



GoldFish: Serverless Actors with Short-Term Memory State for the Edge-Cloud Continuum

Cynthia Marcelino

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

Jack Shahhoud

TU Wien
Vienna, Austria
e1631081@student.tuwien.ac.at

Stefan Nastic

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

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. Serverless stateless nature requires functions to leverage external services, such as object storage and KVS, to exchange data. Serverless actors have emerged as a solution to these issues. However, the state-of-the-art serverless lifecycle and event-trigger invocation force actors to leverage remote services to manage their state and exchange data which impacts the performance, incurs additional cost and dependency on third-part services. To address these issues, in this paper, we introduce a novel serverless lifecycle model that allows short-term stateful actors, enabling actors to maintain their state between executions. Additionally, we propose a novel serverless Invocation Model that enables serverless actors to influence the processing of future messages. We present GoldFish, a lightweight WebAssembly short-term stateful serverless actor platform which provides a novel serverless actor lifecycle and invocation model. GoldFish leverages WebAssembly to provide the actors with lightweight sandbox isolation, making them suitable for the Edge-Cloud Continuum, where computational resources are limited. Experimental results show that GoldFish optimizes the data exchange latency by up to 92% and increases the throughput by up to 10x compared to OpenFaaS and Spin.

CCS Concepts

• **Software and its engineering** → **Cloud computing**; **Message passing**; **Middleware**.

Keywords

Serverless computing, WebAssembly, Wasm, FaaS, Actor model, Serverless actor, Data-intensive workflows, Edge-Cloud

ACM Reference Format:

Cynthia Marcelino, Jack Shahhoud, and Stefan Nastic. 2024. GoldFish: Serverless Actors with Short-Term Memory State for the Edge-Cloud Continuum. In *14th International Conference on the Internet of Things (IoT 2024)*, November 19–22, 2024, Oulu, Finland. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3703790.3703797>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
IoT 2024, November 19–22, 2024, Oulu, Finland
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1285-2/24/11
<https://doi.org/10.1145/3703790.3703797>

1 Introduction

Serverless Computing is a paradigm that offers automated infrastructure management, scale to zero, and elastic scaling. Typically, a Serverless application consists of a series of interconnected functions, also known as a Serverless Workflow, that exchange ephemeral data, which can be discarded after function processing. Due to the Serverless stateless design, functions in a workflow leverage external services such as object storage, message brokers, and Key-Value stores (KVS) to exchange ephemeral data and manage their state. Although external services provide benefits such as computing and IO separation, they add significant latency overhead [18, 21, 27, 30]. Moreover, functions are not directly accessible; they are accessible via platform ingresses such as API Gateway and Load Balancer [5, 15, 40], thus making direct communication more challenging. Serverless actors [1, 4, 6, 8, 16, 25, 33] have emerged addressing these issues, thus enabling direct communication, state persistence, and concurrency management, which is crucial for Serverless functions.

Actors [2, 14] are isolated entities that can ① create other actors, ② directly communicate with other actors and ③ influence the processing or state for the next received message [6, 8, 14, 29, 35]. Serverless functions are ① stateless, ② non-addressable, and ③ event-triggered [5, 15, 27, 40].

Existing Serverless actor approaches [1, 6, 16, 25, 33] leverage the state-of-the-art Serverless design characteristics such as lifecycle [26] and event-trigger invocation [18, 20, 40] to enable stateful and addressable actors. However, in the current Serverless function lifecycle [26], Serverless functions are stateless. Therefore, existing actor-like Serverless approaches leverage remote services, incurring network overhead and costs with additional services.

Existing approaches that enable persistent stateful functions include: (a) *Programming Models* [6–8, 32] 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 as they rely on external services. (b) *Sidcars* [1, 17] 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, sidcars 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. (c) *Custom Sandboxes* [9, 33] 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 Knative, OpenFaas, and OpenWhisk. Although these approaches offer state preservation, allowing Serverless functions to execute as actors, they still rely on external services for state management, causing up to 95% of the function latency [3, 19]. To fully utilize actor potential, the Serverless lifecycle must ensure actors can process multiple requests while preserving their state for a short period. Consequently, actors maintain states between executions, avoiding unnecessary state propagation of ephemeral data for interconnected events.

Existing Serverless actors enable direct communication and persistent statefulness by leveraging the existing Serverless lifecycle and event-triggered invocation. As a result, a series of interconnected events lead to multiple actor instances that still rely on external services to exchange ephemeral data and store their state, impacting the performance significantly. To address these issues, in this paper, we propose novel Serverless lifecycle and invocation models that enable actors to process multiple interconnected requests and facilitate serverless actors to influence the processing of future messages. Finally, we present GoldFish, a lightweight actor-based Serverless platform that executes serverless functions as actors. The main contributions of this paper include:

- *LCM: A novel Serverless Lifecycle Model* that natively executes serverless functions as actors. It allows serverless actors to preserve a short-term state between the executions, thereby reducing multiple actor instantiating for multiple requests.
- *SIM: A novel Serverless Invocation Model* that allows actors to influence the processing of future messages, enabling them to handle multiple messages. Busy actors can reject future messages or queue them for processing when available. Hence, SIM facilitates the processing of a set of connected events, such as in Serverless Workflows, and thus, it maintains the context and continuity of event processing.
- *GoldFish: A WebAssembly Serverless Actor Platform* that leverages Wasm to provide a lightweight isolation. GoldFish architecture leverages the LCM model to enable serverless functions to execute as actors. Furthermore, GoldFish introduces its dedicated message middleware that enables direct communication and leverages SIM to enable actors to influence the processing of future messages, thus processing multiple messages.

This paper has eight sections. Section 2 presents the illustrative scenario and research questions. Section 3 describes GoldFish Serverless lifecycle and invocation model as well as architecture overview. Section 4 describes the lifecycle management and the event-triggered message invocation introduced by GoldFish and their usage. Section 5 shows the prototype implementation details. Section 6 discusses the experiments and evaluation, Section 7 presents related work. Section 8 concludes with a final discussion and future work.

2 Motivation

2.1 Illustrative Scenario

To better motivate our research, we present a use case for real-time field monitoring and disease detection in smart agriculture.

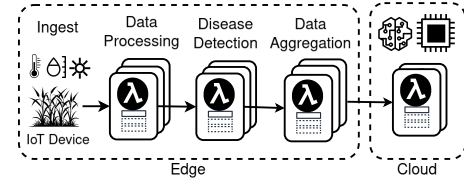


Figure 1: Simplified Serverless Workflow for Disease Control for Smart Agriculture

To achieve this, IoT devices are strategically positioned throughout the fields to detect crop properties such as soil moisture, temperature, humidity, and sunlight. A Serverless workflow is employed to identify and respond to these agricultural needs.

Our workflow utilizes four Serverless functions, partially executed on the Edge close to the data source to reduce communication latency and partially executed on the Cloud. Edge tasks are responsible for processing large real-time data streams, sensor data, and simple disease detection. On the other hand, tasks that require more powerful computing resources, such as model training and inference, are carried out in the Cloud. Our motivating scenario is inspired by a Serverless Workflow for real-time environmental monitoring [11].

In Fig. 1, in *Ingest* stage, real-time data captured by IoT devices are transmitted to edge nodes via a streaming framework, where serverless functions responsible for *Data Processing* are activated to execute tasks such as filtering, labeling and join the sensor data close to the source, thus reducing latency. Then *Disease Detection* function processes part of the data, identifying specific patterns such as temperature and soil moisture. Then, *Disease Detection* functions send data to *Data Aggregation* functions, which combines current and historical data to enhance accuracy and reliability of the results. Finally, the processed data is transmitted to the cloud, where more resource-intensive tasks are performed, such as AI model inference to enhance disease analysis.

GoldFish decreases this workflow latency by enabling actors to process connected message events within a single actor. By softening some Serverless properties, such as statelessness, actors can keep a short-term state between executions, avoiding the need for remote services to exchange data. Additionally, actors can influence the processing of the next message, choosing to keep it in the queue for processing or reject it completely, allowing another actor to process the message. Thus, GoldFish enhances performance while maintaining the serverless nature of the function, as it still scales down to zero when not in use. On the other hand, it avoids the network overhead associated with external remote services for state persistence and data exchange. This tradeoff enhances performance without compromising the fundamental benefits of Serverless Computing.

2.2 Research Challenges

We identify the following research challenges to enable Serverless actors to maximize their performance in the Edge-Cloud Continuum.

RC-1: How to enable short-term stateful Serverless actors in the Edge-Cloud Continuum while preserving Serverless characteristics such as scale-to-zero?

The current Serverless function lifecycle supports either succeeded or failed states, which leads to platforms creating multiple function instances for handling multiple function executions. Current approaches for Serverless actors preserve their state in remote storage and load the previous state in 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 [8, 16, 26]. Communication with external services causes the most function latency and additional costs, significantly affecting the performance and cost-efficiency of Serverless workflows. Relying on external services for state management adds latency, complexity, potential points of failure and costs due to frequent data retrieval [3, 19]. Short-term stateful actors allow for state preservation within the actors themselves, eliminating the need for external services for state persistence. This minimizes the number of created instances, decreases latency and costs, and preserves resources at the edge. Hence, actors can scale to zero in the absence of invocations while still providing the advantages of stateful functions.

RC-2: How can we enable direct communication between actors while allowing them to influence the processing 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 ① forward to the actor for processing, ② enable actors to keep the message in the middleware until the actor becomes available again, or ③ forward to another actor in case of rejection by the existing actor. By enabling actors to influence the processing of the message, users can decide to process connected message events in same actors, thus decreasing latency and network traffic overhead, crucial for enhancing performance in sensitive edge environments [3, 22, 27].

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

Isolation is critical to ensure that failures in one actor do not impact others. WebAssembly (Wasm) provides a secure sandboxed environment that reduces the overhead associated with traditional container-based isolation methods. Wasm lightweight isolation allows serverless actors to execute with reduced cold start, latency and resource consumption, which is crucial for the Edge-Cloud Continuum. Furthermore, actors can profit from the reduced cold starts Wasm, decreasing the actors startup time [13, 22, 24].

3 GoldFish Serverless Models and Architecture Overview

3.1 GoldFish Serverless Lifecycle Model

Goldfish Serverless Lifecycle Model (LCM) provides an enhanced Serverless lifecycle specifically tailored for serverless actors to preserve their state between multiple executions while they are still alive. LCM still maintains Serverless characteristics such as elastic scaling and scale-to-zero while enabling actors short-term state

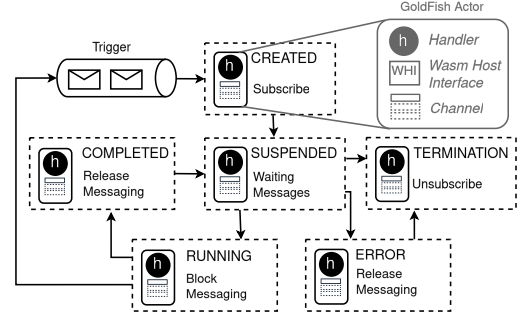


Figure 2: GoldFish Serverless Lifecycle Model

memory. Thus, LCM optimizes 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 utilized and consequently minimizing the overhead associated with creating new actors.

3.1.1 GoldFish Actor. Fig. 2 shows GoldFish Actor and LCM Serverless Lifecycle. GoldFish actor is one entity composed of *Channel*, *Wasm Host Interface*, and *Handler*.

Channel. It is identifiable by a unique ID and serves as a dedicated communication channel for the actor. 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.

Wasm Host Interface (WHI). It is a sidecar process that creates the Wasm VM, 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. Upon receiving a message, WHI sends a signal to the Middleware to temporarily block any new incoming messages, ensuring actors process only a single message at a time.

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*.

3.1.2 GoldFish Serverless Lifecycle Phases. Fig. 2 shows LCM actor phases from the initialization to the termination. LCM phases are designed to ensure actor isolation and enable the (short-term) state management, key properties of actor model [8, 29, 35]. Each phase is responsible for specific tasks described below.

CREATED. This initial phase prepares the actor for operation and reserves the resources necessary. In addition, the actor receives a unique ID, which is later used for actor communication.

SUSPENDED. In this phase, the actor is not currently processing any tasks but is ready and waiting for new input or to terminate the actor if it remains in this phase for a long time. The period the actor remain in suspended phase is determine by the user. **SUSPENDED** phase enables actors to keep the actor active but not running, it is waiting for incoming events to become active. This phase is essential

for managing the efficient allocation of resources, enabling GoldFish to quickly respond to new messages without the overhead of the initialization phase.

ERROR. In this phase, the actor has failed either during startup or execution. To enable message reprocessing, the actor releases the message and moves to *TERMINATION* as a self-destroy mechanism.

RUNNING. During this phase, GoldFish wakes up the actor from the *SUSPENDED* phase and forwards the message to the actor. In this phase, it is where the actual data processing or task execution takes place. Additionally, in this phase, actors can create other actors by sending an addressed message to GoldFish Middleware.

COMPLETED. Once the messaging process is completed, the actor sends the results to the GoldFish middleware and signals its availability for further tasks. Then the actor can transition back to the *SUSPENDED* phase.

TERMINATION. The final phase of the lifecycle, where the actor stops receiving new messages, deregisters itself. In this phase, GoldFish releases resources and updates the actor state to reflect that the actor is no longer active.

3.2 GoldFish Serverless Invocation Model

The GoldFish Serverless Invocation Model (SIM) design ensures actors only handle one message at a time, which means concurrent requests is only possible with multiple actors, thereby avoiding concurrency issues and maintaining state integrity during the message delivery. Specifically, it enables processing multiple messages within a single actor message rather than handling each in isolation. Hence, SIM supports a set of connected events, facilitating more efficient workflow execution. Moreover, SIM enables serverless actors to influence future messages by keeping the message waiting to be executed, thus avoiding the use of remote services to store state and exchange data. As a result, it optimizes resource usage and reduces latency, which is crucial for improving the performance of functions in the Edge-Cloud Continuum.

Fig. 3 shows how SIM introduces a new way of triggering serverless actors in response to events such as incoming messages. The GoldFish SIM model ensures that new actors are created only when necessary while existing actors are reused by introducing a invocation Middleware with three queues: *waiting*, *ready* and *done*. GoldFish SIM model enables GoldFish Buffer to identify the availability and state of actors via the actor lifecycle phase. If the actor is *SUSPENDED*, it transitions to the *RUNNING* phase to handle the

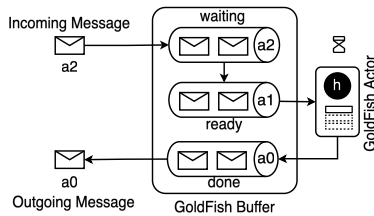


Figure 3: GoldFish Serverless Invocation Model

message. If the actor is busy, the Buffer keeps the message or forwards it to another available actor, ensuring seamless processing without message loss. Thus, SIM invocation model enables Serverless actors to process connected messages such as in a Serverless Workflow. To avoid a long waiting time, the Buffer has a time and message size limit defined by the user; once the time has reached, a new actor instance is created instead of reusing an existing actor.

3.3 GoldFish Architecture Overview

GoldFish leverages actor model properties such as addressability, isolation, and state to enhance serverless function execution by transforming them into Serverless actors [14, 29, 35]. Each serverless actor in GoldFish is uniquely identifiable, allowing for direct, addressable communication, thereby facilitating efficient data and message exchanges across the actors in the Edge-Cloud Continuum.

The GoldFish architecture, shown in Fig. 4, leverages Wasm to provide an isolated and secure sandbox for each actor. Moreover, GoldFish’s LCM manages the lifecycle of serverless actors from initialization to termination. GoldFish LCM enables Serverless actors to retain and efficiently manage their state, thus facilitating complex functions that require persistent state across sessions.

3.3.1 GoldFish Components. GoldFish is composed of main components: *GoldFish Middleware*, *Registry*, *GoldFish Buffer* and *Actor Dispatcher*.

GoldFish Middleware. It accepts messages and ensures the messages are routed to the buffer in the correct node. When a GoldFish Buffer initiates, it registers itself in the GoldFish Middleware in the control plane. This registration enables the middleware proxy to route messages accurately to the designated actor dispatcher node.

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 dispatcher.

GoldFish Buffer. It is a queue for the busy actors, keeping waiting messages, thus allowing actors to influence the sequence of messages. When GoldFish Middleware receives a message, it passes it to the GoldFish Buffer if there is enough processing capacity. The Buffer then checks if the actors can handle new messages and sends them to the Actor Dispatcher. If the actors are unable to process

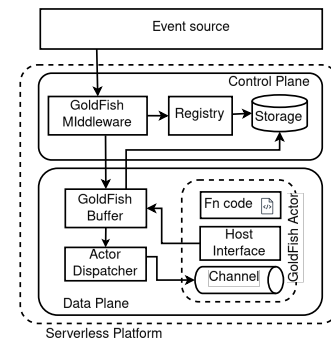


Figure 4: GoldFish Architecture Overview

new messages, the message is rejected. When a message is rejected, it is either kept waiting in the buffer or sent to another Actor Dispatcher until it is accepted. The message processing is defined by the actor, who can choose to receive the next message or reject it.

Actor Dispatcher. It manages the actors and their phases. The Actor Dispatcher receives the messaging events, identifies whether the actor exists by its unique ID, and forwards the message. The Actor Dispatcher updates Actor references in storage that are available via the control plane.

4 GoldFish Mechanisms

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

4.1 GoldFish LCM Lifecycle Phases Management

To execute Serverless functions as actors, GoldFish leverages the LCM to create and reuse actors. Fig. 5 shows each phase and which services are necessary to enable the LCM.

In ①, in Fig. 5, when the actor is *CREATED*, it subscribes to a specified channel with its unique ID. Then, GoldFish Middleware stores actor references for future usage. *CREATED* is the initial phase where the platform executes tasks to prepare for the actor run, such as physical resource reservation and deployments. In the next phase in ②, the actor enters the *SUSPENDED* phase, waiting for incoming messages for a period of time defined by the user. This is necessary to avoid actors to run constantly. In ③, a message is received, and the middleware retrieves information from the storage to identify the actor and forwards the message to the actor via the actor channel. Once the actor receives the message, it sends an event to the GoldFish Middleware to block new incoming messages. GoldFish middleware then updates the actor reference to the storage, finalizing this actor is busy and cannot receive any new message. In ④, the actor completes the message processing

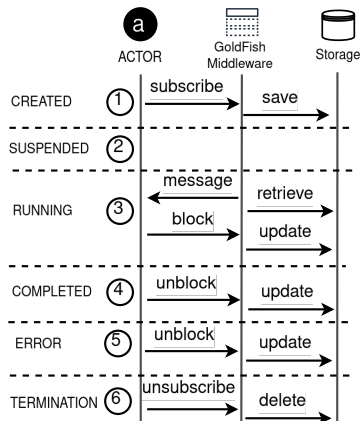


Figure 5: GoldFish Serverless Lifecycle Management

and sends a signal to GoldFish to unblock the actor. GoldFish Middleware updates the actor reference and removes the block. After this phase, the actor returns to phase ② to receive new messages. After a period defined by the user, the actor moves to the final phase *TERMINATION* in ⑥. Phase ⑤ represents an error state in the actor, the actor has either failed to startup or during execution. After entering the *ERROR* phase, the actor unblocks the message in GoldFish Middleware which updates the actor reference in the storage. In ⑥, the *TERMINATION* phase, the actor unsubscribes to the channel. GoldFish Middleware deletes the specific channel and removes the actor reference from the storage. In this phase, the platform also releases reserved resources and removes any actor reference.

4.2 GoldFish SIM Serverless Message Invocation

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

Fig. 6 shows how GoldFish Middleware distributes the message from the event source to the user function code. In ①, an event arrives at the Middleware with the unique address of the actor. In ② the Middleware fetches from the storage existing actors information such as address and lifecycle phase to find out if any existing node contains such an actor already. In ③, the middleware forwards the message to either an existing actor that is suspended, an existing actor that signaled that they want to process it as the next message, or to the first free buffer that can potentially create a new actor to process such message. In ④ the buffer queries the actor state to know whether it is immediately available if the actor wants to process the message next or reject it. In ⑤ the buffer

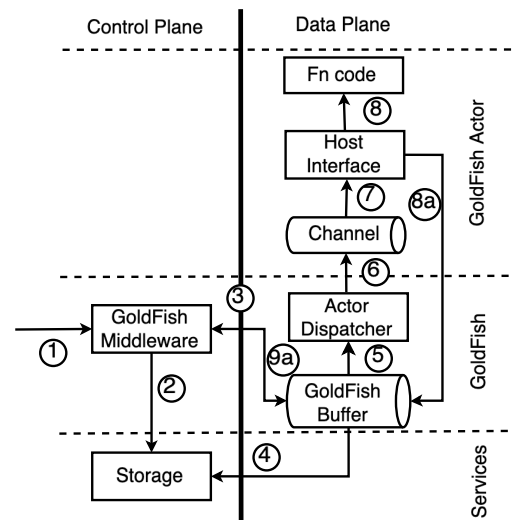


Figure 6: GoldFish Distributed Messaging Middleware Flow

forwards the message to the Actor Dispatcher or keeps the message in memory for future processing. To avoid multiple storage queries, the buffer also forwards the actor information, which is necessary for the decision-making in the *Actor Dispatcher*. In ⑥, the Actor Dispatcher creates an actor with its channel or forwards the message to an existing actor channel. This decision is made during the actor lifecycle phase. In ⑦ the host interface receives the message from the channel and creates the Wasm VM. In ⑧, 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. In ⑨, the buffer forwards the message to the middleware, which starts the process for the new message receiving from in ①. In ⑧ the *Host Interface* starts the Wasm VM with the user function code.

5 Prototype Implementation

GoldFish is published as an open-source framework part of the Polaris SLO CLOUD. Polaris itself is part of the Linux Foundation Centaurus project. GoldFish source code is available on GitHub¹.

The actor in GoldFish comprises a message channel, a Wasm host interface, and a Wasm binary containing the user function code. We utilize WasmEdge[38] as the runtime, along with WasmEdge libraries, to create the Wasm VM. To ensure scalability, we use Docker[10] to run GoldFish actors, and the Rust wasmedge-sdk[39] facilitates interaction with WasmEdge. Events are sent to the middleware using WasmEdge Host Functions, which enable WebAssembly to call native Rust functions by passing them as imports to Wasm modules. The middleware is responsible for receiving and forwarding events to dispatchers, registering itself in the middleware registry upon startup. It communicates with Redis[31] to verify actor information such as phase and address and is implemented using gRPC interfaces. The middleware registry collects references to active middleware via gRPC and stores these references in Redis. Actor dispatchers respond to events received by the middleware, creating an OCI Bundle with Docker that encapsulates the actor, ensuring interoperability with state-of-the-art platforms. Implemented in Rust, the dispatchers use Rust libraries to create gRPC interfaces that are available to the bus.

6 Evaluation

We design our experiments to evaluate our GoldFish based on our illustrative scenario, shown in Section 2.1, and on the most common invocation patterns of Serverless Computing: Sequential Executions and Fan-out execution, as discussed in [18]. The goal of the evaluation is to measure the performance of the contributions LCM, SIM and Goldfish platform presented in Section 1.

Baselines & Experimental Workflows. We compare GoldFish to OpenFaaS[28] and Spin[37]. We have chosen OpenFaaS to compare GoldFish with a standard container Serverless Platform that has wide support in the open-source community. As GoldFish, Spin leverages WebAssembly, and therefore, it is important for GoldFish to compare with a framework that leverages similar technologies. We execute *Chained Functions* and *Serverless Workflow*, based on our illustrative scenario in Section 2.1, for all three baselines (OpenFaaS,

Spin and GoldFish) with three functions to simulate real-world data-intensive Serverless use cases. In Chained Functions, we show the use case when a serverless functionA calls a serverless functionB. In Serverless Workflow, the next function is only executed once the previous function has finished.

Metrics. *Latency* 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. Moreover, *Throughput* measures the number of executions a framework can process in a specific timeframe. We measure the performance of GoldFish under high load. The goal of Throughput experiments is to identify how many requests can the function process at a time and if there are bottlenecks in the proposed framework once the function load increases.

6.1 Experiment Setup

To evaluate GoldFish, we execute the designed experiments on a Ubuntu 22.04 LTS machine CPU ARM64 (AARCH64) with 8 GB of RAM, 4 cores, and 39 GB of storage. The experimental functions and workflow are written in Rust for all the baselines. The baseline functions used for the evaluation expose REST API endpoints for receiving and processing requests from external sources. For the HTTP requests, we use Rust libraries 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.

6.2 Experiment: Sequential Executions

In this experiment, we perform sequential request executions for our two experimental workflows: Chained Functions and Serverless Workflow.

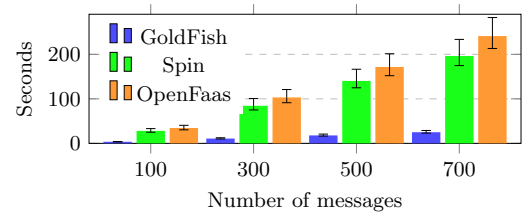


Figure 7: Message Exchange Latency

In Fig. 7, we show the latency of multiple sequential message requests processed by GoldFish Middleware, and the baseline Spin and OpenFaaS. In the x axis, we display the number of messages and, in y, the latency to process these messages. GoldFish Middleware shows latencies from approximately 3.56 to 25.04 seconds, while Spin displays an increase from about 28.06 to 196.25 seconds, and OpenFaaS shows latency growing from roughly 34.58 to 241.00 seconds. The results show that GoldFish reduces latency up to 89% when compared to the baseline. Fig. 8a shows the input data size on the x axis and the latency in seconds on the y axis. GoldFish 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

¹<https://github.com/polaris-slo-cloud/goldfish>

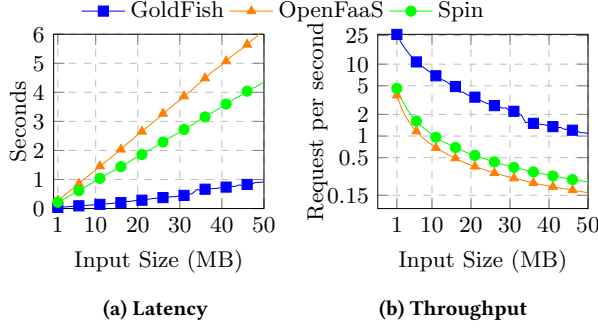


Figure 8: Sequential Execution: Chained Functions

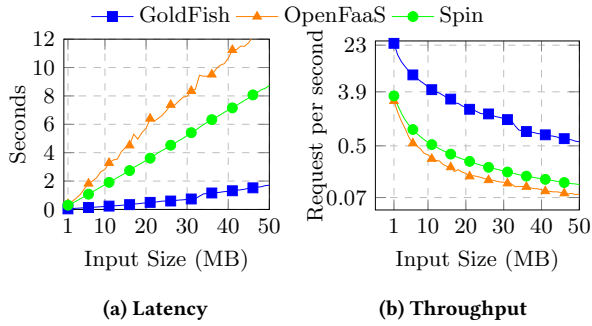


Figure 9: Sequential Execution: Serverless Workflow

analysis reveals that GoldFish decreases the latency by up to 85% and 79% compared to OpenFaaS and Spin, respectively. These latency experiments show a significant latency reduction of GoldFish, with all three systems demonstrating a generally linear increase in response times, indicative of stable performance across the increasing load. Fig. 8b shows the throughput of GoldFish, 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 , GoldFish'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, GoldFish maintains a throughput up to 6.8 times higher than OpenFaaS and up to 4.7 times higher than Spin.

Fig. 9a presents the input data size in megabytes on the x axis and the response latency on the y axis. As input size increases, GoldFish 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 GoldFish reduces latency by up to 86% compared to OpenFaaS and 80% relative to Spin.

Fig. 9b 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. GoldFish 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. GoldFish presents up to 7.4 times higher throughput than OpenFaaS and up to 5.4 times more than Spin.

6.3 Experiment: Fan-out Parallel Executions

In these experiments, we measure GoldFish scalability with fan-out parallel request executions for Chained Functions and Serverless Workflows.

Fig. 10a 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. Fig. 10a that GoldFish 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 Spin showing a latency of around 50 milliseconds. GoldFish shows up to an 87% reduction in latency compared to OpenFaaS and Spin.

In Fig. 10b, GoldFish 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 Fig. 10a. OpenFaaS and Spin also display consistent throughput, with OpenFaaS and Spin presenting around 50 requests per second even when the function load increases in axis x . Overall, GoldFish has up to 9x higher throughput when compared to OpenFaaS and Spin.

Fig. 11a 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.

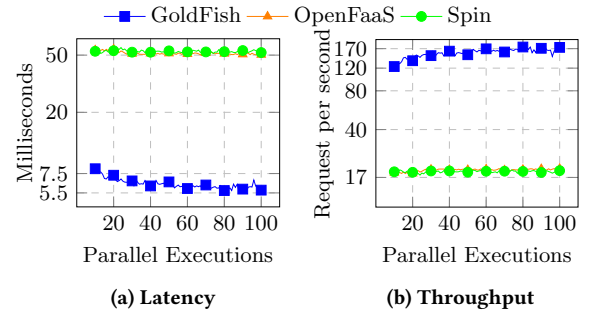


Figure 10: Parallel Execution: Chained Functions

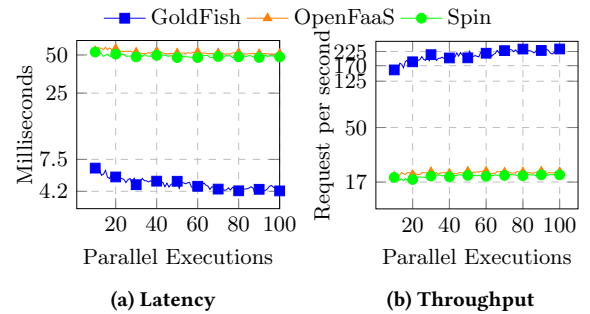


Figure 11: Parallel Execution: Serverless Workflow

GoldFish 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 Fig. 10a, around 50ms. Compared to the baselines, GoldFish's latency is lower, showing an improvement of approximately 92% for serverless workflows.

In Fig. 11b, GoldFish 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 GoldFish. These results show that GoldFish has up to 10x higher throughput compared to the baselines, showing stability for high-load serverless workflows while maintaining high throughput and low latency.

7 Related Work

Serverless Actor Model. μ Actor [16] introduces a lightweight stateful serverless platform able to execute actors not only on the cloud but also at the edge with limited resources such as micro-controllers. μ Actor enables actors to send and receive messages from another actor via publish/subscribe mechanisms. Furthermore, actors may have access to additional devices such as sensors, actuators, databases, and DSP chips. Nevertheless, the introduced platform is not interoperable with the existing state-of-the-art platforms such as Knative, OpenFaas, and OpenWhisk, while Goldfish implements the actor a standard container which can be used by most of open source and commercial Serverless platforms. Microsoft Azure's Durable Functions (DF) [6] introduces programming model abstractions to enable function state handling while ensuring reliable task progression. DF combines task and actor parallelism to create a fault-free function model. However, DF is specifically designed for the Azure platform, limiting its usage across other Serverless Platforms such as AWS Lambda, OpenFaaS, and OpenWhisk. Akka [1] introduces a side-car container that intercepts the incoming and outgoing traffic to manage the function state via external storage and proxies the traffic to the user function container. Nevertheless, Akka introduces an additional system that runs on an additional container, leading to potential increased resource usage, thus limiting its usage in the Edge-Cloud Continuum, where computational resources are limited. Ray [25] introduces a fully managed serverless platform tailored for AI that natively integrates the actor properties in the serverless functions, ensuring fault recovery and at-least-once message delivery mechanism. Ray preserves the state between the serverless AI workflow, wrapping multiple functions into one actor, such as extract and process frames, thus enabling low latency as functions are embedded in one actor. Although these approaches enable Serverless actors, they still rely on external services to persist the actor state even for ephemeral and intermediate data, thus increasing latency, costs, and digital waste. Goldfish keeps a short-term memory state in the actor so that actors can leverage the state to exchange ephemeral data exchange.

Stateful Serverless. Faasm [33] introduces a stateful Serverless via faaslet and a two-tier state architecture for state and message exchange via faabric [34]. Faaslet provides lightweight isolation for each function, while the two-tier state architecture enables local and global function state storage based on the function location.

Nevertheless, Faasm introduces customized isolation mechanisms incompatible with the OCI specs [12] of the current state-of-the-art serverless platforms. Cloudburst [36] proposes a stateful Serverless platform that leverages Anna [41] Key-Value Store (KVS) for data exchange. Cloudburst replicates part of the cache locally for each function, allowing low-latency access, while remote data is accessed via Anna KVS. Although Cloudburst offers low latency and a highly scalable serverless platform, it might introduce duplicate cached data, leading to network overhead and duplicate serialization, a challenge for the limited resources of the Edge-Cloud Continuum. Although the presented approaches enable stateful serverless, they focus on a persistent state, leading to network overhead, dependency on external systems, and additional costs. As Goldfish provides short-term state and multiple request executions, actors can keep their state for a short period between executions, avoiding the need for external service and thus improving performance significantly.

8 Conclusion & Future Work

In this paper, we presented Goldfish, a short-term stateful Serverless for the Edge-Cloud Continuum that provides a novel Serverless Lifecycle Model (LCM) that allows actors keep a short-term state. Goldfish provides also SIM, a novel Serverless Invocation Model that enables actors to influence the processing of future messages, thus enabling one actor to process multiple requests. GoldFish leverages Wasm to provide a secure and isolated sandbox while enabling efficient ephemeral-data communication among serverless actors, thus optimizing performance and scalability in distributed environments.

Our evaluation demonstrates that GoldFish decreases latency and increases throughput, thereby enhancing performance in the Edge-Cloud Continuum. Specifically, GoldFish reduces latency by up to 92% and increases throughput by up to 10 times. GoldFish is specifically designed to address the requirements of the Edge-Cloud Continuum. Goldfish provides a lightweight Wasm sandbox, which fits the limited resource environment of the Edge Cloud Continuum.

In the future, we plan to expand Goldfish into the 3D Edge Cloud Space Continuum. To achieve this, we intend to integrate Orbital Edge Computing (OEC) requirements, including satellite positioning, into the Goldfish platform requirements. This will enable Goldfish to execute workflows within the 3D Continuum seamlessly. Moreover, we intend to expand GoldFish by implementing a smart and serialization-free actor state. This enhancement will allow the platform to identify if the actors necessitate a remote state, thus preventing unnecessary state persistence. As a result, resource usage will be optimized and latency reduced by skipping the loading of actor states. Finally, we aim to integrate Goldfish into ML pipelines in Edge-Cloud Continuum to facilitate statefull data-intensive workloads such as [23].

Acknowledgments

This work is partially funded by the Austrian Research Promotion Agency (FFG) under the project RapidREC (Project No. 903884). This research received funding from the EU's Horizon Europe Research and Innovation Program under Grant Agreement No. 101070186. EU website for TEADAL: <https://teadal.eu>.

References

- [1] [n. d.]. Akka Actor Systems. <https://doc.akka.io/docs/akka/current/general/actor-systems.html>
- [2] Gul Agha. 1986. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA.
- [3] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC '18)*. USENIX Association, Boston, MA, 923–935. <https://www.usenix.org/conference/atc18/presentation/akkus>
- [4] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard Paris, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference (Davis, CA, USA) (Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 41–54. <https://doi.org/10.1145/3361525.3361535>
- [5] Luciano Baresi and Danilo Filgueira Mendonça. 2019. Towards a Serverless Platform for Edge Computing. In *2019 IEEE International Conference on Fog Computing (ICFC)*. 1–10. <https://doi.org/10.1109/ICFC.2019.00008>
- [6] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. 2021. Durable functions: semantics for stateful serverless. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 133 (oct 2021), 27 pages. <https://doi.org/10.1145/3485510>
- [7] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (Cascais, Portugal) (SOCC '11)*. Association for Computing Machinery, New York, NY, USA, Article 16, 14 pages. <https://doi.org/10.1145/2038916.2038932>
- [8] Roberto Casadei, Ferruccio Damiani, Gianluca Torta, and Mirko Viroli. 2024. *Actor-Based Designs for Distributed Self-organisation Programming*. Springer Nature Switzerland, Cham, 37–58. https://doi.org/10.1007/978-3-031-51060-1_2
- [9] Marcin Copik, Alexandru Calotiu, Rodrigo Bruno, Gyorgy Rethy, Roman Böhringer, and Torsten Hoeffler. 2022. *Process-as-a-Service: Elastic and Stateful Serverless with Cloud Processes*. Technical Report.
- [10] Docker. 2024. Docker: Accelerated, Containerized Application Development. <https://www.docker.com/> Accessed: 2024-06-30.
- [11] Muhammad Shoaib Farooq, Shamyla Riaz, Adnan Abid, Kamran Abid, and Muhammad Azhar Naem. 2019. A Survey on the Role of IoT in Agriculture for the Implementation of Smart Farming. *IEEE Access* 7 (2019), 156237–156271. <https://doi.org/10.1109/ACCESS.2019.2949703>
- [12] The Linux Foundation. 2024. *Open Container Initiative Runtime Specification*. <https://github.com/opencontainers/runtime-spec/blob/main/spec.md>
- [13] Philipp Gackstatter, Pantelis Frangoudis, and Schahram Dustdar. 2022. Pushing Serverless to the Edge with WebAssembly Runtimes. 140–149. <https://doi.org/10.1109/CCGrid54584.2022.00023>
- [14] Philipp Haller. 2012. 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 (Tucson, Arizona, USA) (AGERE! 2012)*. Association for Computing Machinery, New York, NY, USA, 1–6. <https://doi.org/10.1145/2414639.2414641>
- [15] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651* (2018).
- [16] Raphael Hetzel, Teemu Kärkkäinen, and Jörg Ott. 2021. μ Actor: Stateful Serverless at the Edge. In *Proceedings of the 1st Workshop on Serverless Mobile Networking for 6G Communications (Virtual, WI, USA) (MobileServerless'21)*. Association for Computing Machinery, New York, NY, USA, 1–6. <https://doi.org/10.1145/3469263.3470828>
- [17] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (Virtual Event, Germany) (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 691–707. <https://doi.org/10.1145/3477132.3483541>
- [18] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv:1902.03383* [cs.OS]
- [19] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC '21)*. USENIX Association, 805–820. <https://www.usenix.org/conference/atc21/presentation/kotni>
- [20] Samuel Kounev, Nikolas Herbst, Cristina L. Abad, Alexandru Iosup, Ian Foster, Prashant Shenoy, Omer Rana, and Andrew A. Chien. 2023. Serverless Computing: What It Is, and What It Is Not? *Commun. ACM* 66, 9 (aug 2023), 80–92. <https://doi.org/10.1145/3587249>
- [21] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *2021 USENIX Annual Technical Conference (USENIX ATC '21)*. USENIX Association, 285–301. <https://www.usenix.org/conference/atc21/presentation/mahgoub>
- [22] Cynthia Marcelino and Stefan Nastic. 2024. CWASI: A WebAssembly Runtime Shim for Inter-function Communication in the Serverless Edge-Cloud Continuum. In *Proceedings of the Eighth ACM/IEEE Symposium on Edge Computing (Wilmingon, DE, USA) (SEC '23)*. Association for Computing Machinery, New York, NY, USA, 158–170. <https://doi.org/10.1145/3583740.3626611>
- [23] Maximilian Maresch and Stefan Nastic. 2024. VATE: Edge-Cloud System for Object Detection in Real-Time Video Streams. In *The 8th IEEE International Conference On Fog and Edge Computing (ICFEC 2024)*.
- [24] James Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2022. 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 (Minneapolis, MN, USA) (FRAME '22)*. Association for Computing Machinery, New York, NY, USA, 3–8. <https://doi.org/10.1145/3526059.3533618>
- [25] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association, Carlsbad, CA, 561–577. <https://www.usenix.org/conference/osdi18/presentation/moritz>
- [26] Chris Munns. 2019. Tracking the state of AWS Lambda functions. <https://aws.amazon.com/blogs/compute/tracking-the-state-of-lambda-functions>
- [27] Stefan Nastic, Philipp Raith, Alireza Furutanpey, Thomas Pusztai, and Schahram Dustdar. 2022. A Serverless Computing Fabric for Edge & Cloud. In *2022 IEEE 4th International Conference on Cognitive Machine Intelligence (CogMI)*. 1–12. <https://doi.org/10.1109/CogMI56440.2022.00011>
- [28] OpenFaaS. 2024. OpenFaaS – Serverless Functions Made Simple. <https://www.openfaas.com/> Accessed: 2024-06-30.
- [29] Daniel Barcelona Pons, Alvaro Ruiz Ollobarren, David Arroyo Pinto, and Pedro García Lopez. 2018. Studying the feasibility of serverless actors. In *Proceedings of the European Symposium on Serverless Computing and Applications, ES-SCA@UCC 2018, Zurich, Switzerland, December 21, 2018, Vol. 2330*. CEUR-WS.org, 25–29. <https://ceur-ws.org/Vol-2330/short1.pdf>
- [30] Philipp Raith, Stefan Nastic, and Schahram Dustdar. 2023. Serverless Edge Computing—Where We Are and What Lies Ahead. *IEEE Internet Computing* 27, 3 (2023), 50–64. <https://doi.org/10.1109/MIC.2023.3260939>
- [31] Redis. 2024. Redis. <https://redis.io/> Accessed: 2024-06-30.
- [32] Sanjin Sehic, Fei Li, Stefan Nastic, and Schahram Dustdar. 2012. A Programming Model for Context-Aware Applications in Large-Scale Pervasive Systems. In *Proceedings of the IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob 2012)*. IEEE Computer Society, 142–149. Vortrag: IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob 2012), Barcelona, Spain; 2012-10-08 – 2012-10-10.
- [33] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, 419–433. <https://www.usenix.org/conference/atc20/presentation/shillaker>
- [34] Simon Shillaker, Carlos Segarra, Eleftheria Mappoura, Mayeul Fournial, Lluis Vilanova, and Peter Pietzuch. 2023. Faabric: Fine-Grained Distribution of Scientific Workloads in the Cloud. *arXiv preprint arXiv:2302.11358* (2023).
- [35] Jonas Spenger, Paris Carbone, and Philipp Haller. 2024. *A Survey of Actor-Like Programming Models for Serverless Computing*. Springer Nature Switzerland, Cham, 123–146. https://doi.org/10.1007/978-3-031-51060-1_5
- [36] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M. Faleiro, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13 (2020), 2438–2452.
- [37] Fermyon Technologies. 2024. Spin - Fermyon Developer Documentation. <https://developer.fermyon.com/spin/v2/index> Accessed: 2024-06-30.
- [38] WasmEdge. 2024. WasmEdge. <https://wasmedge.org/> Accessed: 2024-06-30.
- [39] WasmEdge. 2024. WasmEdge Rust SDK. <https://github.com/WasmEdge/wasmedge-rust-sdk> Accessed: 2024-06-30.
- [40] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. 2023. Rise of the Planet of Serverless Computing: A Systematic Review. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 131 (jul 2023), 61 pages. <https://doi.org/10.1145/3579643>
- [41] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. 2018. Anna: A KVS for Any Scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 401–412. <https://doi.org/10.1109/ICDE.2018.00044>