Truffle: Efficient Data Passing for Data-Intensive Serverless Workflows in the Edge-Cloud Continuum

Cynthia Marcelino
Distributed Systems Group, TU Wien
c.marcelino@dsg.tuwien.ac.at

Stefan Nastic
Distributed Systems Group, TU Wien
snastic@dsg.tuwien.ac.at

Abstract—Serverless computing promises a scalable, reliable, and cost-effective solution for running data-intensive applications and workflows in the heterogeneous and limited-resource environment of the Edge-Cloud Continuum. However, building and running data-intensive serverless workflows also brings new challenges that can significantly degrade the application performance. Cold start remains one of the main challenges that impact the total function execution time. Further, since the serverless functions are not directly addressable, Serverless workflows need to rely on external (storage) services to pass the input data to the downstream functions. Empirical evidence from our experiments shows that the cold start and the function data passing take up the most time in the function execution lifecycle.

In this paper, we introduce Truffle – a novel model and architecture that enables efficient inter-function data passing in the Edge-Cloud Continuum by introducing mechanisms that separate computation and I/O, allowing serverless functions to leverage the cold starts to their advantage. Truffle introduces Smart Data Prefetch (SDP) mechanism that abstracts the retrieval of input data for the serverless functions by triggering the data retrieval from the external storage during the function's startup. Truffle's Cold Start Pass (CSP) mechanism optimizes inter-function data passing and data exchange within serverless workflows in the Edge-Cloud Continuum by hooking into the functions' scheduling lifecycle to trigger early data passing during the function's cold start. Experimental results show that by leveraging the data prefetching and cold-start data passing, Truffle reduces the IO latency impact on the total function execution time by up to 77%, improving the function execution time by up to 46% compared to the state-of-the-art data passing approaches.

Index Terms—Serverless, Cold start, Inter-function, Data-Intensive, Workflows

I. INTRODUCTION

Serverless computing is a paradigm where short-lived functions are executed in response to an external triggering event. Once the trigger event is received, the Serverless platform provisions the necessary environment infrastructure for the function to execute. The function provisioning at the startup of the function lifecycle is also known as cold start [1, 2]. Despite efforts in minimizing it [3, 4], cold starts represent a significant proportion of the function's total execution time, reaching up to 80% [5, 6]. In Serverless computing, where functions are short-lived, over 50% have execution times under 100ms. Thus, leveraging the longest tasks of the serverless function lifecycle, such as cold starts, becomes crucial for improving the overall function execution time [7, 8].

The stateless design of Serverless Computing pushes functions to rely on external systems for data exchange, with storage services representing over 60%, and messaging services comprising 38%. Furthermore, 30% of the functions process data over 10 MB, whereas 26% are over 100MB [8, 9]. Nevertheless, using external services to pass data adds significant network and latency overhead and consequently increases the function end-to-end execution [10, 11, 12]. Moreover, external services introduce additional development effort as applications must explicitly fetch their ephemeral input data [13, 14]. Therefore, constant efforts are necessary to improve the data passing between serverless functions.

The most common approaches for data passing in serverless functions include: (a) *Remote storage services*, such as object storage [15], Key-Value Store (KVS) [13, 12] and cache [16, 17] are the most common mechanism for data passing. Nevertheless, it increases network overhead and latency up to 95% [18]; (b) *Local storage:* approaches such as disk storage [10, 19], shared memory [20, 21] and local cache [22, 12] shift the data storage to local workers. It enables serverless platforms to exploit function and data locality to improve latency and throughput for co-located functions. Thus, decreasing remote communication. Unfortunately, these approaches do not consider the function cold start in the data passing mechanisms.

Furthermore, cold starts remain a challenge in Serverless computing. The Serverless platform provisions the full infrastructure for function execution. During the cold start phase, the infrastructure is not fully operational, leading to idle waiting time for both the function and the host. Consequently, the cold start influences the overall duration of end-to-end function execution [23, 24].

Common approaches that address cold start include: (a) *Caching* solutions propose to store full or partial function sandbox snapshots in memory for a certain time. Serverless platforms create a snapshot of the function sandbox to reuse it in new requests. Thus, Serverless platforms only provision the language runtime instead of the language runtime and infrastructure [5, 25]. Although caching decreases the cold start latency significantly, it increases resource usage [26]; (b) *Sandbox sharing* approach relies on multiple functions running in one sandbox. As functions run in a shared sandbox, platforms profit from a single cold start for multiple functions, reducing cold start latency and optimizing function execution time [11, 27, 28]. (c) *Sandbox techniques* such as tiny VM and Web Assembly VM propose minimal sandboxes with

only the necessary components and features required by the function. Additionally, it offers an additional layer of security. Thus, sandbox techniques decrease the resource provisioning overhead while increasing security [20, 29].

The presented approaches optimize serverless functions cold starts by addressing the function preparation and setup during the function initialization. Nevertheless, cold start solutions typically overlook the crucial phases of the function lifecycle, such as data transfer which along with the cold start have a significant impact on the Serverless function execution time.

In this paper we introduce Truffle, a modular architecture that enables input data fetching and passing. Truffle's communication technique leverages cold starts to transfer input data to another function efficiently. We can summarize our contributions as follows:

- **Truffle**, a novel model and architecture that separates computation and I/O to enable functions to execute tasks in parallel to the functions' cold starts for optimized end-to-end function execution.
- **The Smart Data Prefetch (SDP) mechanism**, a function input data fetching mechanism that abstracts the retrieval of input data for individual functions. Truffle seamlessly identifies the input data storage type and triggers the data retrieval at cold start, i.e., before function execution.
- **The Cold Start Pass (CSP) mechanism**, an inter-function data passing mechanism that optimizes data exchange between serverless functions. It identifies the function target host immediately after the scheduling phase preceding function execution. Truffle utilizes the cold start to transmit the data, optimizing inter-function communication. Consequently, when the target function executes, the input data is already stored in a buffer close to the function.

Truffle reduces the IO latency impact on the function execution time by up to 77%, achieving up to 46% reduction of the overall function execution time by leveraging the cold start to prefetch and pass the data to another function. Moreover, applications with longer cold starts of up to 10s profit nearly 30% more from Truffle compared to applications with short cold starts of up to 2s.

II. Motivation & Illustrative Scenario

A. Motivation

Serverless Computing is typically characterized by *stateless* and *non-addressable* functions deployed on entirely managed platforms. In the Edge-Cloud Continuum, Serverless computing enables efficient function execution on resource-limited edge devices while leveraging the Cloud for computing intense tasks, thus optimizing performance and reducing latency across distributed environment of the Edge-Cloud Continuum. By design, serverless functions lack direct addressability. Instead, these functions are accessible to clients through platform ingresses such as Load Balancers and API Gateways [30, 31]. Furthermore, Serverless platforms limit their function input data size (AWS λ: 6MB, OpenFaas: 1MB, GCP: 10MB), pushing
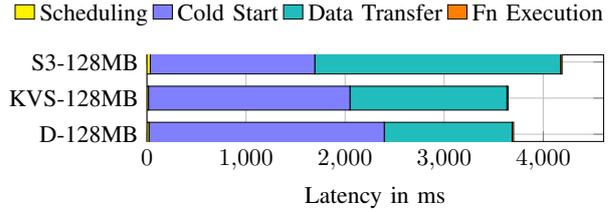

Fig. 1: Knative Serverless function lifecycle steps duration with data passing approaches Direct (D), Key-Value Store (KVS), and Object Storage (S3) for 128MB

functions to leverage external services such as object storage, KVS and message queues to pass larger amount of data which not only introduces complexity in the data handling process but also adds significant latency due to the reliance on external services [10, 32].

Fig. 1 shows the function latency of each step in the Serverless function lifecycle with the most common storage types, such as AWS S3, KVS, and Direct. Experimental evidence indicates that cold start and data transfer constitute a significant portion of the function execution time, and combined, they contribute to up to 99% of the function latency.
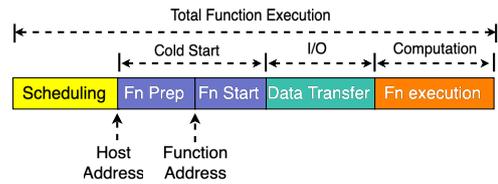

Fig. 2: Serverless Function Lifecycle with 1MB Input Data Transfer

Fig. 2 shows the four phases of function lifecycle: scheduling, cold Start, I/O and computation. Moreover, Fig. 2 shows that the function host is known immediately after the *Scheduling* phase. Nevertheless, the *Data Transfer* (I/O) only initiates after *Fn Start* when the Serverless function executes. In summary, the function and host remain idle until the serverless function provision has been completed. As shown in Fig. 2, the function *Cold Start* and *Data Transfer* contribute significantly to the function's end-to-end execution time. Moreover, the function is only available to process the request after *Fn Start* when the infrastructure has finished and the language runtime is fully running. However, the host is known after the *Scheduling*, implying that I/O tasks such as data transfer may initiate immediately after scheduling in parallel to the cold start. Despite this parallelization potential, the delays caused by cold starts and data transfers remain substantial, presenting challenges for optimizing performance in latency-sensitive applications such as data-intensive Serverless workflows in the Edge-Cloud Continuum.

To better motivate our challenges, we present an illustrative scenario 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 [33, 34, 35].

In Fig. 3, in *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 analyze them to identify specific fire patterns such as smoke and flames. Following the detection process, the *Object Detection* functions communicate directly with the *Alarm 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.

Truffle enhances Serverless workflows by separating computation and I/O, which allows Truffle to prefetch and transfer the input data during cold starts. Decreasing function latency enables quick data processing, which is crucial for real-time applications like fire detection and emergency response in the Edge-Cloud Continuum. Truffle optimizes data transfer between edge devices and cloud servers, ensuring data is available for computation as soon as the functions are initialized. By separating I/O from computation, Truffle can incorporate the input
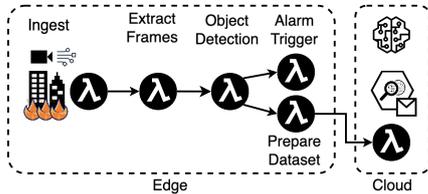

Fig. 3: Simplified Serverless Workflow for Fire Detection for Smart Cities

data transfer into the function startup phase, facilitating faster and more efficient data handling, essential for maintaining data-intensive Serverless workflows in the Edge-Cloud Continuum.

III. TRUFFLE MODEL & ARCHITECTURE OVERVIEW

A. Truffle Model

The execution time τ of a Serverless function can be modeled by the sum of its individual tasks. Given a Set of functions λ with function index i, where $i \in \mathbb{N}$. Thus, the cold start β_i can be defined as:

$$\beta_i = v_i + \eta_i \tag{1}$$

Where v represents the function infrastructure setup and η the function startup. Cold starts introduce additional steps such as scheduling, infrastructure setup, and language runtime startup, resulting in increased latency. Consequently, cold starts directly influence the overall latency of end-to-end functions. Additionally, in the state-of-the-art platforms, every step of the Serverless functions cold starts are executed in sequence. Truffle proposes to execute the cold and the data transfer in parallel to optimize the function latency. Thus, the improvement φ_i is defined as follows:

$$\varphi_i = \max(\beta_i, \delta_i) \tag{2}$$

Where β_i is the function cold start and δ_i is the data transfer time. Since Truffle executes the cold start and data transfer simultaneously, φ is defined by the longest task between these two tasks, i.e., cold start and data transfer.

Therefore, the end-to-end execution time τ of a Set of functions λ can be described as follows:

$$\tau(\lambda_i) = \sum_{i=1} \alpha_i + \max((v_i + \eta_i), \delta_i) + \gamma_i \tag{3}$$

Here, the total execution time of a function λ_i is equal to the scheduling α_i plus the maximum value between the sum of function infrastructure setup v_i + function startup η_i and data transfer δ_i. Thus, γ_i represents the function execution time [10, 36, 37, 38]. Therefore, we can define truffle improvement Δ_i in the end-to-end execution time as follows:

$$\Delta_i = (\beta_i + \delta_i) - \max(\beta_i, \delta_i) \tag{4}$$

Finally, the Truffle optimization model aims to minimize the end-to-end execution of a Serverless workflow composed of a Set of serverless functions λ. To achieve that, Truffle considers three main phases for each function: the scheduling time α_i, the longer time between the cold start and data transfer $\max(\beta_i, \delta_i)$, and the function execution time (γ_i) as follows:

$$\min \sum_{i=1}^{n} (\alpha_i + \max(\beta_i, \delta_i) + \gamma_i) \tag{5}$$

Hence, Truffle improvements are directly connected to the cold start latency and data input size. Longer cold starts associated with longer data transfer have more profit than shorter data transfer and cold starts.

B. Truffle Architecture Overview

In the state-of-the-art platforms, I/O tasks such as data transfer initiates after fully provisioned function. Furthermore, as Serverless functions are sensitive to network latency, the function execution time grows proportionally to the network data transfer [10, 11, 39].

Truffle is designed to optimize the data input handling for serverless functions. It introduces a novel architecture to identify, retrieve, and transfer the data input based on function properties such as storage type, event type, serverless platform, and orchestration tools. Truffle proposes to separate the computation and I/O so that the network data input transfer can start simultaneously with the cold start process right after scheduling. This means that the functions that are currently running, also known as hot functions, may not benefit from Truffle. In such cases, Truffle works as a proxy and only pass the data through it.

Despite extensive efforts to mitigate cold starts in serverless architectures, the issue persists and can significantly impact the total function execution. By enabling functions to transfer data during cold starts, typically an idle time for the serverless function, Truffle reduces the function latency while alleviating serverless functions' burden of fetching input data. Truffle is located on each computing node to intercept local requests to improve inter-function communication latency. Additionally, it uses orchestration tools like Kubernetes for better scalability and throughput.

Fig. 4 shows Truffle's interaction with the function and the serverless and orchestration platform. Truffle has a modular architecture that allows each component to be extended and eventually replaced. Truffle components enable two main mechanisms: Smart Data Prefetch and Cold Start Data Pass.
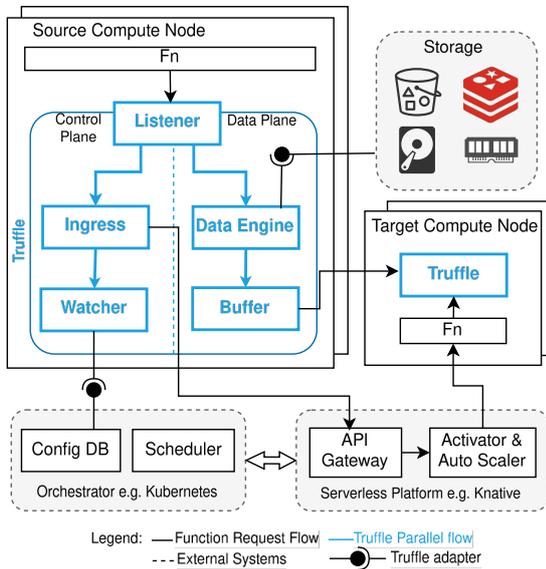

Fig. 4: Truffle Architecture Overview

Smart Data Prefetch, detailed in Section IV-A, is triggered at every function start and retrieves the function input data from the respective storage such as direct, object storage, and KVS in parallel to the cold start. Whereas *Cold Start Data Pass*, described in Section IV-B, facilitates and optimizes the data passing between Serverless functions.

1) Truffle Components: Truffle is composed of five components: Listener, Data Engine, Ingress, Watcher, and Buffer.

a) Listener: It connects to the Serverless functions invoking events. The goal of the listener is to intercept the function request trigger and notify Truffle of an incoming request. It supports multiple serverless functions invoking events such as HTTP requests, streams, queues, and jobs.

b) Ingress: It concurrently triggers the Serverless platform and Truffle's Data Engine to initiate the function input data fetching. More specifically, the ingress accepts requests from the source client functions. Then, it sends an event to the Serverless platform with a key reference to the function input data stored in Truffle's local buffer. Subsequently, the serverless platform is responsible for forwarding the request and implementing the function's scaling procedures if necessary.

c) Data Engine: It identifies the storage type of incoming function data, retrieves it, and stores it in the buffer. This component enables functions to seamlessly receive input data, regardless of their input data storage type, such as KVS and object storage. Truffle's architecture design with adapters facilitates the extension of the data engine, supporting various storage types and multiple cloud providers. Consequently, it increases flexibility and facilitates serverless function development.

d) Watcher: It connects to the underlying orchestration tool, such as Kubernetes via the control API and listens for live events, such as scheduling events and function host assignments. Its primary task is identifying the assigned host for a target function, allowing Truffle to pass data between functions. The watcher recognizes the data input source and notifies Truffle ingress once the target function has been assigned to a worker or if the function already has an assigned worker, indicating that the function is already running. In cases of hot function, Truffle does not interfere and forwards the request without any modification. Moreover, the watcher employs an adapter to communicate with multiple orchestration tools, such as Kubernetes and Nomad.

e) Buffer: It stores the data until the target function is fully provisioned. The buffer can be placed on each node locally or remotely via external services. A local buffer keeps the data closer to the function and enables high-speed access to the data via local storage, such as in-memory storage. The buffer can also leverage remote services, such as databases and KVS, to provide flexibility and scalability. In cases where local computing resources are limited, the remote buffer enables functions to leverage the cold start for data passing with on-demand auto-scaling without local resource overhead.

IV. TRUFFLE RUNTIME MECHANISMS

A. The Smart Data Prefetch

Truffle Smart Data Prefetch (SDP) abstracts and optimizes the data fetching from the serverless function. Fig. 5 shows how Truffle fetches the function input data. ① First, Truffle ingress receives the client request with all the required information to retrieve the input data as arguments to the function handler, such as the bucket name, object ID, and credentials. ② Truffle ingress sends two simultaneous requests. One request to the platform Activator & Auto Scaler and one request to the Data Engine. ②ⓐ At this point, Truffle initiates an asynchronous request path parallel to the original request flow, e.g., requests via serverless platforms. Truffle triggers the Data Engine component to identify and start the data fetching. ③ The Auto Scaler scales up if necessary, and the scheduler gets then notified about new scheduling. ③ⓐ Truffle initiates the input data fetching from the respective storage, i.e., direct, object storage, or KVS. For specific cases where special access and roles are requested, such as AWS S3 credentials and keys, the function must specify and give Truffle the right access via specific function properties. ④ After scheduling, The Activator & Auto Scaler get notified the scheduling has finished and the request may resume. ④ⓐ Once the data fetching is finished in the Data Engine, Truffle stores the input data size in the Buffer. ⑤ The Activator resumes the request and forwards it to the Serverless Platform Queue Proxy. ⑥ The Queue Proxy forwards the request to the Serverless function handler during Fn Execution. ⑦ Once the function executes in Fn Execution, it retrieves its input data from the Truffle Buffer.

Algorithm 1 details the smart data prefetch. Let R be the incoming request, B_N the buffer name, S_T the storage type, and content reference C_R the input. Lines 1 and 2 create an empty buffer B and storage client SC respectively. Let B_T be the buffer type in supported buffer collection BC. If buffer type B_T is not empty and equals the buffer name B_N. Then assign buffer type B_T to buffer instance B. Let S_E be storage in supported storage collection SC. If S_E is not empty and
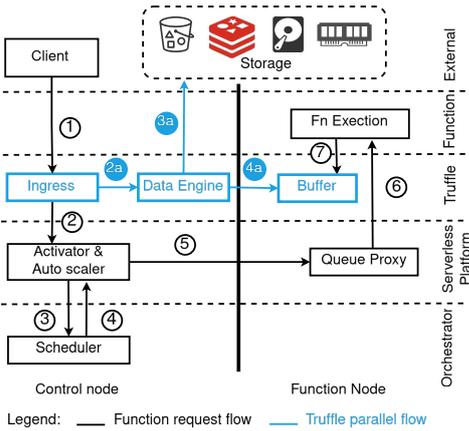

Fig. 5: Truffle Smart Data Prefetch

Algorithm 1 Truffle Smart Data Prefetch

Require: R: incoming request
Require: B_N: buffer name
Require: S_T: storage type
Require: C_R: content reference
1: $B \leftarrow$ buffer
2: $SC \leftarrow$ storage
3: **for all** $B_T \in BC$ **do**
4:     **if** $B_T \neq \emptyset \wedge B_T = B_N$ **then**
5:         $B \leftarrow B_T$
6:     **end if**
7: **end for**
8: **for all** $S_E \in SC$ **do**
9:     **if** $S_E \neq \emptyset \wedge S_E = S_T$ **then**
10:         $SC \leftarrow$ getclient(S_T)
11:     **end if**
12: **end for**
13: $C \leftarrow SC.get(C_R))$
14: $B.set(C)$

S_E equals the storage type S_T, then assign the specific storage client for storage type S_T to SC. In line 13, Truffle calls storage client SC to get the data based on content reference C_R input and assigns it to content C. Finally, set content C in buffer B.

B. Cold Start Pass Mechanism

For data passing between two functions, Truffle first identifies where the source functions want to send the data. Then, once the target receives the data, Truffle stores it in a local buffer.

Fig. 6 shows the communication between two serverless functions with its multiple platforms integration. Fig. 6 shows a simplified interaction between the orchestration and serverless platform for synchronous requests.

First, ① the source function fires a request to the Local truffle on the source. Then, ② Truffle accepts the request and identifies the target function based on information provided to the function. Then Truffle sends a request with an input data reference key to the serverless platforms such as Knative Activator. In ②ⓐ, simultaneously to ②, Truffle starts to listen for the target function host address. The function address is available after the scheduling phase. ③ The platform activator receives the request and scales the function up via the orchestration tools mechanisms. ④ Scheduling is finished and stored in the orchestration storage, e.g., Kubernetes etcd. ⑤ Notifies the serverless platform. ⑥ Resumes the request to the target Host Proxy. ⑥ At this point, the Local Truffle on the source is aware of the target host reported by ⑤, then simultaneously to ⑥, it starts transferring the outgoing input data to the Truffle buffer on the target host worker. In ⑦, the Host Proxy forwards the request to the serverless platform ingress, such as Queue Proxy. In ⑧, the Queue Proxy forwards the request to the serverless function handler during Fn Execution. ⑨ The function handler retrieves its input data
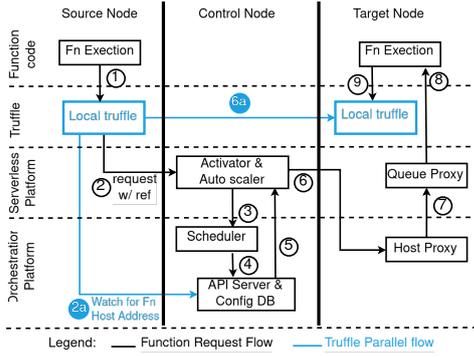
Fig. 6: Truffle Cold Start Pass

Algorithm 2 Truffle Cold Start Pass Request

Require: T_F: target function
Require: T_C: target content
1: $H_A \leftarrow \emptyset$
2: $W \leftarrow$ resource watcher
3: **for all** $F(F \in F_E, \forall F_E \in W)$ **do**
4:     **if** $F_H \neq \emptyset \land T_F = F_N$ **then**
5:         $H_A \leftarrow F_H$
6:     **end if**
7: **end for**
8: *call_target_host_truffle(H_A, T_C)*

---

from the Local truffle buffer with the input data reference key existent in the request from ②.

Truffle Cold Start Data Pass relies on a data-passing request, Algorithm 2 executed on the truffle instance on the source function host. Once the an incoming request, is executed in the target function host, the function retrieves the data from the buffer. Let the target function T_F and target content T_C be the input. Let F be function information in a function event F_E, where all function events belong to the watcher W. If function host information F_H is not empty and target function T_F equals the function event name F_N. Then, assign function host F_H to target host address H_A. Once H_A is known, stop listening for new events. Finally, call_target_host_truffle with host address H_A and target content T_C.

V. TRUFFLE PROTOTYPE IMPLEMENTATION

Truffle is published as an open-source framework part of the Polaris SLO CLoud. Polaris itself is part of the Linux Foundation Centaurus project. Truffle source code is available on GitHub[1]. It is implemented in Go and currently supports Kubernetes as the orchestration tool and Knative as the Serverless Platform.

Truffle runs as a DaemonSet on Kubernetes to ensure every node has an active instance and supports high availability with multiple instances per node, managed by a load balancer. Functions connect to Truffle through the HostIP and defined

[1]https://github.com/polaris-slo-cloud/truffle

port. To improve latency, Truffle uses goroutines for parallel task execution, listening for events, and leveraging httputil Golang libraries to create a reverse proxy. Its data engine fetches data from remote services using external libraries, such as golang aws-sdk-go for AWS S3 and go-redis for KVS, and reads directly from the HTTP request body for direct communication. The Ingress component handles incoming requests, initiates parallel tasks with goroutines, creates a reference key, and forwards this to the Serverless platform, allowing the Serverless function to retrieve data from the local Truffle instance buffer. The Watcher monitors orchestration tools, specifically Kubernetes pod events via Kube API, enabling data transfer between functions immediately after a function is assigned to a node.

VI. EVALUATION

A. Overview

We design experiments to evaluate our proposed solutions with two Serverless workflows: Chained Functions and Video Analytics. Our Chained functions workflow is composed of two functions a and b that represents sequential execution of data-intensive functions. It receives certain input data and forwards it to the function. Our Video Analytics workflow is composed of the functions: *Video Streaming*, *Decoder*, and *Image Recognition* inspired by [40, 41] using most important invocation patterns from serverless computing fan-out and fan-in [31]. We compare Truffle to three different data-passing storage types: Direct, Object Storage (S3), and Redis (KVS). As Truffle focuses on data-passing communication, we measure the time when the data is sent until the target functions receive the data.

To verify the proposed solution in this paper, we evaluate our framework by conducting experiments that measure data passing latency during the cold start.

a) Metrics: Latency metric shows the time when a source function sends a request with a message until the target function receives it. This experiment includes also the entire target function startup including scheduling, cold start, and data transfer latency. Cold Start Delay metrics evaluate how cold start delays can influence the function's end-to-end execution time and how truffle can improve these delays to transfer large amounts of data. Therefore, we increase the function cold start in seconds and collect the function data passing latency (sec) for different input sizes.

B. Experiment Setup

We conduct experiments featuring the designed serverless workflow with AWS S3 and Redis KVS as the baseline. We use MicroK8s as the orchestration tool, and Knative as the Serverless Platform. Our microk8s cluster contains multiple nodes to ensure remote data passing experiments. For this purpose, we leverage Kubernetes node affinity to ensure the function scheduling on different nodes. Each cluster node is a VM with 4-core Intel Xeon Processor 2GHz, 8GB RAM with Ubuntu 22.04 LTS. To avoid misleading results, we repeated the experiments and calculated the average result. Except for

AWS S3, every tool was locally installed in the cluster with its vanilla version without configuration modifications including storage type Redis. For experiments with object object storage type, we leverage AWS S3. The same experiment setup is used to obtain the results for Fig. 1 and Fig. 2

C. Latency Results

In this experiment, we analyze how Truffle improves the total workflow latency when compared to the baselines. Fig. 7 presents the total workflow execution latency for Chained Functions while highlighting the function lifecycle phases. Due to Truffle's SDP that pre-fetches the data during the cold start, Truffle reduces the IO latency impact and consequently decreases the total workflow execution latency. More specifically, in Fig. 8, we show the IO latency impact on the total workflow latency. Truffle demonstrates an IO latency impact reduction of up to 77%, improving the overall workflow execution latency by up to 46% compared to the baselines: Direct, KVS and S3.

Further in Fig. 9 and Fig. 10, we use http synchronous serverless invocation trigger. We evaluate two applications: Chained Functions and Video Analytics. We increase the input size to validate the solution with different loads. Fig. 9 shows latency results for Chained Functions and Fig. 10 shows latency results for Video Analytics workflow. Fig. 9 shows three different data passing strategies: direct, KVS, and object storage AWS S3. The function input size is represented on axis x in MB and latency is shown on axis y in seconds.

Fig. 9a shows that Truffle has latency from 1.852 sec to 2.697 sec while Direct HTTP requests show from 2.329 sec to 4.353 sec. Fig. 9d shows an improvement of up to 46% on the direct data passing via Truffle. In Fig. 9b, Truffle with KVS has latency from 2.060 sec to 4.827 sec while vanilla KVS shows from 2.121 sec to 5.073 sec. Truffle shows an improvement of up to 5% on the KVS data passing compared to the baseline (Fig. 9d). Fig. 9c indicates that Truffle with S3 shows latency
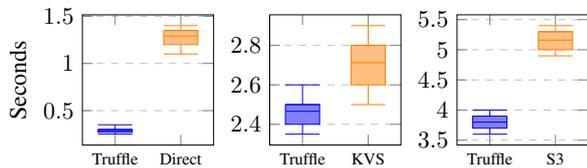
between 1.838 sec and 6.210 sec, whereas S3 alone ranges from 2.711 sec to 7.560 sec. This shows an improvement of up to 18% in data passing via Truffle (Fig. 9d).

In the Video Analytics use case in Fig. 10, Fig. 10a reveals that Truffle has latency from 4.208 sec to 5.163 sec, while Direct HTTP requests range from 4.363 sec to 5.495 sec, an improvement of up to 6% in direct data passing with Truffle (Fig. 10d). In Fig. 10b, Truffle with KVS has latency between 3.954 sec and 8.631 sec, compared to KVS alone, which ranges from 4.637 sec to 10.315 sec. This represents an improvement of up to 16% in data passing with Truffle, shown in Fig. 10d. While in Fig. 10c indicates that Truffle with S3 shows latency from 3.958 sec to 5.712 sec, while S3 alone ranges from 5.568 sec to 7.536 sec. Truffle demonstrates an enhancement of up to 24% (Fig. 10d).

D. Cold Start Delay Results

In this experiment, we compare the same storage approach in its conventional usage, i.e., after the function startup, and with Truffle, which leverages the cold start for fetching the input data. As Truffle leverages the cold start to fetch and pass data, the cold start latency directly impacts the function execution time and truffle performance. Thus, in this experiment, we increase the cold start latency for the different storage types, object storage AWS S3 and KVS. Truffle still leverages the same storage type as the given baseline. Furthermore, we add additional delay to the existing cold start to simulate longer cold start delays. Therefore, in Fig. 11, the baseline and truffle never intercept at $x = 0$ because x represents the additional cold start. For example, $x = 0$ in Section VI-D is similar to Fig. 9c $x = 100MB$. Hence, we evaluate how Truffle benefits applications with longer cold starts.

Fig. 11 shows the latency in seconds in axis y and the cold start delay in axis x using object object storage type AWS S3 and KVS. Section VI-D shows the latency and cold start delay for an input size of 100MB while The orange line represents AWS S3 and KVS, and the blue line is Truffle + AWS S3. As applications only fetch the input data after the function has fully initialized, in Section VI-D, the function latency (axis y) grows linear to the cold start delay x in orange. On the other hand, in the blue line, Truffle delays the function latency increase. Truffle uses the full cold start to transfer the input data from 0 to 4s in axis x. Only at the second 6s in axis x does the function latency start growing linearly. Consequently, Truffle shows a latency decrease of up to 30% in applications with longer cold starts. While applications with short cold start might profit 3%. In Section VI-D, KVS shows a latency from 3.7 seconds to 14 seconds in orange while Truffle shows from 1.6 seconds to 12 seconds in blue, which means Truffle decreases the latency up to 43%. In Section VI-D, we observe that the input data transfer is faster than any cold start delay, which means even the shortest cold start delay is already sufficient to transfer the input data completely, which means Truffle optimizes execution time even when data transfer is fast enough to mask the cold starts. Therefore, this experiment shows similar linear growth regardless of the cold start delay.
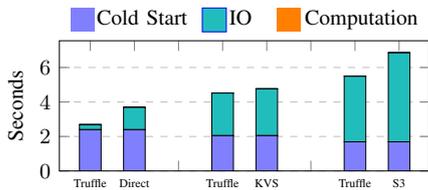

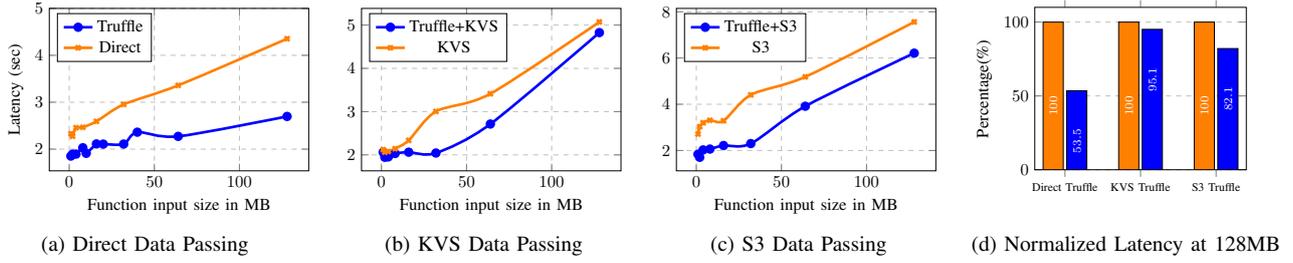Fig. 7: Total Workflow Execution Latency for 128MB, including function lifecycle phases

Fig. 8: IO latency impact on the total workflow execution for 128MB

(a) Direct Data Passing   (b) KVS Data Passing   (c) S3 Data Passing   (d) Normalized Latency at 128MB

Fig. 9: Chained Functions Latency

(a) Direct Data Passing   (b) KVS Data Passing   (c) S3 Data Passing   (d) Normalized Latency at 40MB
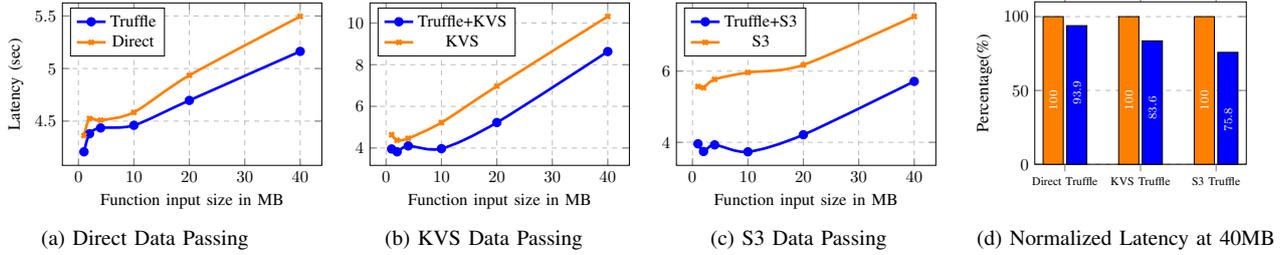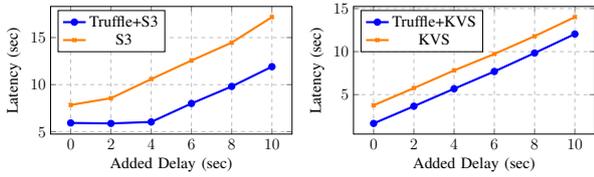
Fig. 10: Video Analytics Latency

Fig. 11: Additional Cold Start Delay with 100MB input

E. Discussion

Truffle enhances serverless function data transfer and reduces latency up to 46% by effectively utilizing cold start delays for data fetching. Therefore, it optimizes function initialization times, which is especially beneficial for data-intensive applications. Additionally, Truffle can integrate with different storage types and orchestrators via Truffle adapters. Furthermore, Truffle proposes function data fetching from different storage types, such as object storage, KVS, or direct passing. Truffle model Section III-A shows that we can minimize function execution time by taking the shortest path which is either data transfer duration or cold start. Effectively, it enables us to overlap communication with cold start. Truffle utilizes this model and reduces the end-to-end latency by overlapping data transfer with the cold start.

As discussed in Section III-A, functions with longer cold starts profit more from Truffle than those with shorter cold starts. Results in Section VI-D show that functions with longer cold starts profit up to 30% more than functions with shorter cold starts. Truffle leverages the cold start for data fetching and transfer, which would otherwise be idle during the function lifecycle. Thus, small input data sizes do not significantly improve as functions with larger data input sizes.

Truffle's data transfer optimization reduces latencies and decreases function execution times for data-intensive tasks with a cold start. Therefore, during hot function executions, i.e., functions already running, Truffle checks whether the functions are running and forwards the data to the Serverless platform, acting as a transparent proxy. In such cases, Truffle's decision-making process during runtime is a trade-off. In the future, we plan to optimize Truffle's proxy for hot function to address this technical limitation. In our evaluation, discussed in Section VI, we measure and analyze Truffle's improvement, i.e., *Latency* for a serverless function with a cold start and *Cold Start* duration, which shows how functions with long cold starts benefit from Truffle compared to those with short cold starts. We did not conduct throughput experiments to measure the data volume Truffle can handle, as it requires executing many requests in parallel. Therefore, throughput experiments only measure hot functions that do not contain any Truffle improvement. Furthermore, we have conducted experiments with real-world applications to demonstrate broad applicability. However, the specific workflows we used in our experimental evaluation (Chained Functions and Video Analytics) may only represent some of the possible use cases for serverless computing. Moreover, our experiments use default configurations for the external systems such as Kubernetes and Knative which means any customized configuration that affects the performance of these systems might impact Truffle's result. Additionally, we compare Truffle with three different data-passing storage types: Direct, Object Storage (S3), and Redis (KVS), using the default configuration. The choice of these particular storage types and

their default configurations may impact the results, as our experiments rely on the performance of these external services. Thus, any potential customized configuration affecting the system, network latency, or performance issues with S3 or Redis could affect our results.

VII. Related Work

Cold Starts. SEUSS [3] suggests creating and caching locally a snapshot to skip repetitive cold start steps, e.g., language runtime download and setup. Xanadu [4] reduces cascading cold start overheads by using speculative and just-in-time resource provisioning in a workflow. However, Xanadu focuses only the cold start mitigation which means functions are still responsible for fetching its input data. Nevertheless, SEUSS does not address the data passing which contributes to the function latency significantly. Faasm [20], CWASI [28] and SAND [11] follow a similar approach to sandbox sharing. These solutions propose to reuse a single sandbox for highly trusted functions, such as functions from the same workflow. As functions are executed in a single sandbox, there is only a simple cold start. Hence, reducing cold start from n functions in the workflow to a single cold start process. Nevertheless, sandbox sharing reduces the isolation and security provided by containerization.

Truffle does not reduce the cold start latency but instead leverages it to optimize data transfer. It achieves this by integrating SDP and CSP, reducing overall latency, and improving serverless applications' efficiency. Cold start optimization approaches are not seen as contenders but rather as approaches that could be used in conjunction with Truffle.

Function Data Handling. Lambdata [14] enables Serverless functions to create a cache of the function data so that subsequent requests and scheduling are prioritized based on the function and data locality. Although Lambdata has data locality in its scheduling, it does not consider the cold start. Thus, they do not implement input data fetching. SONIC [10] relies on local storage and remote object storage to exchange data. For co-located functions, it exchanges data via local storage. For remote inter-function communication, Sonic leverages a metadata manager to send function references e.g. IP and Path. For persistent storage, Sonic also supports object storage such as AWS S3.

Truffle improves the function of data handling by introducing the Smart Data Prefetch (SDP) mechanism, which automatically prefetches and transfers data. It reduces dependence on third-party services and abstracts data handling from the developer.

Inter-function Communication. CloudBurst [12] relies on a local cache on each function host to allow low latency access to frequent data from a remote KVS. Although remote KVS is low latency and highly scalable KVS, it proposes data exchange via remote third-party service. Pocket [13] introduces elastic ephemeral storage for intermediate data exchange for serverless functions. Pocket in-memory storage offers scalability and improves communication compared to traditional object storage such as S3. Nevertheless, it still relies on an additional storage solution. XDT [42] creates direct communication between the functions by buffering the content on the sender and shipping a content key and the sender address. However, XDT does not consider the function cold start neither different storage types for the function data fetching, limiting its approach to inter-function communication.

Truffle improves inter-function communication by implementing the Cold Start Pass (CSP) mechanism. This mechanism leverages the cold start to transfer data between functions, simplifying data transfer processes. Truffle integrates CSP with components such as Ingress and Buffer components, enhancing overall workflow efficiency and reducing latency in serverless applications.

VIII. Conclusion & Future Work

In this paper, we present Truffle, a novel architecture that enables serverless functions to execute parallel tasks to the cold start and consequently optimize the overall function end-to-end execution time. Our experimental evidence indicates that the longest tasks during overall function execution are cold start and data transfer. Moreover, our analysis reveals that every task in the state-of-the-art serverless platform is executed sequentially, even when not dependent on one another.

To address this issue, we have designed Truffle, which facilitates efficient data passing among Serverless functions. Truffle enables input data fetching and inter-function data passing during the cold start, i.e., ahead of the function execution. Furthermore, Truffle abstracts the function input data fetching for multiple storage types and optimizes the end-to-end function execution time.

We evaluated Truffle by running different real-world Serverless applications, chained functions, and video analytics. Our experiments show that Truffle improves latency up to 46% compared to the vanilla state-of-the-art Serverless platforms. The experiments also show that Serverless workflows with longer cold start delays profit more from Truffle, emphasizing its significance for applications that demand longer initialization, such as machine learning or not optimized language runtimes. Functions with longer cold starts of up to 10s profit nearly 30% more than functions with shorter cold starts of up to 2s.

In the current state-of-the-art sequential lifecycle phase execution, important tasks to the function are rarely applied due to increased function latency. We plan to leverage Truffle to execute tasks important to the Serverless function, such as serialization, deserialization, and argument validation. Serverless functions can potentially leverage Truffle Buffer to store its state handling and leverage the cold start to perform data-related tasks such as serialization without increasing the latency.

REFERENCES

[1] Johann Schleier-Smith et al. 2021. What serverless computing is and should become: the next phase of cloud computing. *Commun. ACM*, 64, 5, (Apr. 2021), 76–84.

[2] Philipp Raith et al. 2023. Serverless edge computing—where we are and what lies ahead. *IEEE Internet Computing*, 27, 3, 50–64.

[3] James Cadden et al. 2020. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (EuroSys '20). Association for Computing Machinery, Heraklion, Greece.

[4] Nilanjan Daw et al. 2020. Xanadu: mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference* (Middleware '20). Association for Computing Machinery, Delft, Netherlands, 356–370.

[5] Alexander Fuerst and Prateek Sharma. 2021. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '21). Association for Computing Machinery, Virtual, USA, 386–400.

[6] Xuanzhe Liu et al. 2023. Faaslight: general application-level cold-start latency optimization for function-as-a-service in serverless computing. *ACM Trans. Softw. Eng. Methodol.*, 32, 5, (July 2023).

[7] Arjun Singhvi et al. 2021. Atoll: a scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing* (SoCC '21). Association for Computing Machinery, Seattle, WA, USA, 138–152.

[8] Simon Eismann et al. 2022. The state of serverless applications: collection, characterization, and community consensus. *IEEE Transactions on Software Engineering*, 48, 10, 4152–4166.

[9] Simon Eismann et al. 2021. Serverless applications: why, when, and how? *IEEE Software*, 38, 1, 32–39.

[10] Ashraf Mahgoub et al. 2021. SONIC: application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, (July 2021), 285–301.

[11] Istemi Ekin Akkus et al. 2018. SAND: towards High-Performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, (July 2018), 923–935.

[12] Vikram Sreekanti et al. 2020. Cloudburst: stateful functions-as-a-service. *Proc. VLDB Endow.*, 13, 2438–2452.

[13] Ana Klimovic et al. 2018. Pocket: elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, (Oct. 2018), 427–444.

[14] Yang Tang and Junfeng Yang. 2020. Lambdata: optimizing serverless computing by making data intents explicit. *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, 294–303.

[15] 2023. Amazon s3. https://aws.amazon.com/de/s3/.

[16] Ao Wang et al. 2020. InfiniCache: exploiting ephemeral serverless functions to build a Cost-Effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, (Feb. 2020), 267–281.

[17] Zhuo Huang et al. 2023. Duo: improving data sharing of stateful serverless applications by efficiently caching multi-read data. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 875–885.

[18] Swaroop Kotni et al. 2021. Faastlane: accelerating Function-as-a-Service workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, (July 2021), 805–820.

[19] Cynthia Marcelino et al. 2024. Goldfish: serverless actors with short-term memory state for the edge-cloud continuum. In *Proceedings of the 14th International Conference on the Internet of Things* (IoT '24). ACM, Oulu, Finland.

[20] Simon Shillaker and Peter Pietzuch. 2020. Faasm: lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, (July 2020), 419–433.

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

[22] Djob Mvondo et al. 2021. Ofc: an opportunistic caching system for faas platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems* (EuroSys '21). Association for Computing Machinery, Online Event, United Kingdom, 228–244.

[23] Parichehr Vahidinia et al. 2020. Cold start in serverless computing: current trends and mitigation strategies. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, 1–7.

[24] Paulo Silva et al. 2020. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference* (Middleware '20). Association for Computing Machinery, Delft, Netherlands, 1–13.

[25] Rohan Basu Roy et al. 2022. Icebreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '22). Association for Computing Machinery, Lausanne, Switzerland, 753–767.

[26] Li Pan et al. 2022. Retention-aware container caching for serverless edge computing. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, 1069–1078.

[27] Edward Oakes et al. 2018. SOCK: rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, (July 2018), 57–70.

[28] Cynthia Marcelino and Stefan Nastic. 2023. Cwasi: a webassembly runtime shim for inter-function communication in the serverless edge-cloud continuum. In *The Eighth ACM/IEEE Symposium on Edge Computing (SEC 2023)*.

[29] Alexandru Agache et al. 2020. Firecracker: lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, (Feb. 2020), 419–434.

[30] Stefan Nastic et al. 2022. A serverless computing fabric for edge & cloud. In *2022 IEEE 4th International Conference on Cognitive Machine Intelligence (CogMI)*, 1–12.

[31] Eric Jonas et al. 2019. Cloud programming simplified: a berkeley view on serverless computing. (2019).

[32] Joseph M. Hellerstein et al. 2018. Serverless computing: one step forward, two steps back. (2018).

[33] Juan José López Escobar et al. 2024. Unleashing the power of decentralized serverless iot dataflow architecture for the cloud-to-edge continuum: a performance comparison. *Annals of Telecommunications*, 1–14.

[34] C. Sicari et al. 2023. Tema: event driven serverless workflows platform for natural disaster management. In *2023 IEEE Symposium on Computers and Communications (ISCC)*. IEEE Computer Society, Los Alamitos, CA, USA, (July 2023), 1–6.

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

[36] János Czentye and Balázs Sonkoly. 2024. Serverless application composition leveraging function fusion: theory and algorithms. *Future Generation Computer Systems*, 153, 403–418.

[37] Alim Ul Gias and Giuliano Casale. 2020. Cocoa: cold start aware capacity planning for function-as-a-service platforms. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 1–8.

[38] Thomas Pusztai et al. 2024. Hyperdrive: scheduling serverless functions in the edge-cloud-space 3d continuum. In *Proceedings of the Ninth ACM/IEEE Symposium on Edge Computing* (SEC '24). Association for Computing Machinery, Rome, Italy.

[39] Tom Kuchler et al. 2023. Function as a function. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (SoCC '23). Association for Computing Machinery, Santa Cruz, CA, USA, 81–92.

[40] Dmitrii Ustiugov et al. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. ACM.

[41] 2023. Vswarm - serverless benchmarking suite. https://github.com/vhive-serverless/vSwarm.

[42] Dmitrii Ustiugov et al. 2023. Expedited data transfers for serverless clouds. (2023).