multiple languages, such as C, C++, and Rust, support WebAssembly as a compilation target, as shown in Figure 2.2.

It does not need pre-packaged file systems or low-level operating-system specifics. It is a stack-based virtual machine [MPMB20]. WebAssembly also provides better sandboxing by introducing Software-based Fault Isolation (SFI) [Str98].

In order to run WASM outside of the browser, WebAssembly System Interface (WASI) [9] was developed. This interface provides different APIs for Wasm runtime with POSIX style capabilities in order to access multiple resources of the system [MPMB20].
It should be also possible to run the generated byte code on the hardware. For that, there have been multiple runtimes developed such as wasmtime [10] and wasmedge [11] that map the byte code to native hardware instructions.

2.5.1 Components

We mention some of the WebAssembly Components [HRS+17]:

Modules

A binary is represented as a module containing definitions for functions, globals, tables, and memories. Each definition can be imported or exported. The instantiation of modules is provided by an embedder, such as an operating system or Javascript VM.

Functions

Modules consist of functions. Functions can have parameters and return types using WebAssmebly values.

[9]`https://github.com/WebAssembly/WASI`
[10]`https://wasmtime.dev/`
[11]`https://wasmedge.org/`

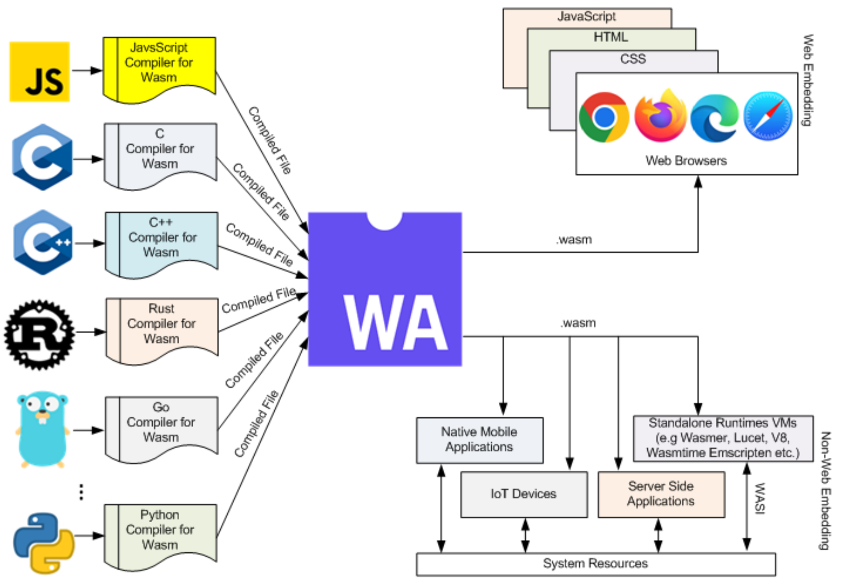


Figure 2.2: WebAssembly Use Cases Overview [Ray23].

Instructions

It uses a stack machine. Each instruction is popped from the stack, and values are pushed on the stack.

Traps

Traps are thrown when something faulty or unexpected occurs while executing an instruction. These traps should be handled by the embedder.

Machine Type

WebAssembly supports integers and IEEE 754 floating point numbers, each in 32 and 64 bit width.

Linear Memory

It uses a large array of bytes known as linear memory. Each module has its own memory that can be shared using import/export. It offers instructions to create, grow, load, and access memory.

2.5.2 Security

Linear memory is separated from the execution stack and engine data structure. This ensures that the compiled code cannot access other locations, and it can only corrupt its own memory. This facilitates running untrusted code, even compiled from other languages.

2.5.3 Webassembly Virtualization

We use WebAssembly VM as a virtualization method to run our function code. As mentioned in Section 2.5, it provides isolation and security using Software-based Fault Isolation. As shown in Figure 2.3, Webassembly does not need to package libraries and is also not relying on the kernel. It makes it easier for the developer to run their custom app without the need for pre-configuring system libraries. Each actor contains a Wasm VM, which executes the code. The WASM binary is loaded into the VM in order to execute the function. Languages provide different libraries to interact with the Wasm VM. There are multiple runtimes that offer Webassembly as a runtime.

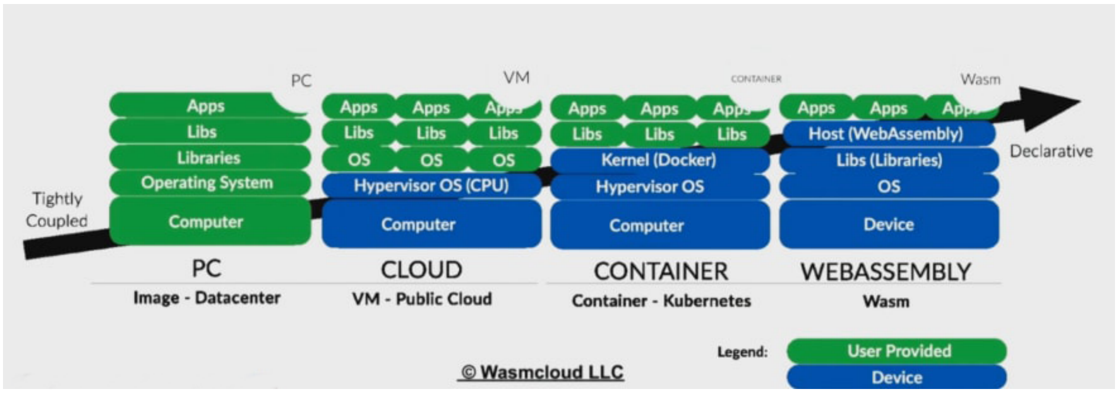


Figure 2.3: WebAssembly Virtualization[12]

2.6 Actor Model

Actor Model [Hew10] is a concurrent model that supports distributed systems. Actors send messages, as shown in Figure 2.4, between themselves without the need for a live connection. This enables asynchronous communication, which is fundamental for the actor model. Messages are sent to the target actor on a best-efforts basis, and the sender is no longer responsible for the message. Messages can be received in any order.

The Actor Pattern is becoming more popular and many languages such JavaScript Comedy[13], Python Pykka[14], Rust Actix[15] and Java/Scala Akka[16] are supporting it. These libraries implement the Actor Model in different variation.

[12]https://wasmcloud.com/
[13]https://github.com/untu/comedy
[14]https://github.com/jodal/pykka
[15]https://github.com/actix/actix
[16]https://akka.io/
[17]https://www.brianstorti.com/the-actor-model/

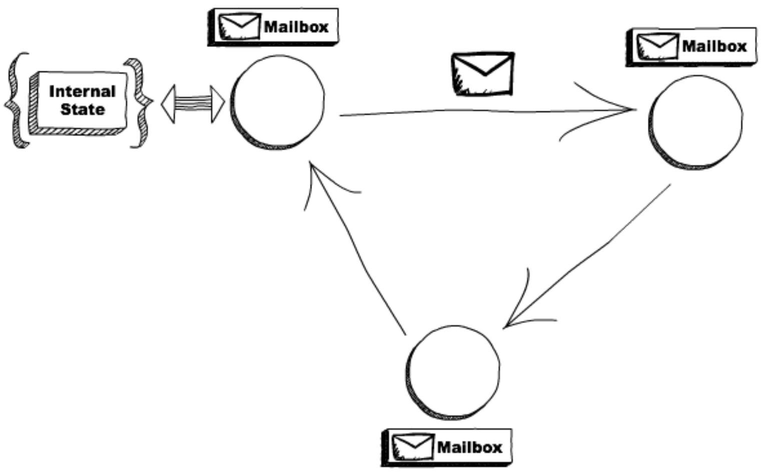


Figure 2.4: Actor Model Overview[17]

2.6.1 Features

An Actor offers the following capabilities:

- Communication between actors.
- Create new actors.
- Define behaviour when a new message is received.
- Concurrent execution when receiving messages.
- The receiver is responsible after getting the message from the sender.

2.6.2 Use Cases

There are multiple systems that can be modeled using Actors:

- Emails: Email accounts can be the actors, and email addresses are the actor addresses.
- Functional and Logical Programming
- Web Services: The services can be represented as actors and endpoints as actor addresses.
- Objects with Locks.

2.6.3 Advantages

- Scalability: Scaling the number of actors can be adapted according to the system load since actors can be created, paused, or stopped concurrently and independently.
- Actor State: Each actor has its own state, and other actors cannot access or change the state of other actors.
- Actor Communication: Each actor has their own address. This allows actors to run on different hosts/nodes and still, using their addresses, send/receive messages to/from actors running on different hosts/nodes.

CHAPTER 3

SIWA System Design

In this chapter, we discuss the design process for SIWA. We go through the different requirements for our system and discuss the goal and the desired improvement. We talk through actors, distributed communication, and virtualization. After knowing the requirements and the goals, we go through our system design, discussing the taken approaches and the core concepts.

3.1 Requirements

3.1.1 Statefull Functions

A major disadvantage of serverless functions is that the state and data of functions between invocations is not be stored. This leads to functions seeking external storage systems, such as databases or object storage, to store and retrieve the data with every new invocation. For the mentioned reasons, we make use of actor in order to support stateful functions, for the purpose of increasing the performance and not rely on other systems to exchange data.

3.1.2 Reusable Actors

After the actor finishes function execution, we want to make use of the same actor to process other requests. Reusable actors help to fully utilize allocated resources to execute functions. This leads to reduced latency, since the resources are already allocated and also reuse available resources. We want also to decide the behaviour of future messages that the actor receives.

3.1.3 Scalable Actors

Depending on the request load, we want to scale actors automatically in order to maintain the performance. We also offer scale-to-zero, where if there are requests currently processed, then no actors should run in the background.

3.1.4 Distributed Communication

Actors and functions need to be addressable and have the capability to exchange messages. Communication and addressability should be handled distributively, where functions running on different nodes and hosts should be able to communicate directly.
Also, functions should be able to be invoked internally or as part of a workflow. The communication should be scalable and handle different sizes of input and load.

3.1.5 Lightweight Virtualization

As discussed in Section 2.2.2, VMs and containers are used as a wrapper for functions. They offer strong isolation and security between functions, which often leads to slow cold starts, increased overhead, and wasted resources.
For these reasons, we should rely on a lightweight virtualization method without weakening isolation and security. It should maintain or decrease the cold start time and also offer strong isolation.

3.2 Core Concepts

3.2.1 Actors

We base our functions on the actor model. As mentioned in section 2.6, actors are suitable for executing specific tasks, where each actor is in control of its state and does not share it with other actors. Other external management systems are not required to manage the actors and the actor itself is responsible for handling provided messages. Each actor represents a function that can be executed, and the state is stored and updated in the actor itself. The actor relies on the WebAssembly runtime for running the function code with the provided input. We use the LifeCycle model as a method to persist in the state of the actor.
Actors can exchange messages through a distributed middleware, and they can be invoked as part of a workflow or through actors themselves.

Actor LifeCycle

The Actor LifeCycle consists of multiple cycles or states that are responsible for managing the actor's functionality. LifeCycles help the actors switch to different states without the need to rerun a previous or current state. This increases the performance and also helps the actor maintain its state. Also, using LifeCycles, actors can remain in a warm state and reuse resources for new executions.

3.2.2 Actor Dispatcher

The Actor Dispatcher is responsible for creating and forwarding the messages to the actors. The dispatcher subscribes to a specific topic and reacts to new messages available from the middleware. We can have multiple dispatchers in order to reduce the load and distribute the work across different dispatchers.

3.2.3 Distributed Middleware

The distributed middleware is responsible for communication between actors. It receives requests coming from external or internal sources and handles them. Each middleware consists of a messaging queue that makes sure that each message is passed exactly once to the interested subscribers. Each subscriber has its own logic for handling the message. For example, actor subscribers are responsible for passing the message to the actor. New subscribers can subscribe to the interested message content.
Each middleware has a wrapper, called a middleware interface, that is responsible for accepting requests. In case the middleware is not available, the wrapper does not accept the request. The request is forwarded to a different available middleware until it is accepted.

3.3 SIWA Models

3.3.1 SIWA LCM Serverless Lifecycle Model Overview

In order to optimise the actor itself, we created the Actor LifeCylce, where each actor has its own lifecylce. LifeCylce Model (LCM) provides a novel Serverless lifecycle tailored for serverless actors to facilitate state management and optimise communication by enabling serverless actors to preserve their state between multiple executions.
The functionality is spread across multiple states, where each state has its own defined strategy. The necessary resources are allocated once and reused in other states. In our case, the Wasm VM is created once with all configuration in a specific state and is reused afterwards.
Each state has its own defined steps and functionalities, shown in Figure 3.1, and is responsible for defining the next state that should be switched into. Actor LifeCycle consists of the following states:

- CREATED: An actor is created. Necessary resources are allocated, such as Wasm VM, and actors are addressable using the assigned unique ID. The actor informs using the channel about its availability to process new messages.
- IDLE: The actor is in an idle state. In this state, the actor is either waiting for new requests to execute or passing a specific duration to switch to the termination state. This state plays an important role by keeping the actor in a waiting state without consuming resources. The actor notifies about it readiness to accept new messages.

- ERROR: During execution, an unexpected error occurred, and the actor could not execute the function.
- RUNNING: The actor is running the function with the provided input. The actor cannot process other messages while the current execution is not done, and the channel blocks any new messages. The actor was in the *IDLE* state, where the resources were already allocated, and switched to the running state. The channel informs the middleware that the actor cannot accept any new messages at the moment.
- COMPLETED: The actor has finished the execution successfully. SIWA middleware is informed about the result, and the actor is marked as ready for new executions using the channel. The actor switches to the *IDLE* state, waiting for new messages.
- TERMINATION: The actor is terminated and cannot accept more messages. It deregisters itself, and SIWA does not forward any messages to this actor anymore. Any resources related to actors are free. The actor can enter the *TERMINATION* either after the *IDLE* state duration is passed or the actor switched from the *ERROR* state to the *TERMINATION* state.

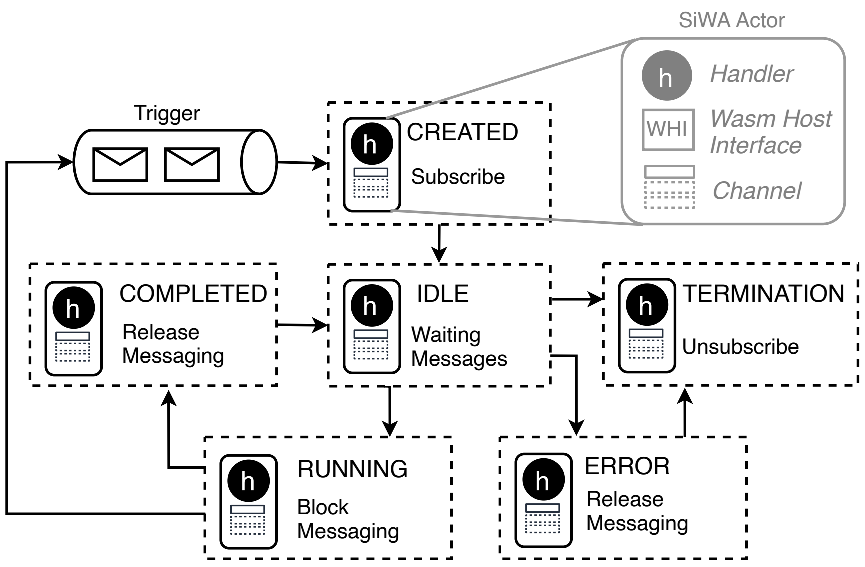


Figure 3.1: SIWA Serverless Lifecycle Model

SIWA Actor. 3.1 shows SIWA Actor and LCM Serverless lifecycle. SIWA actor is one entity composed of *Channel*, *Wasm Host Interface*, and *Handler*.
LCM optimises resource usage, reduces latency, and improves performance and scalability across the dynamic environments of the Edge-Cloud Continuum by ensuring that existing actors are efficiently utilised and consequently minimising the overhead associated with creating new actors.

3.3.2 SIWA Serverless Invocation Model Overview

The SIWA Serverless Invocation Model (SIM), in 3.2, introduces a new way of triggering serverless actors in response to events such as incoming messages. The SIWA SIM model ensures that new actors are created only when necessary, while existing actors are reused by introducing an actor message buffer. The SIWA SIM model enables SIWA Middleware to identify the availability and state of actors via the actor lifecycle phase. If the actor is *IDLE*, it transitions to the *RUNNING* phase to handle the message. If the actor is busy, the Middleware buffers the message or forwards it to another available actor, ensuring seamless processing without message loss. The SIM invocation model enables Serverless actors to process sticky messages.

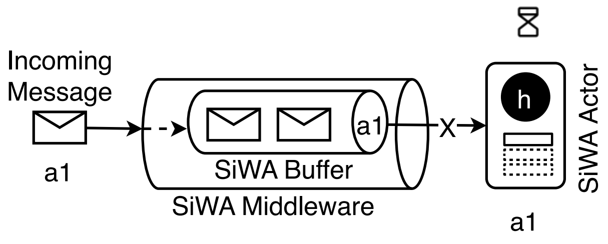


Figure 3.2: SIWA Serverless Invocation Model

SIWA Platform Architecture Overview

The SIWA Invocation Model design prevents data races by ensuring actors only handle one message at a time, thereby maintaining state integrity during the message delivery. Additionally, the SIWA Invocation Model enables actors to maintain state between executions and influence the behaviour of future messages, reducing the need for remote services to exchange data. By enabling serverless actors to influence future messages, the SIWA SIM Model avoids the use of remote services to store state and exchange data. As a result, it optimises resource usage and reduces latency, which is crucial for improving the performance of applications in the Edge-Cloud Continuum.

3.4 Actor Components

3.4.1 Channel

It is identifiable by a unique ID and serves as a dedicated communication channel for the actor. The behaviour of the channel is defined by the current state of the actor. Upon receiving a message, switching from state *IDLE* to state *RUNNING*, the channel sends a signal to the middleware to temporarily block any new incoming messages, ensuring actors process only a single message at a time. When the actor is in state *RUNNING*, the channel does not accept any new messages. In *COMPLETED* state, the channel informs the middleware about its availability. In addition, it enables actors to carry their previous state to the next one. Proactive message blocking ensures that each actor processes only one message at a time, preventing data races and maintaining the integrity of the execution process.

3.4.2 Wasm Host Interface (WHI)

It is a sidecar process that creates the Wasm VM with all neccessary configuration, allowing for secure, isolated execution of the Wasm binary. It acts as a mediator between the Wasm binary and the channel, forwarding the input and output from the binary to the message channel. Once the Wasm VM is created, it is reused in other states without the need to recreate or reconfigure it.

3.4.3 Handler

It encapsulates the user-defined code compiled into a Wasm binary file. Functions execute in a Wasm sandbox, which means a controlled environment that limits access to the host system, receiving inputs and producing outputs through the *Wasm Host Interface*. WebAssembly also provides secure isolation out-of-the-box.

3.5 SIWA Components

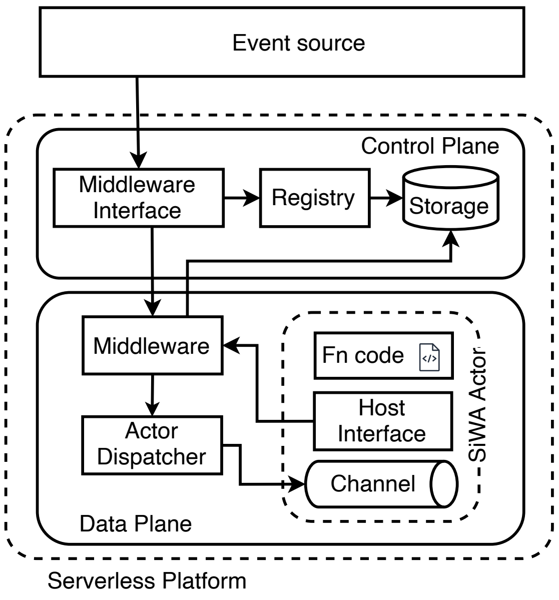


Figure 3.3: SIWA Architecture Overview

3.5.1 SIWA Middleware Interface

It connects *Middleware Registry* to all the middleware instances in different nodes. It ensures the messages are routed to the middleware at the correct node.

Each Middleware has a wrapper called the Middleware Interface. It's the first checkpoint for incoming requests before passing them on to the middleware itself. It controls the following:

- Middleware Availability: Each middleware processes messages from external sources. Since a message can be processed by any middleware, the load can exceed the limit or put a high pressure on a specific middleware, making the middleware busy with processing a high load of messages and causing a delay in consuming these messages.
  For this reason, the middleware interface communicates with the middleware and checks if it is still able to process a new message or if the message should be forwarded to a different middleware.
- Pre-processing: Each message should contain metadata. For new messages, the message is provided with the required metadata, such as the unique message and the current middleware ID. Also, it is possible to transform specific types of messages into a different structure before passing them to the middleware.

3.5.2 SIWA Middleware

As shown in Figure 3.3, it accepts the message and forwards it to the Actor Dispatcher. The Middleware Interface checks if the actor can process new messages; if not, it rejects the event. In case of rejection, the message is kept in a buffer or forwarded to another Actor Dispatcher until it is accepted. The message behaviour is defined by the actor, who can choose to receive the next message or reject it. The middleware ensures that an actor processes a single message exactly once.

3.5.3 SIWA Buffer

It is a queue for the busy actors, keeping waiting messages, thus allowing actors to influence the sequence of processing messages. It's also possible to redirect the message from one buffer to another to optimise message processing.

3.5.4 SIWA Middleware Registry

It maintains a reference to the middleware across different nodes. When a middleware initiates, it registers itself within the registry. This registration enables the middleware proxy to route messages accurately to the designated actor. This supports the distributed processing of functions and scatters the requests across all nodes.

3.5.5 Actor Dispatcher

Actor dispatchers are subscribed to the function execution topic. Function execution events include information about executing a function with the provided input. The actor dispatcher creates an OCI bundle with Docker containing the actor, and the actor

itself is implemented in Rust.
Depending on the function Id, if there is an actor waiting for a new execution with the same function Id (Warm Function), the dispatcher forwards the input directly to that actor. The dispatcher is written in rust and provides an GRPC interface for communication.

CHAPTER 4

Implementation

In this chapter, we go through the implementation process of SIWA. We go through the different mechanisms and methods that are being used to realise the design. We discuss the details of an actor's LifeCycle and invocation model.
After that, we discuss how the actors are communicating through the distributed middleware and how the distributed middleware is implemented in detail. We talk through virtualization using WebAssembly and how it plays an important role in our framework.

4.1 SIWA Mechanisms

SIWA leverages the LCM Serverless Lifecycle Model and SIM Serverless Invocation Model to enable an actor-native serverless platform. SIWA platform relies on two key mechanisms: LCM Serverless Lifecycle Phases Management and the SIWA SIM Serverless Event-triggered Message Invocation.

4.1.1 SIWA LCM Serverless Lifecycle Phases Management

To execute serverless functions as actors, SIWA leverages the LCM to create and reuse actors. An overview of the states and their transitions is show in Figure 3.1. Figure 4.1 shows each phase and which services are necessary to enable the LCM. In ①, in 4.1, when the actor is *CREATED*, it subscribes to a specified channel with its unique ID. Then, SIWA Middleware stores actor references for future usage as a warm actor. *CREATED* is the initial phase where the platform executes tasks to prepare for the actor run, such as resource allocation, Wasm VM creation, and configuration. In the next phase in ②, the actor enters the *IDLE* phase for a specific duration, waiting for incoming messages. No resources are consumed in this phase.

In ③, a message is received, and the middleware retrieves information from the storage to identify the actor, check if it exists, and forward the message to the actor via the actor channel. Once the actor receives the message, it sends an event to the SIWA Middleware to block new incoming messages. SIWA middleware then updates the actor reference to the storage, finalising that this actor is busy and cannot receive any new messages. In ④, the actor completes the message processing and sends a signal to SIWA middleware to unblock the actor. SIWA Middleware updates the actor reference and marks the actor as available. The actor informs the middleware about the result. After this phase, the actor returns to phase ② to wait and receive new messages.

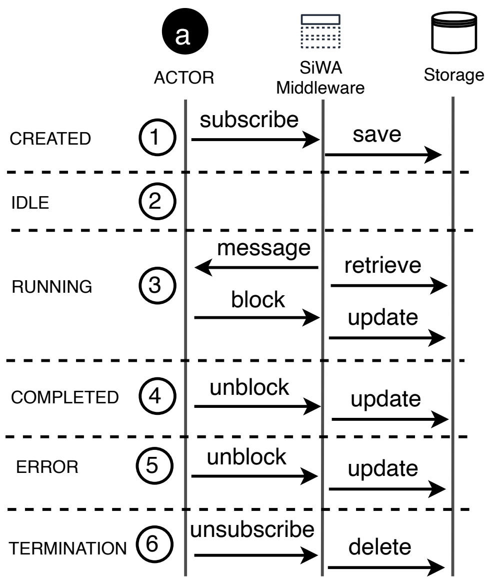


Figure 4.1: SIWA Serverless Lifecycle Management

After a period defined by the user, the actor moves from *IDLE* to the final phase *TERMINATION* in ⑥. Phase ⑤ represents an error state in the actor, the actor has either failed to startup or there was an unexpected error during execution. After entering the *ERROR* phase, the actor unblocks the message in SIWA Middleware, which updates the actor reference in the storage. In ⑥, the *TERMINATION* phase, the actor unsubscribes to the channel. SIWA Middleware deletes the specific reference to the channel and removes the actor reference from the storage. In this phase, the platform also releases reserved resources and removes any actor reference.

4.1.2 SIWA SIM Serverless Event-triggered Message Invocation

SIM is a novel Serverless Invocation Model that enables serverless actors to influence future messages. SIWA message middleware leverages the SIM model to trigger and exchange messages between serverless actors. SIWA actors decide the behaviour of future messages based on the actor input; the SIWA middleware decides whether to keep the message waiting in the buffer or forward it to the next actor.

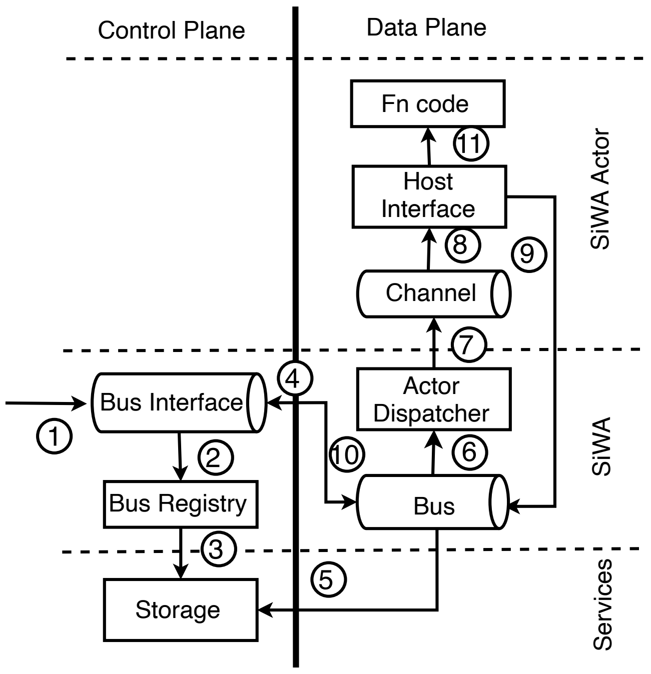


Figure 4.2: SIWA Distributed Messaging Middleware Flow

Figure 4.2 shows how SIWA Middleware distributes the message from the event source to the user function code. In ①, an event arrives at the Middleware Interface with the unique address of the actor. In ②, the Middleware Interface queries the Middleware Registry to find out if any existing node contains such an actor already. In ③ the Middleware Registry fetches from the storage existing actors information such as address and its availability. In ④, the Middleware Interface forwards the message either to the dedicated current middleware or redirects the message to another middleware if the current one is not free.
In ⑤ the middleware checks if any actors are available with the dedicated function ID that is waiting for new requests. In ⑥ the middleware forwards the message to the Actor Dispatcher or keeps the message in memory for future processing. To avoid multiple storage queries, the middleware also forwards the actor information, which is necessary for the decision-making in the *Actor Dispatcher*. In ⑦, the Actor Dispatcher creates an actor with its channel or forwards the message to an existing actor channel. This decision is dependent on the information passed from the middleware alongside the message.
In ⑧ the host interface receives the message from the channel and creates the Wasm VM, in case the actor is newly created. In ⑨, if the actor wants to create another actor, e.g., send a message to another actor, the Wasm Host interface also communicates to the middleware to send a specific message containing all necessary information. In ⑩, the middleware forwards the message to the middleware interface, which starts the process for the new message receiving from in ②. In ⑪ the *Host Interface* starts the Wasm VM with the user function code, and results are returned to the dedicated middleware

directly.

4.1.3 Message Execution Sequence

In this section, we describe the interaction between SIWA components in order to process a message. An invocation request is sent to SIWA API Gateway. The API Gateway forwards the message to an available middleware. The middleware interface checks if the dedicated middleware is available and has sufficient dispatchers. After confirmation of middleware availability, the interface forwards the message to the middleware.
The middleware looks up the function ID to check if any available warm actors are available for this specific function. The message with the gathered information as metadata is forwarded to an available dispatcher. Depending on warm actor availability, the dispatcher either creates a new actor or reuses an existing one with the provided ID. The actor, upon creation, registers itself to the middleware and is ready to accept new requests. The actor executes the desired function and returns the result to the middleware. The entire sequence is described in Figure 4.3.

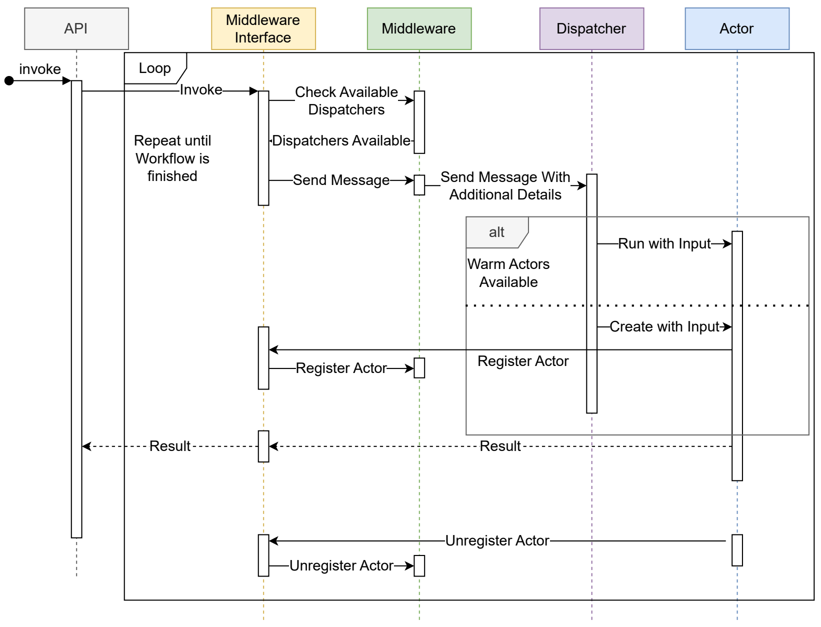


Figure 4.3: SIWA Execution Sequence

4.1.4 Message Execution Flow

In this section, we go through the control flow of processing a new message. The interface receives a new request, as shown in Figure 4.4, for executing a function. It checks if a dispatcher is available for processing new messages. If not, it forwards the message to another middleware until the message is accepted. If no available middleware is found,

the request is rejected.
After a middleware is found, the interface forwards the message to the middleware. The middleware finds an available dispatcher and passes the message to it. The dispatcher analyses the message to know if a warm actor is available. If there are no warm actors available, the dispatcher creates a new actor. The dispatcher delivers the message to the actor and is no longer responsible for the message. The actor executes the function with the provided input and transmits the result back to the middleware. If the flow is part of a workflow, the entire process is repeated until all steps in the workflow are executed.

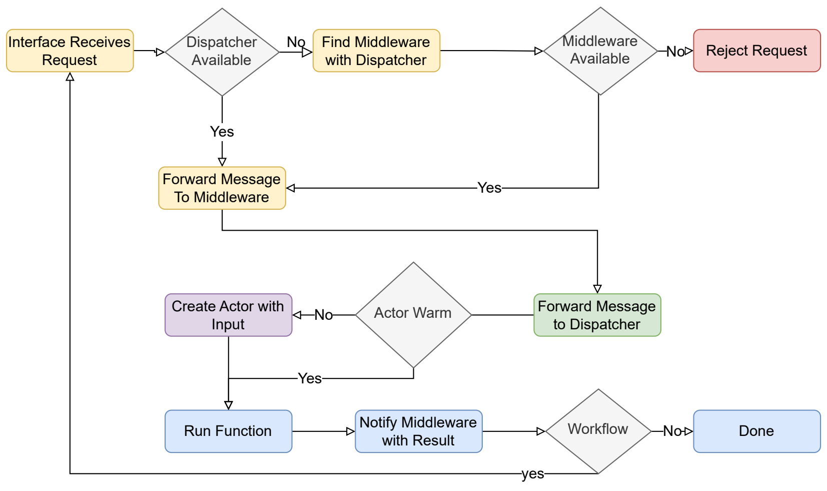

Interface Receives Request
Dispatcher Available
No
Find Middleware with Dispatcher
Middleware Available
No
Reject Request
Yes
Forward Message To Middleware
Yes
Create Actor with Input
No
Actor Warm
Forward Message to Dispatcher
Yes
Run Function
Notify Middleware with Result
Workflow
No
Done
yes

Figure 4.4: SIWA Execution Flow

4.2 Implementation

4.2.1 Actor

We start implementing the actor that is responsible for executing our functions. An actor is an execution unit that reacts to messages, and each message has a defined behavior. It can process one message at a time and is blocked until the message is processed. Each actor has a unique ID assigned upon creation that makes the actor addressable.

Actors are addressable and communicate through a distributed middleware, which is discussed in the following sections. The actors execute the functions by relying on Wasm VM. The actor retrieves the code, a `.wasm` file, from an external storage and loads it into the Wasm VM, which is responsible for executing the function. Each actor has its own Wasm VM, and the actor acts as a wrapper for the Wasm VM, as shown in Figure 4.5. In order to enable communication, each actor exposes multiple endpoints and external resources can interact with the actor endpoints using GRPC. The interface is defined in a `.protobuf` file that can be imported and used.

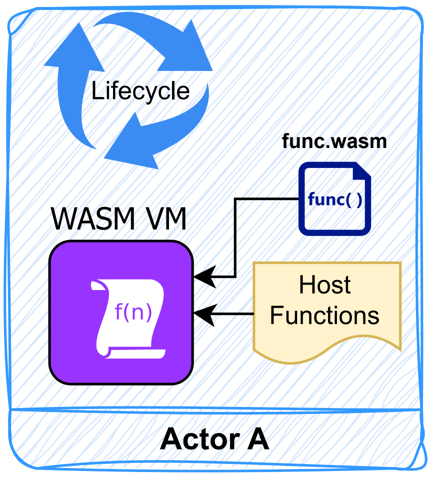


Figure 4.5: SIWA Actor

Actor Interface

Each actor exposes the following endpoints:

- Start: The actor is created and ready to accept new requests. It initializes the actor with all necessary components and creates the Wasm VM. Using Actor LifeCycle, the actor starts at state *CREATED*. This endpoint can be called once before creation.

- Run: The actor accepts the input and function ID. The actor fetches the function code from a storage using the function ID and loads the code to the Wasm VM. After the VM is initialized and ready to execute, it executes the function with the provided input. The actor LifeCycle has switched from *IDLE* to *RUNNING*.

4.2.2 WebAssembly Virtualization

We want the actors to execute our functions using virtualization. We do not not rely on containers and VMs for virtualization. We use WebAssembly for isolating and running our functions.

Webassmebly

Multiple languages support WebAssembly as a compilation target. We take Rust as an example, and we write our functions in Rust. After implementing the desired function, the code is built and compiled using Wasm32 as the target. The result is a `.wasm` file that can be run using any WebAssembly runtime.

WasmEdge

In order to execute the `.wasm`, we need a server-side WebAssembly runtime. WasmEdge is the fastest Wasm VM [LTHY21]. It supports WASI, microservices on edge cloud, IoT devices, and blockchain smart contracts [1]. Its recently also started being used for running Large Lanugage Model (LLM) [2] using less resources than other models that rely on Python.

In order to use it with actors, we need to install WasmEdge on the system that is hosting the actors. WasmEdege supports multiple platforms, such as Linux, MacOS, and Windows. We run our system on Ubuntu and install the required WebAssmebly and WasmEdge packages.

Rust and WebAssembly

We want to embed the WasmEdge VM in our actors. For that, WasmEdge provides a Rust crate called "wasmedge-sdk" [3]. This crate interacts with WasmEdge installed on the system using the WasmEdge C API. We configure the VM with the standard configuration, enabling WASI. As discussed in Section 2.5, we need to load our `.wasm` binary using modules. After loading the module in the VM, all related tables, functions, and globals are exported and available.

Since WebAssembly only supports integers and floating-point numbers, we also need to make it possible to pass strings or other data structures to our functions. We do this by asking the loaded module to provide an available memory address so that the input bytes are written directly to the memory by the actor. The same process happens when the function returns an object; we need to get the address of the returned value and the size of it so that the actor, after the function is done with execution, can retrieve the result of the execution.

4.2.3 Distributed Middleware

Actors are function execution components. Actors need to be addressable and should be able to communicate with each other. For this reason, we have created a distributed middleware that is responsible for facilitating communication between actors.

[1]`https://github.com/WasmEdge`
[2]`https://www.secondstate.io/articles/wasm-runtime-agi/`
[3]`https://github.com/WasmEdge/wasmedge-rust-sdk`

SIWA leverages actor model properties such as addressability, isolation, and state to enhance serverless function execution by transforming them into serverless actors [Hal12, POPL18, SCH24]. Each serverless actor in SIWA is uniquely identifiable, allowing for direct, addressable communication, thereby facilitating efficient data and message exchanges across the actors in the Edge-Cloud Continuum.

The SIWA architecture leverages Wasm to provide an isolated and secure sandbox for each actor. Moreover, SIWA's LCM manages the lifecycle of serverless actors, from initialization to termination. SIWA LCM enables Serverless actors to retain and efficiently manage their state, thus facilitating complex applications that require persistent state across sessions.

Event Processing

The distributed middleware is event-based and relies on the publish/subscribe pattern. Each message is an event, and the event is pushed into an event queue. An event can be on a specific topic, and each topic can have multiple subscribers, called dispatchers, that are subscribed to all events on that specific topic. Each event is delivered exactly once to a dispatcher. Each dispatcher can have its own defined behaviour, and new dispatchers can be added.

4.2.4 Communication

The internal components of SIWA communicate through GRPC [4]. There is also the possibility to use HTTP, but we went with grpc for the following reasons:

- Grpc payload is smaller than http payload size
- Multiple requests and responses can be sent using a single connection, increasing the performance.
- The communication interface is defined using a `.protobuf` file, that supports code generation for multiple languages. This file can be imported and used for communication between different components.

External sources have a REST API available to interact and send messages to SIWA.

4.3 Workflows and Internal Calls

4.3.1 Workflows

Functions can be executed by sending requests. We often need to execute a chain of functions or workflows to achieve our desired functionality. In SIWA, a workflow request

[4] `https://grpc.io/`

can be sent with the input and the workflow id. The metadata of the request also includes information about the current step and workflow id. This helps to keep track of the workflow when a function finishes execution and sends the result to the middleware. Internally, the workflow plan, a json file shown in Listing 1, is stored in an external system and can be retrieved upon request. The plan includes the order of the function execution, where the output of the first function can be defined as the input of the second function.

```json
{
  "start": "add-number",
  "add-number": {
    "function": "add",
    "next": "sub-number"
  },
  "sub-number": {
    "function": "sub",
    "next": "multiply-number"
  },
  "multiply-number": {
    "function": "multiply",
    "end": true
  }
}
```

Listing 1: Workflow Plan Example

4.3.2 Internal Function Calls

In many systems, there is a requirement to make async calls and call other systems. WebAssembly support for async calls is still not fully supported, which means that external calls cannot be done purely with WebAssembly. Functions often have the need to call other functions, and they can directly send a request to the middleware using GRPC calls. For the reason mentioned above, the actor who is wrapping the function enables this functionality by using host functions.

Host Functions

Host functions are provided as host modules, where the host functions, globals, and tables can be imported. In rust, to mark a function as an external function (host function in this case), we need to mark the function as `extern "C"`, as shown in Listing 2. The functions are implemented in rust, and they must be provided as part of an imported module from the actor itself. The function assumes that these external functions will be there at runtime, which means that the imported module must be loaded and registered to the VM so that the WebAssembly function can have access to the functions.

```
extern "C" {
    fn call_function(url_pointer: *const u8, url_length: i32,
                         pointer: *const u8) -> i32;
    fn get_memory(url_length: i32) -> i32;
}
```

Listing 2: Example of host functions used in Rust

As we notice in Listing 3, the macro `#[host_function]` is used. This implements a wrapper for the function. This wrapper is responsible for creating a host function instance that can be imported into the WasmEdge VM. `WasmValue` represents WebAssembly value type, and `Caller` makes it possible for the developer to access different WebAssembly instances, such as Execution instance and Memory instance.

```
#[host_function]
fn call_function(caller: Caller, args: Vec<WasmValue>) ->
                    Result<Vec<WasmValue>, HostFuncError> {}
#[host_function]
fn get_memory(_caller: Caller, args: Vec<WasmValue>) ->
                 Result<Vec<WasmValue>, HostFuncError> {}
```

Listing 3: Host functions provided by the actor

After implementing the function, we need to import it to the WasmEdge VM. As shown in Listing 4, we create an import object that contains our hosted functions. We make the imported function available under the same name as used in our WebAssembly function, as shown in Listing 2. This import object contains the host functions, global variables, memory, and table.

4.4 Execution Example

We write our desired function using Rust. The function, show in Listing 5, must have the name `run`, so that SIWA recognizes and executes it.

After implementing our desired functionality, we execute the following command in order to compile the code:

```
let import = ImportObjectBuilder::new()
            .with_func::<(i32, i32, i32,i32, i32), i32,
               NeverType>("call_function", call_function, None)
            .with_func::<(i32), (i32),
               NeverType>("get_memory", get_memory, None)
            .build::<NeverType>("env", None);
vm.register_import_module(&import);
```

Listing 4: Host Functions imported to WasmEdge

```
#[no_mangle]
pub unsafe extern fn run(input_pointer: *const u8,
                          input_length: i32) -> i32 {

}
```

Listing 5: SIWA Function

```
$ cargo build --target wasm32-wasi --release
```

This generates a `.wasm` file that is uploaded to a storage with a unique ID, where SIWA can download and load it. After importing the `.wasm` file, the module is created, and the function is read to be executed. In order to run the function, we need to specify the name of the created module, as shown in Listing 6, and the function name `run` 5, and provide the function parameters.

```
vm.run_func(Some("functionModule"),"run",
            params![input, input_length])
```

Listing 6: SIWA WebAssembly Function Execution

4.4.1 Function Execution

We can now execute our function by sending the request shows in Listing 7. SIWA receives the request and executes the defined function in `resource_id`. We indicate that we want to execute a function by passing `function` as `topic`, as shown in Listing 7. In `context`, we can pass the `input` that is required. The function is executed, and response is returned.

```
POST /invoke HTTP/1.1
Host: SIWA.com
Content-Type: application/json

{
    "context": {
        "input": "data"
    },
    "resource_id": "FunctionA",
    "topic": "function"
}
```

Listing 7: SIWA Function Execution Request

4.4.2 Workflow Execution

We can also execute a workflow by uploading a file, as shown in Listing 1, using a unique ID. The functions mentioned in the workflow plan should exist in SIWA so that they can be executed. We can execute the workflow by sending the request shown in Listing 8. The request structure is the same as a function execution request. SIWA receives the request and executes the defined workflow in `resource_id`. We indicate that we want to execute a workflow by passing `workflow` as `topic`, as shown in Listing 8. In `context`, we can pass the `input` that is required. The function is executed, and the response is returned.

```
POST /invoke HTTP/1.1
Host: SIWA.com
Content-Type: application/json

{
    "context": {
        "input": "data"
    },
    "resource_id": "PlanA",
    "topic": "workflow"
}
```

Listing 8: SIWA Workflow Execution Request

CHAPTER 5

Evaluation

In this chapter, we evaluate our SIWA framework using different loads and methods. SIWA source code can be found in Github [1]. We compare it with two other serverless frameworks, OpenFaas [2] and Spin [3]. Also, we use Redis [4] as a bus in OpenFaas and Spin. We go through our setup and determine what kinds of experiments are used and what results are achieved.

5.1 Overview

In this Section, we discuss the different frameworks that are chosen to benchmark SIWA. We go through different benchmarks that we apply to compare the performance of SIWA with other frameworks. Also, we talk about the different methods that are utilized for calculating our results.

5.1.1 Frameworks

There are multiple open-source serverless platforms on the market that are available to benchmark our framework. We chose the following frameworks:

- OpenFaas: OpenFaas is a serverless platform that relies on Kubernetes to manage functions. Each function is running in a Docker container that is managed by Kubernetes. They support multiple languages by providing default language templates, but the developer can also implement a new custom template by using Docker images. We chose this platform because it also relies on containers to wrap the functions.

[1]`https://github.com/JackShahhoud/SIWA/`
[2]`https://www.openfaas.com/`
[3]`https://www.fermyon.com/spin`
[4]`https://redis.io/`

- Spin: Spin is a serverless platform that relies on WebAssembly to run functions. It uses Wasmtime as the WebAssembly runtime to execute the functions. They offer triggers to support different functionalities, such as API interfaces and database interactions. We selected Spin since it also relies on WebAssembly to execute functions.
- Redis: The mentioned frameworks do not provide distributed communication methods between functions. For this reason, Redis is used for communication between the functions. We chose Redis because it is lightweight and offers Pub/Sub messaging. The mentioned frameworks rely on Redis Pub/Sub messaging pattern with default settings to exchange messages.

5.2 Benchmarks

There are different criteria that are relevant for evaluating FaaS that are provided by different frameworks. In the mentioned benchmarks, depending on the applied method, we apply different number of parallel requests and input size. We evaluate our framework based on the following criteria:

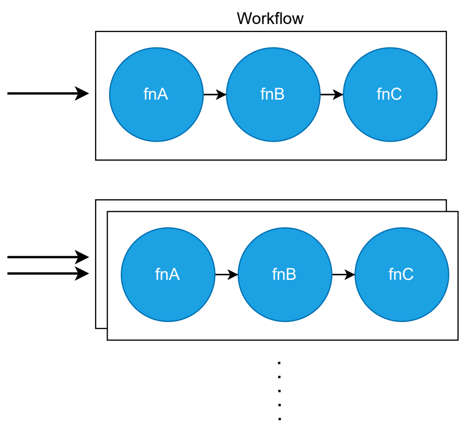
- Latency: This metric shows the execution time for the message passing between two actors. We use seconds and milliseconds for our latency experiments for sequential and parallel execution, respectively.
- Throughput: This metric measures the number of executions a framework can process in a specific timeframe. We measure the performance of SIWA under high loads. The goal of throughput experiments is to identify how many requests the application can process at a time and if there are bottlenecks in the proposed framework once the application load increases.

5.3 Experiments

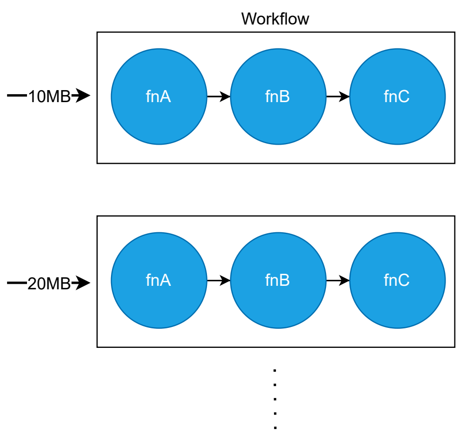
We apply different methods during our evaluation of the frameworks. This allows us to analyse the performance under different conditions. We use the same methods in all the frameworks and observe the results.

Workflows

This method uses a chain of function executions. It's a sequence of function executions where the output of a function is the input of the following function. We measure how the frameworks perform using different numbers of parallel requests sent, as shown in Figure 5.1a, and different input sizes, as shown in Figure 5.1b. SIWA uses its own implementation of the distributed bus, and the other frameworks rely on Redis for communication. Each workflow consists of three functions, where the functions are executed in sequence.



(a) Parallel Requests

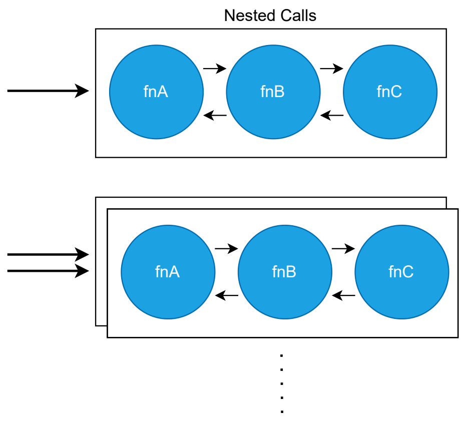


(b) Single Request with Input Variation

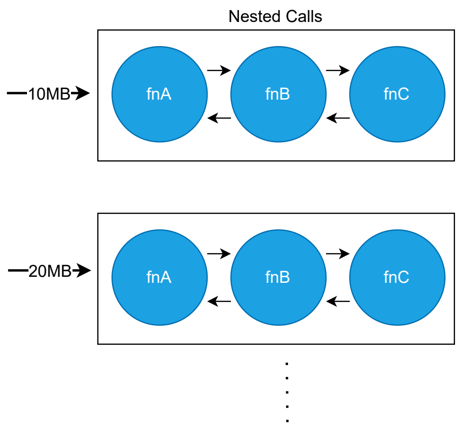
Figure 5.1: Workflow Execution

Nested Calls

In this experiment, each function during execution calls a different function. The current execution is stopped until the results of the called function are returned. We apply different numbers of parallel requests, as shown in Figure 5.2a, and different input sizes, as shown in Figure 5.2b. SIWA uses its own implemented distributed bus, and the other frameworks depend on Redis for exchanging messages. There are three nested function executions for each experiment.



(a) Parallel Requests



(b) Single Request with Input Variation

Figure 5.2: Nested Execution

5.3.1 Setup

To evaluate SIWA, we execute the designed experiments on a virtual machine (VM). The VM is running on Ubuntu 22.04 LTS, where the CPU architecture is ARM64 (AARCH64) with 8 GB of RAM, 4 cores, and 39 GB of storage. The experimental applications are written in Rust. The baseline applications used for the evaluation expose REST API endpoints for receiving and processing requests from external sources. For the HTTP requests, we use Rust reqwest[5] client, and Rust streams futures[6] for sending multiple parallel requests concurrently. To ensure the consistency of the results and avoid bias, we executed the experiments seven times and calculated the average as the desired result.

5.4 Results

In Table 5.1, we can see a sample of the performance using the latency as a benchmark. In this section, we discuss the results of the experiments and analyse them.

Experiment	Workflow-Parallel	Workflow-Input	Nested-Parallel	Nested-Input
SIWA	5.04	0.59	6.11	0.36
OpenFaaS	51.56	6.98	51.4	3.12
Spin	47.70	4.32	53.54	2.20

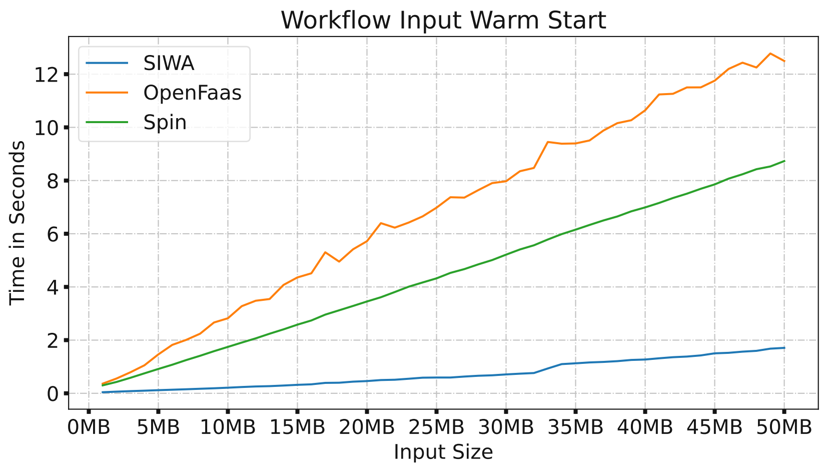
Table 5.1: Latency with Parallel (50 REQ) and Input (25MB) in Seconds.

5.4.1 Workflow

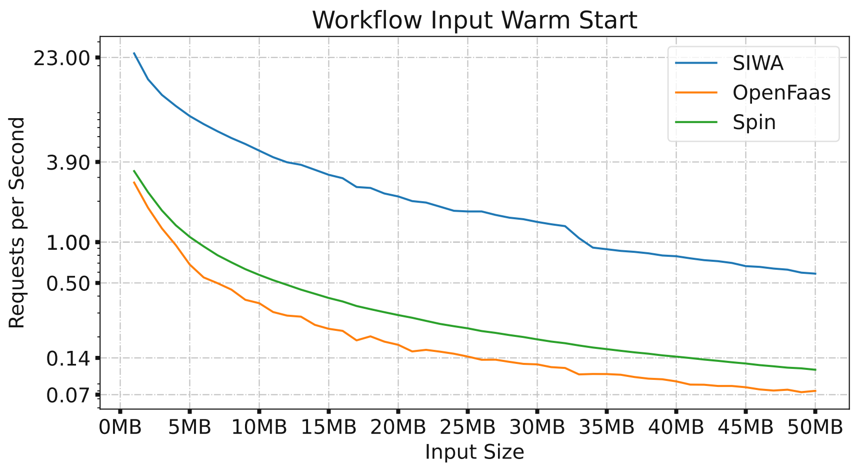
Workflow Execution using different Input Size

Figure 5.3a shows the input data size on the x axis and the latency in seconds on the y axis. SIWA displays response times ranging from 0.039 to 0.919 seconds, OpenFaaS shows an increase from about 0.272 to 6.144 seconds, and Spin's response time grows from 0.218 to 4.362 seconds. The latency analysis reveals that SIWA decreases the latency by up to 85% and 79% compared to OpenFaaS and Spin, respectively. These latency experiments show a significant latency reduction of SIWA, with all three systems demonstrating a generally linear increase in response times, indicative of stable performance across the increasing load. Figure 5.3b shows the throughput of SIWA, OpenFaaS, and Spin as increasing the input size. The x axis represents the input data size, while the y axis shows requests per second. Over axis x, SIWA's throughput decreases from about 25.93 to 1.09 requests per second, OpenFaaS declines from 3.68 to 0.16 requests per second, and Spin drops from 4.60 to 0.23 requests per second. All systems experience a linear decrease in throughput as the input size increases, indicating a linear throughput decrease with the input size. Additionally, SIWA maintains a throughput up to 6.8 times higher than OpenFaaS and up to 4.7 times higher than Spin.

[5]https://github.com/seanmonstar/reqwest
[6]https://github.com/rust-lang/futures-rs



(a) Latency



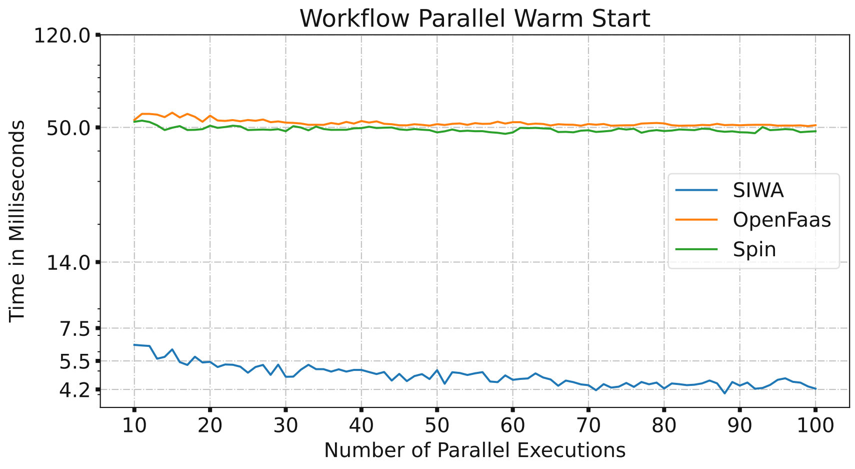
(b) Throughput

Figure 5.3: Workflow Execution using different input size

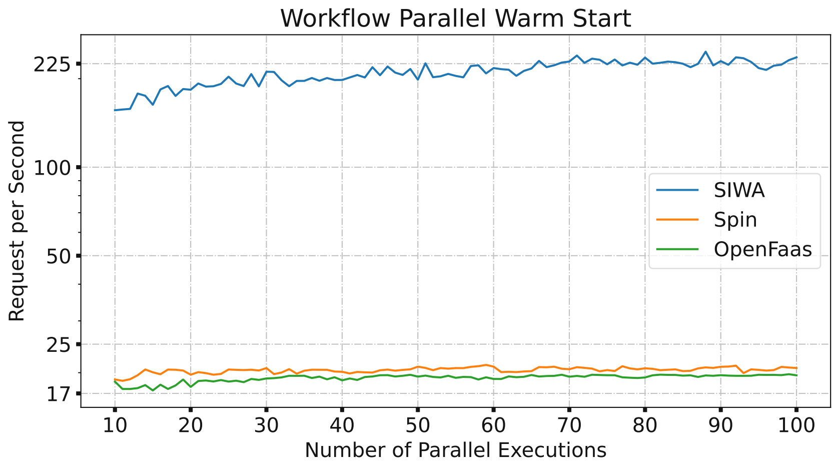
Parallel Workflow Executions

Figure 5.4a showcases the latency from parallel execution for Serverless Workflows, where the x axis indicates the number of parallel executions and the y axis measures the latency in milliseconds. SIWA demonstrates stability in latency, which ranges from 6.4 ms to 4.23 ms as the parallel execution count increases. In contrast, OpenFaas and Spin display higher latency similar to the nested functions, in 5.6a, around 50ms. Compared to the baselines, SIWA's latency is lower, showing an improvement of approximately 92% for serverless workflows. In Figure 5.4b, SIWA maintains a high throughput ranging from 156.25 to 236.41 requests per second. Both OpenFaas and Spin also show consistent throughput; however, they show around 20 requests per second, significantly lower than

SIWA. These results show that SIWA has up to 10x higher throughput compared to the baselines, showing stability for high-load serverless workflows while maintaining high throughput and low latency.



(a) Latency



(b) Throughput

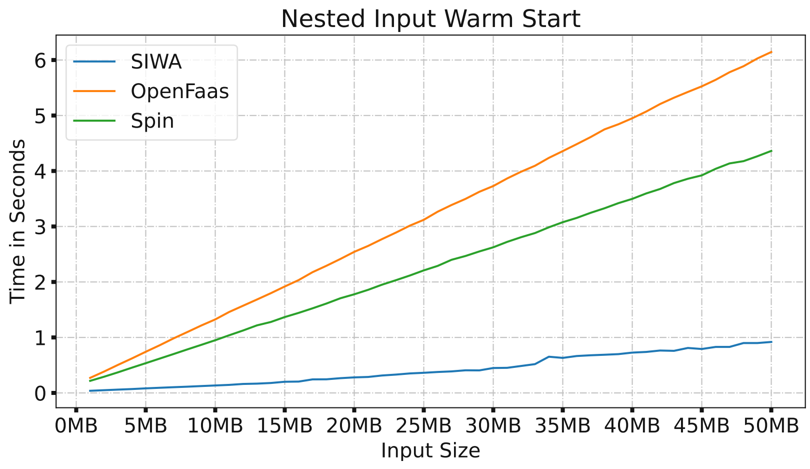
Figure 5.4: Workflow Execution using parallel executions

5.4.2 Nested Calls

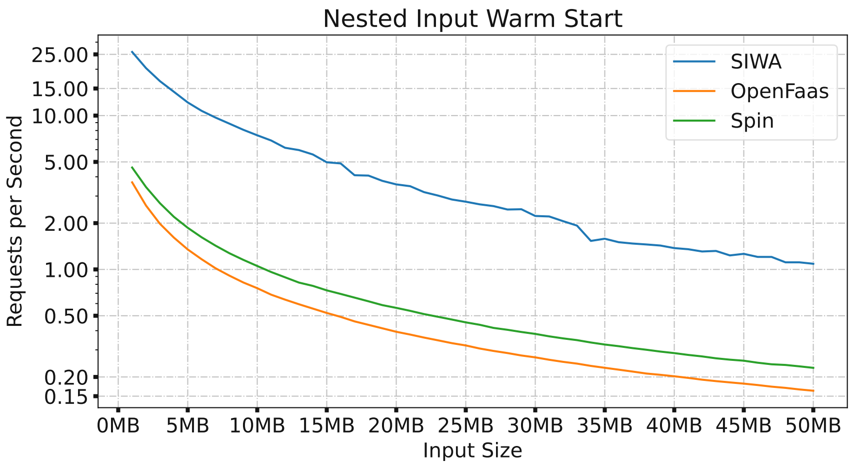
Nested Calls using different Input Size

Figure 5.5a presents the input data size in megabytes on the x axis and the response latency on the y axis. As input size increases, SIWA shows latency improvements ranging from 40 milliseconds to 1.71 seconds. OpenFaas displays latency from 363 milliseconds to approximately 12.5 seconds, while Spin maintains an increase from 299 milliseconds to

8.73 seconds. This experiment shows that SIWA reduces latency by up to 86% compared to OpenFaas and 80% relative to Spin. Figure shows the throughput metrics, where the input data size is in megabytes on the x axis and the requests per second on the y axis. SIWA displays a throughput decrease from 24.65 to 0.59 requests per second, while OpenFaas and Spin show reductions from 2.75 to 0.08 and from 3.34 to 0.11 requests per second, respectively. SIWA presents up to 7.4 times higher throughput than OpenFaas and up to 5.4 times more than Spin.



(a) Latency



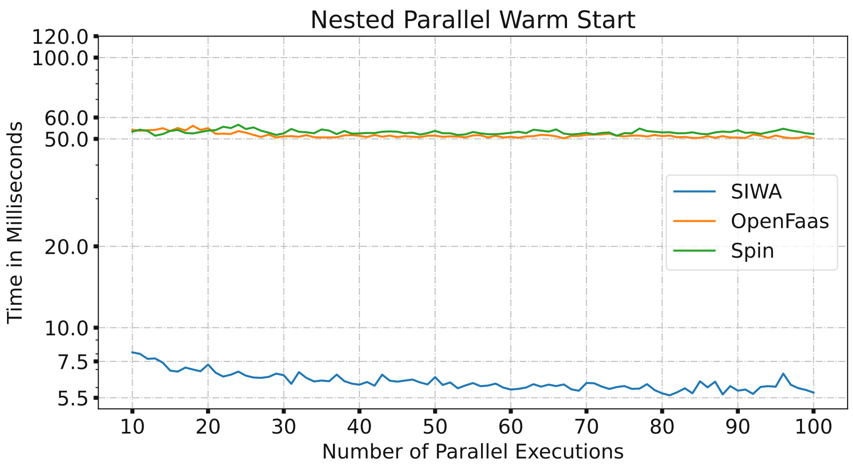
(b) Throughput

Figure 5.5: Nested Calls using different input size

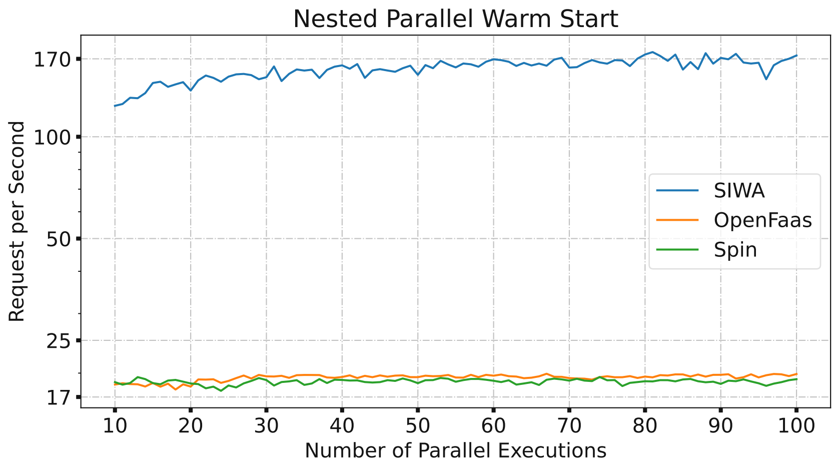
Parallel Nested Calls

Figure 5.6a presents the latency from the parallel execution experiments, where the x axis represents the number of parallel executions and the y axis reflects latency in milliseconds.

Figure 5.6a that SIWA maintains a relatively stable latency ranging from 6.9 milliseconds to around 5.75 milliseconds, even as the number of parallel executions increases. In comparison, OpenFaas and Spin exhibit slightly higher latency under higher loads, with OpenFaas and Sping showing a latency of around 50 milliseconds. SIWA shows up to an 87% reduction in latency compared to OpenFaas and Spin. In Figure 5.6b, SIWA maintains higher throughput, ranging from 123.45 to about 173.91 requests per second, which aligns with its efficient latency results under parallel operations in 5.6a. OpenFaas and Spin also display consistent throughput, with OpenFaas and Spin presenting around 50 requests per second even when the application load increases in axis x. Overall, SIWA has up to 9x higher throughput when compared to OpenFaas and Spin.



(a) Latency



(b) Throughput

Figure 5.6: Nested Calls using parallel executions

CHAPTER 6

Related Work

In this chapter, we present some related work, that partially rely on the mentioned methods and technologies.

6.1 Stateful Serverless

6.1.1 FAASM

Stateless containers do not allow to share memory directly, leading to the duplication and serialization of data repeatedly. For this reason, a new isolation method was needed to reduce overhead. FAASM [SP20] is a stateful serverless runtime based on WebAssembly. It uses Faaslets to provide lightweight isolation and efficient shared memory access. We count some of the Faaslets properties:

- Lightweight Isolation: It provides isolation by depending on software-based fault isolation (SFI). Functions with Faaslet are compiled with its library to WebAssembly. Each faaslet is executed in a dedicated thread within a single address space, where isolation is provided using cgroups and network namespaces.
- Efficient State Access: Faaslet uses a two-tier state architecture. The local tier allows in-memory sharing; the global tier supports distributed access. Also, Faaslets are in the same address space, which allows efficient memory access.
- Initialisation Time: FAASM pre-initializes a faaslet and stores its memory. When needed, this snapshot is restored to start a Faaslet.
- Flexible Host Interface: Faaslet provides a POSIX-like interface for accessing network, memory, and file I/O. It provides sufficient virtualization to ensure security.

Both local and global tiers need to be in sync to ensure message processing, which introduces network overhead when a large number of messages need to be processed.

6.1.2 Cloudburst

Cloudburst [SWL$^+$20] is a stateful FaaS that focuses on low-latency mutable state and communication without losing autoscaling capabilities. It achieves this by relying on key-value store (KVS) and mutable caches that are co-located with function executors. The system relies on Anna [WFLH18], an autoscalable, low-latency key value store. It takes use of the design to ensure consistently merging concurrent updates.

It accepts programs written in vanilla python and triggers them in the cloud. Result of the computation can be returned directly to the client or stored in KVS and retrieved later. Function parameters can be directly Python objects or KVS references that are fetched and deserialized before function invocation.
For optimizing the performance, it tries to execute function on a machine that can have the KVS reference value cached. To Enable statefull functionality, it offers Anna KVS API to directly store and fetch Python objects. Object serialization and encapsulation for consistency is handled by the runtime. Cloudburst relies on fault management of Anna. As other FaaS, it tries on retrun the function after faulty execution. Other unexpected side effets should be handled by the client. It might introduce duplicate cached data, leading to network overhead and duplicate serialization, a challenge for the limited resources of the Edge-Cloud Continuum.

6.2 WebAssmebly in Serverless

6.2.1 Spin

Spin [1] is a serverless framework that is based on WebAssembly using Wasmtime [2] as runtime. It supports multiple languages, such as Rust, Javascript, and Go. Other languages are supported, depending on the language's support for WASM and WASI features. Spin offers the following properties:

- Triggers: Functions can be invoked using triggers. An HTTP trigger can be used for requests sent using HTTP requests. Redis trigger puts Redis pub/sub to use, spin subscribers to messages coming from Redis, and reacts depending on the message. It offers an experimental cron trigger that executes at a specific time and frequency.

- Kubernetes Support: Spin can also run on Kubernetes. It implements containerd shim spin that uses runwasi to run spin functions.

[1] `https://developer.fermyon.com/spin/v2/index`
[2] `https://wasmtime.dev/`

Functions still need to communicate with each other over HTTP requests instead of internal communication, which increases the response time and affects the performance.

6.2.2 Krustlet

Microsoft wanted to make it easy to deploy WebAssembly workloads on Kubernetes and show how to build and develop Kubernetes architecture pieces in other languages than Go. Linux containers use OS-based virtualization.
It relies on the kernel to provide isolation and a sandboxed environment, which means that code compiled for Intel chips cannot run on other chips. On the other side, the Wasm binary format makes it possible to run it anywhere without depending on the OS or the hardware, but it is less flexible than the OS-based virtualization.

Krustlet[3] runs on Kubernetes Kubelet. It listens to the event stream for new pods. Kubernetes schedules pods into Krustlet, which runs the pods under the WASI runtime. But this should not replace Kubelet, but make them complimentary. Krustlet relies heavily on kubernetes, which increases complexity and makes it heavily depend on it to deploy and manage functions.

6.3 Actors

6.3.1 μActor

Several companies support running serverless functions on the edge. Aws Lambda@Edge, AWS Greengrass, and other services make it possible to execute functions directly on the edge near the user. These approaches introduce several challenges.
These functions are typically containerized. The increased complexity of managing these containers and problems, such as cold starts, should be avoided. Also, these functions are stateless and need other services to store and maintain data. These services, such as cloud-based storage, are not running on the edge, which means they are required to communicate through the network.

μActor [HKO21] introduces actor-based functions. Actors do not share any state, they have isolation boundaries and can be executed in the same runtime instance. It is only required to provision the actor execution code and not the execution environment, such as in container-based functions. Actors use messages to communicate. These messages in μActor actors are carried through a distributed pub/sub bus.
μActor uses the following systems:

- Execution System: The actors are transient. For this reason, the code needs to be loaded and unloaded in a process virtual machine. μActor loads the desired code and runs it using Lua. This type of virtual machine provides isolation and

[3]https://krustlet.dev/

reduces overhead. The platform passes control to the actor once the actor gets a message. When the actor is done processing the message, it passes back control to the platform.

- Network System: Pub/Sub bus is used for communication. Actors publish messages in key-value format. Actors subscribe to the interested message contents. There are two types of buses. a local bus that facilitates communication between co-located actors, and a global bus that is responsible for communication between actors at different nodes.

Nevertheless, the introduced platform is not interoperable with the existing state-of-the-art platforms.

6.3.2 Ray

Reinforcement learning (RL) [SB18] is when a learning agent interacts with a dynamic environment in order to achieve a goal in a short period of time with limited feedback. Current technologies, such as Apache Spark [Spa] and Map-Reduce [DG08], do not support simulation or serving. Also, Task-Parallel systems such as CIEL [MSS+11] and DASK [SHC+17] provide short-term support for distributed training and serving.

Ray [MNW+18] is a general-purpose cluster-computing framework. It supports training, simulation, and serving for RL. Tasks help to dynamically and efficiently load balance simulations and process large inputs. It provides task-parallel and actor-based computations. The main purpose of the actors is to support stateful computations, such as model training. Ray guarantees fault tolerance and exactly-once message delivery. Nevertheless, this approach reduces the scalability of a single function, binding the scalability of the two functions together, which can lead to increased resource usage as all the embedded functions must be scaled together.

6.3.3 Durable Functions

Durable Function (DF) focuses on providing stateless, statefull, and parallel computation. It takes task and actor parallelism into account. The following function types are supported:

- Activities: Activities are stateless functions in DF. If the function is not executed successfully, it raises an exception. It does not retry that task, and the parent process receives the exception.

- Entities: Entities are stateful functions, and they are based on the actor model. They support the following functionalities:

  - Orchestration Calls and Signals: Orchestrations can call an entity and wait for it to end the operation, or they can signal an entity and not wait for the end of the operation.
  - Entity-to-Entity Signals: Entities can singal each other, which can enable useful patterns such as stateful flows and data streaming.
  - No Entity-to-Entity Calls: Entities cannot call other entities in order to prevent deadlocks.
  - Scheduled Signals: Singals can be scheduled to be sent at a specific time. This makes it possible to run actions that need to be executed constantly at a specific time. Predioc actions are also supported.

- Orchestrations: It allows for workflows and long-running actions by deconstructing the components into tasks. These tasks can be activities, entities, or sub-orchestrations. The storage of the orchestration is saved automatically so that in faulty scenarios, it can continue from the latest state. It supports sequential composition, parallel composition, and eternal orchestrations.

However, DF is specifically designed for the Azure platform, limiting its usage across other Serverless Platforms.

6.4 Middlewares

6.4.1 SAND

SAND [ACR+18] is a serverless platform that promises lower latency, higher resource efficiency, and better elasticity. It offers the following mechanisms:

- Sandboxing mechanism: There are two types of isolation: isolation between different applications and isolation between functions in the same application. Each application runs inside a container. Each container runs a function in a separate process. Related application functions run on the same host.

- Hierarchical Message Queuing: SAND uses two types of messaging buses: local and global. The local bus is responsible for communication between functions running on the same host. The global bus facilitates communication between functions running on different hosts, and it also serves as a backup for the local bus.

  In order to prevent the same event from being processed multiple times, a backup of the local message is sent to the global with a condition flag. This flag contains information about the host that is processing the message, and the global bus keeps track of the progress. Each host runs a local bus, and the global bus runs across the infrastructure.

Although SAND decreases latency, it reduces the isolation from the functions placed in the shared sandbox, compromising the application scalability as all the functions in the same sandbox need to be scaled together.

6.4.2 Sonic

Sonic [MSM+21] is an intermediate for exchanging data between functions. It provides three different methods for interchanging data:

- VM Storage: The local state of the sending function is stored on the VM. All receiving functions are scheduled to run on the same VM. The use of data locality minimizes the latency. This method has its disadvantages, such as scheduling when the sender function memory cannot fit the receiving functions.
- Direct Passing: The output of the sender function is stored on the VM. Sonic Data Manager receives information about the location and file path of the output. Before executing the receiving function, the metadata manager transfers the output to the receiver VM. This method is in favor of parallelism, but a downside is that the input needs to be transferred over the network.
- Remote Storage: The output of the sender function is uploaded to a remote storage, and all receiving functions download it from the remote storage. This method provides high scalability, but the output needs to be serialized during the upload and download processes.

The user is provided with different APIs that can be chosen depending on different factors, such as input size and network bandwidth. There is no method that can be used in all scenarios. Sonic automatically, using the Viterbi algorithm [For73], selects the data passing method for each scenario to optimize cost and latency. However, remote data exchange relies on third-party services, leading to increased maintenance costs and additional services.

CHAPTER 7

Conclusion

After going through our framework, the approaches that are taken, and the methods that are used, we summarise our paper by reiterating the important aspects and what advancement has been achieved. We talk about our contributions and improvements compared to other frameworks. Also, we revisit the research question and revise the most important points.

7.1 Contributions

In this paper, we introduce a serverless framework, SIWA. SIWA relies on actors to execute functions, putting LifeCycle into use. We introduce LCM, a novel Serverless Lifecycle Model that natively executes serverless functions as serverless actors, allowing actors to be reused without the need to reallocate resources. The actor functionality strongly depends on the current state of the actor. We also present SIM, a novel Serverless Invocation Model that enables actors to influence the behaviour of future messages. Additionally, we introduce the WebAssembly serverless actor platform designed for the Edge-Cloud Continuum. WebAssembly provides strong isolation without any unnecessary overhead.

SIWA leverages Wasm to provide a secure and isolated sandbox while enabling efficient communication among serverless actors via SIWA Middleware. The Middleware leverages the SIM model to enable direct communication between actors, optimising performance and scalability in distributed environments.

Our evaluation 5 demonstrates that SIWA decreases latency and increases throughput, thereby enhancing performance in the Edge-Cloud Continuum. Specifically, SIWA reduces latency by up to 92% and increases throughput by up to 10 times. This has a huge impact on applications running on Edge Devices, since the resources are limited there.

Other frameworks rely on containers and virtual machines to execute and isolate functions, which introduces resource overhead and increases complexity. Using WebAssembly, we reduce resource allocation since no platform-dependent packages or libraries are required.

7.2 Research Challenges

We revise the research challenges that we were mentioned previously and go through their details:

RQ-1: How to enable Serverless actors to be reused in the Edge-Cloud Continuum?

By introducing Actor LifeCycle Model, we are able to maintain the state of the actor without the need to rely on other external systems. This also allows us to reuse the actors by relying on the states, which helps to decrease the cold start and reduce resource consumption.
Also, the actor informs SiWA's distributed middleware about its availability. In order to maintain the integrity of the state, an actor can process one message at a time. The actor can stay in an idle state, reducing resource consumption, but also be ready for other new executions and reuse allocated resources.

RQ-2: How can we enable direct communication between actors while allowing them to influence the behavior of future messages?

Communication between actors is enabled by using the SIWA distributed middleware. Each actor receives a unique ID that makes the actor addressable. The actor or an external source can send a request to the distributed middleware, and the middleware is responsible for delivering the messages to the actor.
Actor LifeCylce helps define the behaviour of future messages. The state is maintained in the actor itself, allowing it to influence the behaviour of the incoming messages. Using the distributed middleware, actors running on different hosts can communicate directly. This allows you to execute a chain of function executions or even nested function calls.

RQ-3: How to provide lightweight isolation while enabling the full potential of Serverless actors in the Edge-Cloud Continuum?

We rely on WebAssembly to provide lightweight and secure isolation. As discussed, it does not only provide strong isolation by relying on SFI, but it also makes it easier for developers to focus on the desired implementation instead of taking care of the prerequisite libraries and packages. Also, different runtimes, such as WasmEdge, focus on optimising applications to run on Edge devices.

7.3 Future Work

Currently, SIWA supports only Wasm sandboxes; we plan to extend to enable traditional container actors and thus enable multiple sandbox isolation. SIWA Serverless Lifecycle

Model enables actor reuse and state management. For stateful functions, SIWA leverages existing state-of-the-art mechanisms to persist the actor state.

In the future, we plan to extend SIWA to add a smart and serialisation-free actor state, where the platform recognises whether the actors specifically require the state, avoiding unnecessary state persistence. Thus optimising resource usage and reducing latency by skipping the actor state loading. Furthermore, we plan to expand SIWA by integrating machine learning algorithms to predict traffic patterns and adjust the creation of actors dynamically, thus improving latency.

List of Figures

List of Tables

List of Listings

Bibliography

[ABI+20] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.

[ACR+18] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[Akk] Akka. Akka Actor Systems. `https://doc.akka.io/docs/akka/current/general/actor-systems.html`.

[AMF+09] Armbrust, Michael, Armando Fox, Armando, Griffith, Rean, Joseph, Denis Anthony, Randy Katz, Randy H, Andy Konwinski, Andrew, Gunho Lee, Gunho, Patterson, David A, Rabkin, Ariel, Stoica, and Matei. Above the clouds: A berkeley view of cloud computing. 01 2009.

[BFM19] Luciano Baresi and Danilo Filgueira Mendonça. Towards a serverless platform for edge computing. In *2019 IEEE International Conference on Fog Computing (ICFC)*, pages 1–10, 2019.

[BGJ+21] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. Durable functions: semantics for stateful serverless. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[BGK+11] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[BPP+19] Philip A. Bernstein, Todd Porter, Rahul Potharaju, Alejandro Z. Tomsic, Shivaram Venkataraman, and Wentao Wu. Serverless event-stream processing over virtual actors. In *Conference on Innovative Data Systems Research*, 2019.

[BPSAP+19] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54, New York, NY, USA, 2019. Association for Computing Machinery.

[CBCH23] Marcin Copik, Roman Böhringer, Alexandru Calotoiu, and Torsten Hoefler. Fmi: Fast and cheap message passing for serverless functions. In *Proceedings of the 37th International Conference on Supercomputing*, ICS '23, page 373–385, New York, NY, USA, 2023. Association for Computing Machinery.

[CCB+22] Marcin Copik, Alexandru Calotoiu, Rodrigo Bruno, Gyorgy Rethy, Roman Böhringer, and Torsten Hoefler. Process-as-a-Service: Elastic and Stateful Serverless with Cloud Processes. Technical report, 01 2022.

[CDTV24] Roberto Casadei, Ferruccio Damiani, Gianluca Torta, and Mirko Viroli. *Actor-Based Designs for Distributed Self-organisation Programming*, pages 37–58. Springer Nature Switzerland, Cham, 2024.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[Ell] Alex Ellis. Serverless Cold Start. `https://www.openfaas.com/blog/what-serverless-coldstart/`.

[ERGC24] Juan José López Escobar, Rebeca P. Díaz Redondo, and Felipe Gil-Castiñeira. Unleashing the power of decentralized serverless iot dataflow architecture for the cloud-to-edge continuum: a performance comparison. *Annals of Telecommunications*, pages 1–14, 2024.

[For73] G.D. Forney. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

[GFD22] Philipp Gackstatter, Pantelis Frangoudis, and Schahram Dustdar. Pushing serverless to the edge with webassembly runtimes. pages 140–149, 05 2022.

[Hal12] Philipp Haller. On the integration of the actor model in mainstream technologies: the scala perspective. In *Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions*, AGERE! 2012, page 1–6, New York, NY, USA, 2012. Association for Computing Machinery.

[Hew10] Carl Hewitt. Actor model for discretionary, adaptive concurrency. *CoRR*, abs/1008.1459, 2010.

[HFC+23] Zhuo Huang, Hao Fan, Chaoyi Cheng, Song Wu, and Hai Jin. Duo: Improving data sharing of stateful serverless applications by efficiently caching multi-read data. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 875–885, 2023.

[HFG+18] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[HKO21] Raphael Hetzel, Teemu Kärkkäinen, and Jörg Ott. actor: Stateful serverless at the edge. In *Proceedings of the 1st Workshop on Serverless Mobile Networking for 6G Communications*, MobileServerless'21, page 1–6, New York, NY, USA, 2021. Association for Computing Machinery.

[HRS+17] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery.

[JSS+19] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019.

[JW21a] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 691–707, New York, NY, USA, 2021. Association for Computing Machinery.

[JW21b] Zhipeng Jia and Emmett Witchel. 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, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery.

[KHA+23] Samuel Kounev, Nikolas Herbst, Cristina L. Abad, Alexandru Iosup, Ian Foster, Prashant Shenoy, Omer Rana, and Andrew A. Chien. Serverless computing: What it is, and what it is not? *Commun. ACM*, 66(9):80–92, aug 2023.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[KKL+07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.

[KNGB21] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating Function-as-a-Service workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 805–820. USENIX Association, July 2021.

[KWS+18] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[LGC+22] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *ACM Comput. Surv.*, 54(10s), sep 2022.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, abs/1801.01207:1–19, 2018.

[LTHY21] Ju Long, Hung-Ying Tai, Shen-Ta Hsieh, and Michael Juntao Yuan. A lightweight design for serverless function as a service. *IEEE Software*, 38(1):75–80, 2021.

[MBN+21] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. Ofc: An opportunistic caching system for faas platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 228–244, New York, NY, USA, 2021. Association for Computing Machinery.

[Mic] Microsoft. Windows Containers Isolation Modes. `https://learn.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/hyperv-container`.

[MMR+13] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon

Crowcroft. Unikernels: library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 461–472, New York, NY, USA, 2013. Association for Computing Machinery.

[MN23] Cynthia Marcelino and Stefan Nastic. 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 2023)*, 2023.

[MNW+18] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[MPFS22] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Webassembly as a common layer for the cloud-edge continuum. In *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*, FRAME '22, page 3–8, New York, NY, USA, 2022. Association for Computing Machinery.

[MPMB20] Seán Murphy, Leonardas Persaud, William Martini, and Bill Bosshard. On the use of web assembly in a serverless context. In Maria Paasivaara and Philippe Kruchten, editors, *Agile Processes in Software Engineering and Extreme Programming – Workshops*, pages 141–145, Cham, 2020. Springer International Publishing.

[MSM+21] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301. USENIX Association, July 2021.

[MSS+11] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. {CIEL}: A universal execution engine for distributed {Data-Flow} computing. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[MTA+23] Alex Merenstein, Vasily Tarasov, Ali Anwar, Scott Guthridge, and Erez Zadok. F3: Serving files efficiently in serverless computing. In *Proceedings of the 16th ACM International Conference on Systems and Storage*, SYSTOR '23, page 8–21, New York, NY, USA, 2023. Association for Computing Machinery.

[MTZ23] Fatma M. Talaat and Hanaa Zaineldin. An improved fire detection approach based on yolo-v8 for smart cities. *Neural Computing and Applications*, 07 2023.

[Mun19] Chris Munns. Tracking the state of aws lambda functions, 2019.

[NRF$^+$22] Stefan Nastic, Philipp Raith, Alireza Furutanpey, Thomas Pusztai, and Schahram Dustdar. A serverless computing fabric for edge & cloud. In *2022 IEEE 4th International Conference on Cognitive Machine Intelligence (CogMI)*, pages 1–12, 2022.

[POPL18] Daniel Barcelona Pons, Alvaro Ruiz Ollobarren, David Arroyo Pinto, and Pedro Garcia Lopez. Studying the feasibility of serverless actors. In *Proceedings of the European Symposium on Serverless Computing and Applications, ESSCA@UCC 2018, Zurich, Switzerland, December 21, 2018*, volume 2330, pages 25–29. CEUR-WS.org, 2018.

[Ray23] Partha Pratim Ray. An overview of webassembly for iot: Background, tools, state-of-the-art, challenges, and future directions. *Future Internet*, 15(8), 2023.

[RCG$^+$21] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faast: A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 122–137, New York, NY, USA, 2021. Association for Computing Machinery.

[RND23] Philipp Raith, Stefan Nastic, and Schahram Dustdar. Serverless edge computing—where we are and what lies ahead. *IEEE Internet Computing*, 27(3):50–64, 2023.

[S3] Amazon S3. Amazon S3. `https://aws.amazon.com/de/s3/`.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SCC$^+$23] C. Sicari, A. Catalfamo, L. Carnevale, A. Galletta, D. Balouek-Thomert, M. Parashar, and M. Villari. Tema: Event driven serverless workflows platform for natural disaster management. In *2023 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, Los Alamitos, CA, USA, jul 2023. IEEE Computer Society.

[SCH24] Jonas Spenger, Paris Carbone, and Philipp Haller. *A Survey of Actor-Like Programming Models for Serverless Computing*, pages 123–146. Springer Nature Switzerland, Cham, 2024.

[SHC$^+$17] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017.

[SP20] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, pages 419–433, USA, 2020. USENIX Association.

[Spa] Apache Spark. Apache Spark. `https://spark.apache.org/`.

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

[Str98] Volker Strumpen. Portable and fault-tolerant software systems. *IEEE Micro*, 18(05):22–32, 1998.

[SWL$^+$20] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: stateful functions-as-a-service. *Proceedings of the VLDB Endowment*, 13(12):2438–2452, August 2020.

[SWN$^+$22] Qiang Su, Chuanwen Wang, Zhixiong Niu, Ran Shu, Peng Cheng, Yongqiang Xiong, Dongsu Han, Chun Xue, and Hong Xu. Pipedevice: a hardware-software co-design approach to intra-host container communication. pages 28–30, 10 2022.

[WCJL23] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. Rise of the planet of serverless computing: A systematic review. *ACM Trans. Softw. Eng. Methodol.*, 32(5), jul 2023.

[WFLH18] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. Anna: A kvs for any scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 401–412, 2018.

[WZM$^+$20] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. InfiniCache: Exploiting ephemeral serverless functions to build a Cost-Effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 267–281, Santa Clara, CA, February 2020. USENIX Association.

[XZG$^+$19] Zhengjun Xu, Haitao Zhang, Xin Geng, Qiong Wu, and Huadong Ma. Adaptive function launching acceleration in serverless computing platforms. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 9–16, 2019.

[YZCH+19] Ethan G Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The true cost of containing: A {gVisor} case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.