Styx: An Efficient Workflow Engine for Serverless Platforms

Abhisek Panda, Smruti R. Sarangi

***Abstract*—Serverless platforms are widely adopted for deploying applications due to their autoscaling capabilities and pay-as-you-go billing models. These platforms execute an application's functions inside ephemeral containers and scale the number of containers based on incoming request rates. To meet service level objectives (SLOs), they often over-provision resources by maintaining warm containers or rapidly spawning new ones during traffic bursts. However, this strategy frequently leads to inefficient resource utilization, especially during periods of low activity. Prior research addresses this issue through intelligent scheduling, lightweight virtualization, and container-sharing mechanisms. More recent work aims to improve resource utilization by remodeling the execution of a function within a container to better separate compute and I/O stages. Despite these improvements, existing approaches often introduce delays during execution and induce memory pressure under traffic bursts.**

In this paper, we present Styx, a novel workflow engine that enhances resource utilization by intelligently decoupling compute and I/O stages. Styx employs a fetch latency predictor that uses real-time system metrics from both the serverless node and the remote storage server to accurately estimate prefetch operations, ensuring input data is available exactly when needed. Furthermore, it offloads the output data upload operation from a container to a host-side data service, thereby efficiently managing provisioned memory. Our approach improves the overall memory allocation by 32.6% when running all the serverless workflows simultaneously when compared to Dataflower + Truffle. Additionally, this method improves the tail latency and the mean latency of a workflow by an average of 26.3% and 21%, respectively.

***Index Terms*—Serverless computing, Resource management, Workflow orchestration, Function modeling, Container state modeling**

I. Introduction

Serverless computing is a popular cloud computing paradigm that shifts the responsibility of deployment, autoscaling, load balancing, and fault tolerance from developers to cloud platforms [1]–[6]. Popular cloud platforms, such as Amazon Lambda [7], Microsoft Azure Functions [8], Google Cloud Functions [9], and IBM Cloud Functions [10], offer serverless computing services. It is well-suited for applications that require high scalability and are event-driven, such as IoT, data analytics, machine learning, and web applications [11]–[15]. Furthermore, serverless platforms provide pay-as-you-go billing models that charge a user only for the function execution duration. These advantages make the serverless paradigm cost-effective, especially when the application's demand is variable or unpredictable.

In a typical serverless platform, users submit their applications as a set of functions along with their invocation patterns. Upon receiving a function request, the platform's workflow engine spawns a container, sends a signal to the container for execution, and receives the response from the container. Since containers are ephemeral and stateless in nature, functions use a remote storage system for data exchanges. To ensure elasticity and meet the service level objectives (SLOs) of functions, typical platforms often over-provision resources – by maintaining warm containers or rapidly spawning new ones during traffic bursts. A recent Microsoft study has shown that nearly 50% of serverless functions exhibit significant variation in inter-arrival times [16]. As a result, containers that are spawned during traffic bursts will remain idle in the system, resulting in inefficient resource utilization. Therefore, the problem of resource underutilization in a serverless platform has become a critical concern.

To address the resource underutilization issue, prior work has proposed intelligent resource scheduling mechanisms that aim to allocate the desired number of containers to a function to meet their SLOs [17]–[21]. Furthermore, prior work also focused on designing lightweight virtualization mechanisms and inter-function container sharing mechanisms to avoid the necessity of maintaining idle containers in the system [22]–[25]. These aforementioned mechanisms have matured significantly over time, and the room for improvement over these works is minimal. The recent work on serverless platforms now focuses on remodeling functions' execution within a container to improve the overall system's resource utilization.

Serverless functions usually follow a five-stage lifecycle during execution: initialization, data extraction, data transformation, data storage, and termination. In our experiments with a standard input size, we observe that the extract and storage stages consume at least 18.11% of the workflow's execution time. This expense is primarily due to the necessity of retrieving input files and storing output data in a remote storage system. These stages primarily utilize network resources and minimal memory bandwidth, as they perform I/O operations. During these stages, the provisioned memory for the container is primarily underutilized. To resolve this memory underutilization, prior work has proposed the following mechanisms: prefetching and asynchronous data upload [2], [26], [27].

In prefetching [26], we offload the data extraction stage to a dedicated service, and the function resumes its execution after the data is available, thereby minimizing the duration for which a container needs to be alive. However, we observe that

Abhisek Panda is with the Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi, India, email: abhisek.panda@cse.iitd.ac.in

Smruti R. Sarangi is with the Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi, India, email: srsarangi@cse.iitd.ac.in

the prefetched data is accessed by a container after a delay in our experiments, which leads to an increase in the waiting time for a request. Conversely, in the asynchronous data upload mechanism [2], [27], we offload the data storage stage to a separate thread running in a container, thereby enabling multiplexing of requests within the container. However, when the rate of data processing exceeds the upload rate, the asynchronous data upload mechanism suffers from higher response latency. To resolve this issue, Dataflower [27] proposed a pressure-aware function scaling mechanism that increases the number of containers. While this approach aims to improve performance, it can lead to a higher memory pressure in the system.

In this paper, we propose Styx, a novel workflow engine that addresses the aforementioned limitations by enabling smart decoupling of the compute and I/O stages based on predicted data availability. To ensure that prefetched data is available precisely when needed, Styx employs a fetch latency predictor that leverages real-time system metrics from both the serverless node and the remote storage server to estimate the data fetch latency accurately. To further improve memory utilization, especially when data processing speeds exceed upload threads' throughput, Styx offloads the data storage stage to a separate host-side service. This service continues to upload output data even after the container has terminated, thereby reducing idle containers' occupancy in the system.

To summarize, our contributions are as follows:

1) We conduct an in-depth analysis of resource utilization in a serverless platform to identify the stages where resources remain underutilized.
2) Styx uses a file fetch latency predictor to ensure that the data is available precisely when needed, thereby reducing request waiting time.
3) Styx smartly offloads the data storage stage from a container to a host-side service, thereby minimizing memory wastage.
4) When executing all workflows concurrently, Styx improves the memory allocation of the system by 32.6% and reduces the tail latency and the mean latency of workflows by 26.3% and 21% on average, respectively.

We organize the rest of the paper as follows. We provide the relevant background on serverless platforms and object storage systems in Section II. Subsequently, we perform a comprehensive analysis of the resource utilization of a state-of-the-art serverless platform that utilizes prefetching and the asynchronous data upload mechanism in Section III. Following this, we discuss the design of Styx in Section IV. Section V evaluates the resource utilization and latency of a workflow running on Styx compared to state-of-the-art solutions. We discuss the related work in Section VI, followed by the scope of future work in Section VII, and finally conclude in Section VIII.

II. Background

A. Serverless Computing

The serverless computing paradigm primarily aims to provide autoscaling and a pay-as-you-go billing model. Unlike the microservice-based paradigm, a serverless platform represents an application as a series of functions in the form of a directed acyclic graph, referred to as a serverless workflow. Upon receiving a request, the platform executes these functions in a sandbox. The platform scales the number of containers assigned for executing a function based on the arrival rate of requests, thereby providing autoscaling. Moreover, the platform implements a pay-as-you-go billing model by charging a user only for the functions she has executed.

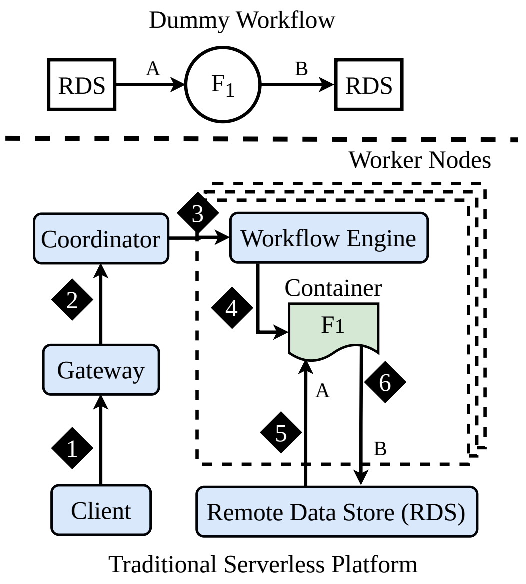


Fig. 1: A high-level overview of a traditional serverless platform executing a dummy workflow.

A serverless platform primarily consists of three components: a gateway, a coordinator, and a set of worker nodes (see Figure 1). First, the gateway receives the request to execute an application from a client (see ①). Subsequently, it routes the request to a coordinator after authentication and also ensures that a client does not exceed the specified rate limit (see ②). The coordinator employs a resource scheduling mechanism to identify a suitable worker node for executing the serverless workflow functions. Subsequently, the coordinator signals the workflow engine of a worker node to execute the function (see ③). The workflow engine then provisions a container with a predefined memory limit and initiates the function's execution (see ④). Upon completion, the response is sent to the workflow engine that either initiates the succeeding function on the same node or signals the coordinator to start it on a different worker node. In case a function needs to transfer data, it loads the output data to a remote data store (RDS), and the succeeding functions extract the data from the RDS (see ⑤ and ⑥ of Figure 1).

1) Model of a Function's Execution: From the perspective of a workflow engine, the operations performed within a container after its creation are divided into five stages: initialization (**I**), data extraction (**E**), data transformation (**T**), data storing (**S**), and termination (**R**) (see Figure 2). The I stage involves receiving a signal from the workflow engine, and initializing variables or objects. In the E stage, the container extracts inputs or intermediate data required by the application, which it processes in the T stage. After processing, the container stores the output or intermediate data on a remote storage server during the S stage. Finally, in the R stage, the container terminates the function and sends the response back to the workflow engine.

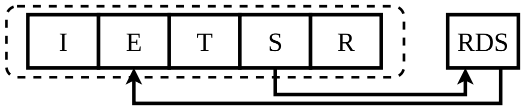


Fig. 2: A granular view of a function's execution within a container.

B. Container's Resource Utilization

A serverless platform runs functions inside containers along with enforcing predefined memory limits on containers to prevent excessive resource consumption. Prior work and our experiments indicate that a function instance primarily utilizes CPU and memory resources during the I, T and R stages [27]. In contrast, the instance relies mainly on network resources while consuming minimal memory (up to 50 MB) when interacting with an RDS during the E and S stages [27]. This imbalance highlights the significant underutilization of container resources. To improve resource utilization, prior work proposed cache mechanisms to store intermediate results generated by a function, thereby avoiding costly RDS accesses. However, there are functions that still need to make RDS accesses for data requiring long-term storage requirements. In the case of these functions, prior work has introduced two key optimizations to improve resource utilization: prefetcher and asynchronous data uploads. Let us elaborate.

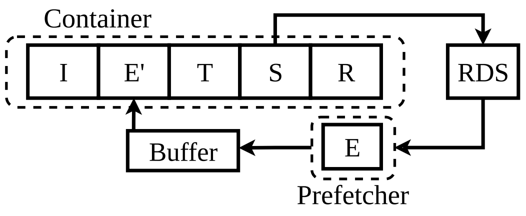


Fig. 3: A granular view of a function's execution with a prefetcher.

1) Prefetcher: In this mechanism, we offload the E stage from a container to a prefetcher (see Figure 3). When a function request arrives, the coordinator selects a worker node, and simultaneously signals its workflow engine and a prefetcher. The workflow engine provisions a container to execute the function. In parallel, the prefetcher parses the HTTP request body to determine required input files, retrieves them, and stores them in a buffer.

After the completion of input data fetching, the workflow engine sends a signal to the container to start function execution. During execution, the function accesses the prefetched data directly from the buffer. As a result, the resource utilization of the system improves by reducing the duration for which the container resources are utilized.

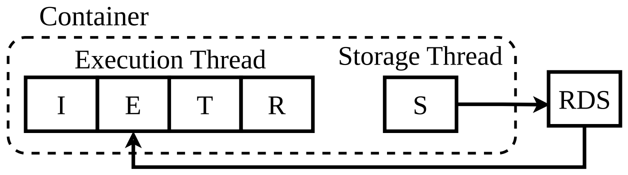


Fig. 4: A granular view of a function's execution with the asynchronous data upload mechanism.

2) Asynchronous Data Uploads: In this mechanism, we offload the S stage to a separate thread running within the same container (see Figure 4). When a function request arrives, the workflow engine schedules it on a container. Inside the container, the execution thread carries out the function's execution. Once execution is complete, it signals the storage thread to store the output data in either the RDS or the cache, and starts executing the next function request. This mechanism enables the container to handle two operations concurrently – execution and storage – thereby enhancing the overall throughput and improving the overall resource utilization.

3) Model of a Container's State: During the execution of a function by a platform with the aforementioned optimizations, a container concurrently performs two tasks – execution (EX) and storage (S). This design leads to three possible states of a container: *EX active*, *S active* and *EX+S active*. The EX active state signifies that only the execution thread is running, while the S active state indicates that only the storage thread is running. The EX+S active state represents a scenario where both the execution and storage threads are running concurrently. The EX active and EX+S active states utilize the provisioned memory resources. Whereas, the S active state primarily utilizes the network resources and a small portion of memory resources (up to 50 MB). The S active state means that the function running inside the container has finished its computation and generated its final outputs. Despite this in default implementations, the container continues to retain memory allocated to the function runtime, libraries, and intermediate data until the function's execution terminates. As a result, memory remains unnecessarily occupied, leading to inefficient resource utilization, as these resources could otherwise be reclaimed and used to run additional containers.

C. Object Storage System

Since function containers are ephemeral in nature, serverless platforms use external object storage systems, such as MinIO [29], Ceph [30], Amazon S3 [31], Azure Blob Storage [32] and Google Cloud Storage [33] to manage the data that needs long-term requirements. These storage systems mainly support two operations: `put` (upload) and `get` (download). In this paper, we use the open-source object storage system MinIO due to its compatibility with Amazon S3, seamless integration with Kubernetes, and widespread use in prior work [34]–[38].

TABLE I: The workflows used in this paper are adopted from real-world serverless applications [11]–[15], [28]. *Note: we allocate containers of memory limit 256 MB to the face_detector, ml_normalize and ml_merge functions, while others have a memory limit of 128 MB.*

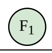
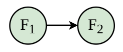
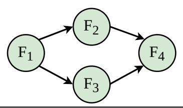
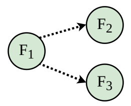
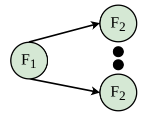
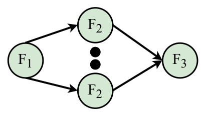
Workflow	**Description**	**Functions**	**Structure**	**Input Size**
compress	reads a file from MinIO, compresses the file, and uploads it back to MinIO.	$F_1 : compress$		4.1 MB
retail	uploads a product's catalog information into CouchDB, and then uploads the product's pictures into MinIO.	$F_1 : load\_catalog$, $F_2 : load\_pictures$		4.9 MB
iot	filters the anomalies, computes the aggregate metrics of the filtered data, and detects trends in the filtered data parallely, and finally uploads analytics data to MinIO.	$F_1 : anomaly\_filter$, $F_2 : data\_analyzer$, $F_3 : trend\_detector$, $F_4 : upload\_data$		7.8 MB
healthcare	processes a patient's consent form to determine how their health record has to be handled. If consent is granted, the anonymized record is uploaded to MinIO; otherwise, the data is encrypted before uploading to MinIO.	$F_1 : parse\_consent$, $F_2 : data\_encrypter$, $F_3 : data\_masker$		4.1 MB
detector	retrieves a video file from MinIO, extracts its frames, performs face detection on each frame, and uploads frames with faces back to MinIO. Note: the face detection model is already included in the function image, and the function fetches only the video from an RDS during the execution.	$F_1 : video\_decoder$, $F_2 : face\_detector$		100 KB
ml	retrieves a dataset from MinIO and partitions it into smaller shards for parallel processing, applies min–max feature normalization to each shard independently, and finally merges all normalized shards into a single preprocessed dataset before uploading it back to MinIO.	$F_1 : ml\_split$, $F_2 : ml\_normalize$, $F_3 : ml\_merge$		5 MB

TABLE II: System configuration

Hardware settings			
Processor	Intel Xeon 6226R CPU, 2.90 GHz		
CPUs	1 Socket, 16 cores	DRAM	256 GB
Software settings			
Linux kernel	v5.15.0-56	Redis version	v7.4.1
MinIO version	R2025-01-20T14-49-07Z	Docker version	v24.0.2

In MinIO, the `put` operation involves the following steps: ① authenticates a request, ② extracts metadata from the request header, ③ splits data into blocks of 1 MB each, ④ applies the Reed-Solomon erasure coding scheme to encode the data block, ⑤ writes data blocks to the selected drives, ⑥ executes the *fdatasync* function, ⑦ creates a metadata file for the object containing checksums, entity tag, and content type, and ⑧ sends the object's metadata along with the status code as a response. In contrast, the `get` operation follows these steps: ① authenticates the request, ② retrieves the object's metadata, ③ retrieves and reassembles the object's data shards, ④ verifies the reassembled object's integrity, and ⑤ sends the data along with the response code.

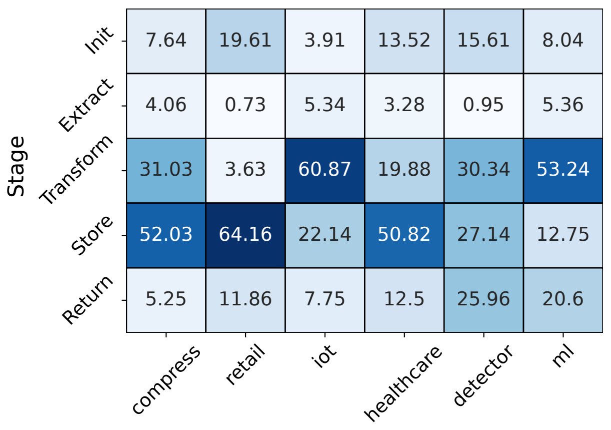


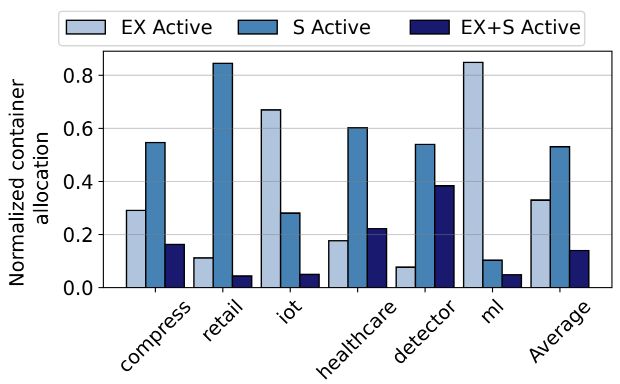
Fig. 5: The percentage contribution of the init, extract, transform, store, and return (termination) stages to the execution latency of workflows listed in Table I.

III. Characterization of a Serverless Platform

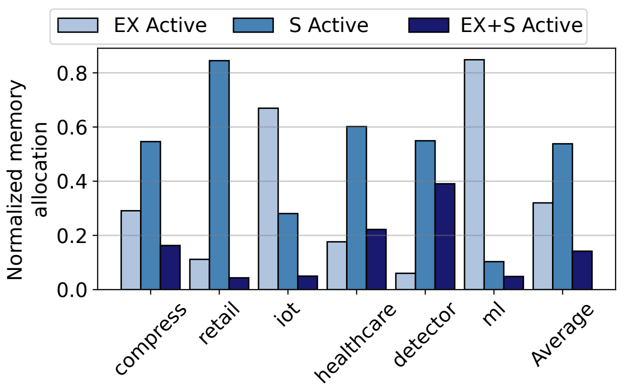
In this section, we evaluate the resource utilization of a serverless platform and the breakup of workflows' execution latency when serving real-world serverless applications.

A. Evaluation Methodology

We evaluate the resource utilization of a state-of-the-art serverless platform equipped with a prefetcher (Truffle [26]) and the asynchronous upload mechanism (Dataflower [27]), on a single server node. This approach aligns with our goal of designing an efficient workflow engine that improves resource utilization of a node, and enabling its replication across multiple nodes in a cluster. To ensure diverse evaluation, we use a range of serverless workflows adapted from real-world applications [11]–[15], [28], as listed in Table I. These workflows have diverse kinds of branching, parallelism and fan-out patterns. They also have varying lengths. The system configuration is detailed in Table II. For characterization, we randomly pick input files of sizes ranging between $1\,KB$ and $10\,MB$. This is because 70% of serverless functions process



(a) Container Allocation



(b) Memory Allocation

Fig. 6: The resource utilization of the platform when containers are in the EX active, S active, and EX+S active states during the execution of workflows listed in Table I normalized to the overall resource utilization of the platform across all container states.

data less than 10 MB [39]. In our setup, we deploy MinIO [29] as an RDS (similar to [36]–[38]) on a dedicated node isolated from application workloads (similar to [40], [41]), while Redis serves as a local data cache (similar to [2], [27]). Note that we evaluate each of these workflows with random input files of varying sizes in Section V.

We assume that the inter-arrival times of workflow requests follow a Poisson distribution [4]. To maintain a balanced load on the framework running on our system, we identify the maximum arrival rate for a workflow. In line with prior work, we execute each workflow with increasing request arrival rates and monitor the response latency [4], [42], [43]. As the arrival rate increases, the system eventually reaches a point where the requests starts dropping, indicating that the system's capacity has been reached. We identify this point in the latency–load curve as the maximum arrival rate that the system can sustain for that workflow.

B. Metrics for Resource Utilization

During workflow execution, a serverless platform dynamically provisions containers based on the incoming request rate, with each container reserving a predefined memory quota on the node. To systematically assess host-level resource utilization, we track the number of active containers serving requests and the corresponding memory reserved by the host over time. To quantify resource utilization, we define two key metrics: *container allocation* and *memory allocation*. To compute these metrics, we first generate time-series curves that capture their respective values throughout the workflow's execution. Subsequently, we derive the allocation values by computing the area under these curves, thereby providing a comprehensive measure of resource utilization.

C. Function Level Breakdown

From the perspective of the workflow engine, we divided the function execution into 5 stages: init (I), extract (E), transform (T), store (S), and termination (R) (refer to Section II-A1). We first study the contribution of these stages post-container creation to the overall execution latency of a serverless workflow. For this study, we execute a workflow once and collect the execution time of the aforementioned stages.

To measure the execution time of the aforementioned stages within a workflow, we instrumented the functions' code. Since we have access to workflows' source code, we first identified the stages within each function and inserted timestamp markers at their *entry* and *exit* points. Each function records the stage name, function name, and corresponding start and end timestamps, which are appended to the function's response object. As the workflow progresses, subsequent functions propagate and extend this information. When the sink function completes, the final response object contains a complete timeline of all stages across the workflow, which we analyze offline to compute per-stage execution times of a workflow. *Note: This instrumentation in the function code is used only for profiling purposes and is not required for the normal operation of Styx.*

In Figure 5, we show the distribution of overall execution latency across different stages, with an average breakdown as follows: 12.13% for I, 3.07% for E, 29.86% for T, 40.29% for S, and 14.65% for R. The E stage exhibits the lowest contribution due to our prefetching mechanism, which retrieves input data from an RDS before workflow execution and caches it for a function's access. After the E stage, we have the I and R stages that primarily handle function initialization, termination, and communication between a container and a workflow engine. The T stage contributes less than the S stage because smaller input sizes were provided to workflows.

We observe that the amount of time a workflow's request spends in the S stage is in the range of 12.75% to 64.16%. In the S stage, a workflow's function might store data into a local data cache or a remote storage system. We observe that functions at the sink of a workflow significantly contribute to the S stage due to remote storage accesses. The primary reason for this is the `fdatasync` operation of MinIO, which accounts for up to 40% of the upload cost. This operation flushes file data to disk, thereby ensuring data durability and consistency across object upload operations. In the *retail* workflow, this contribution is the highest (= 64.16%) because it involves uploading images to MinIO, whose size is 4.9 MB.

1) A workflow's request spends about 12.75% to 64.16% in the store stage for the workflows listed in Table I. This is because the functions at the sink perform remote storage accesses.
2) The storage stage's contribution is higher due to the `fdatasync` operation.

D. System Level Breakdown

From the perspective of resource usage, we divided a container into 3 states: *EX active*, *S active* and *EX+S active* (discussed in Section II-B3). In this section, we examine the resource utilization of containers across these states. For this analysis, we execute a workflow at 80% of its maximum arrival rate while tracking the number of containers over time along with their respective states and memory allocations (similar to [4], [42], [43]).

In Figure 6a, we observe that 32.97%, 53.05% and 13.98% of the overall container allocation across all container states is utilized by containers in the EX active state, the S active state and the EX+S active state, respectively. On the other hand, we observe that 31.85%, 54.48% and 13.67% of the overall memory allocation across all container states is utilized by containers in the EX active state, the S active state and the EX+S active state, respectively (see Figure 6b). For the *retail* workflow, we observe that the majority of resources are allocated for containers in the S active state due to the workload's characteristics. During the S active state, a container is performing only the upload operation that primarily requires network resources and a small portion of memory resources (up to 50 MB). Therefore, we conclude that 54.48% of the overall memory allocated by a platform is utilized inefficiently.

For the workflows outlined in Table I, 54.48% of the memory allocated by a platform is utilized by containers in the S active state, thereby leading to inefficient utilization of memory resources.

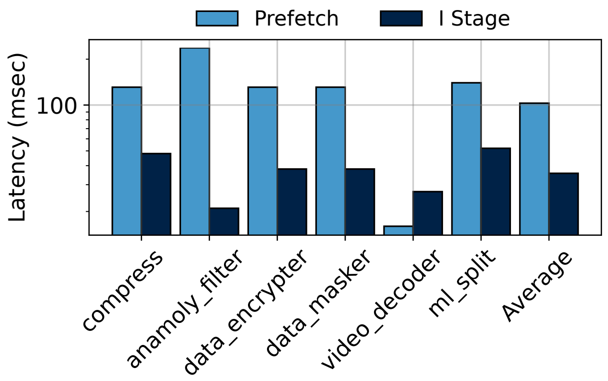


Fig. 7: The prefetch latency and the time spent in the I stage by the source functions of workflows listed in Table I that read input data from a remote storage.

E. Scope of Overlapping Prefetching and Initialization

When a function's request is received, the workflow engine instructs the prefetcher to retrieve the required input files (the E stage), which takes p seconds. Once prefetching is complete, the request is sent to a container that starts accessing the data after completing the I stage, which takes i seconds. In this section, we analyze the relationship between i and p to evaluate the potential performance improvements from overlapping these operations.

In Figure 7, we show that a function request spends approximately 102.85 msec in the prefetcher, while the container waits around 35.62 msec to access the prefetched file. In *video_decoder*, p is smaller than i due to the small 100 KB file size, while p exceeds i in *anomaly_filter* as it involves downloading a larger 7.8 MB file. By overlapping the initialization stage and data prefetching, we can reduce the waiting time of a request. However, if p is greater than i, the container will remain idle, thereby wasting resources. To optimize resource utilization, we need an intelligent mechanism that can determine the correct values of p and i. Subsequently, we can trigger the function request after $p - i$ seconds.

Overlapping the initialization stage and prefetching minimizes request waiting time. However, an intelligent mechanism is needed to trigger execution at the optimal moment, preventing the container from being idle while waiting for data.

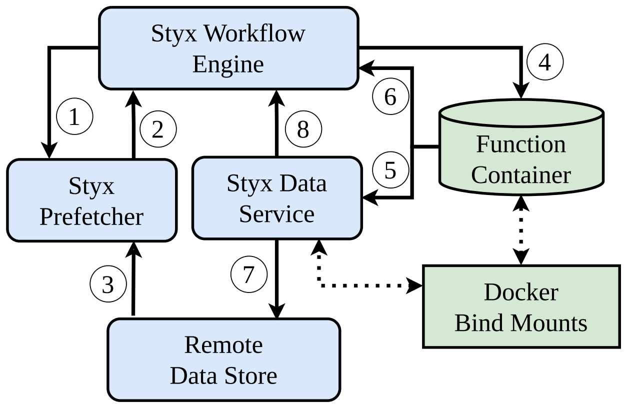


Fig. 8: A high-level overview of Styx.

IV. Design

In this section, we discuss the design of Styx, a novel intra-node workflow engine that aims to reduce the memory allocation on a node and the waiting time of requests.

A. Design Principles

In Section III, we observe that the majority of resources are allocated to containers in the S active state, leading to inefficient utilization of memory. This occurs because the data upload thread retains a container while storing data on an RDS, even after the execution has been completed. We also observe a noticeable delay between the prefetching of data and its access, which increases the waiting time for a request. These insights lead to the following design decisions when handling a function that performs remote data accesses:

1) The I and E stages of a function should be synchronized to ensure that data becomes available precisely when needed – neither too early nor too late.
2) Once the T stage completes, the S stage can be offloaded to a dedicated data service running on the host that multiplexes the S stage of multiple requests, enhancing overall resource utilization.
3) The function generates a soft acknowledgment after the T stage, signaling that the container is ready to handle new requests. Meanwhile, the data service generates a hard acknowledgment upon completion of the S stage, signaling that the request is completed.

B. Styx Components

Figure 8 illustrates the high-level architecture of Styx, which consists of three core components: the Styx prefetcher, the Styx workflow engine and the Styx data service. The Styx prefetcher first predicts the fetch latency of an object and sends this estimate to the Styx workflow engine. It then fetches the object into an in-memory data store from which the function later reads. Based on the estimated time it will take to fetch the object, the Styx workflow engine makes sure the object is ready exactly when the function needs it, allowing the initialization (I) and extract (E) stages to finish at the same time. After the transform (T) stage completes, the container signals the Styx data service to initiate the storage (S) stage if an RDS is involved. Simultaneously, it sends a soft acknowledgment to the Styx workflow engine to trigger the execution of the next request, allowing containers to focus solely on computation. Upon completion of the S stage, the Styx data service sends a hard acknowledgment to the workflow engine, marking the completion of the request's execution. The workflow engine subsequently triggers downstream dependents in the workflow only upon receiving this hard acknowledgment. In the following sections, we discuss the working of Styx in detail.

C. Function Registration

When a user registers an application on a serverless platform, they first decompose it into individual functions. Subsequently, they create a workflow definition file that represents the data flow between these functions. Each entry in the definition file typically includes the following attributes: $func\_name$, $type$, $input\_data$, and $output\_data$. The $func\_name$ attribute denotes the name of the function. The $type$ attribute specifies whether the function is a conditional branch or a normal function. The $input\_data$ and $output\_data$ attributes define the data consumed and the data produced by the function, respectively. We extend these attributes to enable the prefetching of input data, and coincide the end times of the I and E stages.

1) Attributes for Data Prefetching: To enable input data prefetching for a function, Styx extends the $input\_data$ attribute of functions in the definition file. For each entry of $input\_data$, we add the following attributes: $source$ and $param$. The source denotes if the data is available in an RDS or a cache. The param holds a list of HTTP request body fields that must be parsed to extract essential information, such as the access token, bucket name and object name. Note that the definition file is part of the serverless platform, which is a trusted entity; therefore, sharing the access token for the object store is not a security violation [26].

TABLE III: The features and their respective data types collected from the node and the remote storage system for use by the fetch latency predictor.

Features	Data type	Features	Data type
cpu_util	4-byte float	#active_upload_reqs	4-byte unsigned int
net_util	4-byte float	#active_download_reqs	4-byte unsigned int
disk_util	4-byte float	#active_MinIO_reqs	4-byte unsigned int
mem_util	4-byte float	filesize (MB)	4-byte unsigned int

2) Attributes for End Time Coinciding: To enable the coinciding of the end times of the I and E stages of a function, we extend the entry of a workflow definition file with an additional attribute $init\_insts$. This attribute represents the number of instructions executed during the I stage of the function, which is input size independent. The developer provides this value during the function registration process. The platform first profiles the function initialization stage to derive its instructions-per-cycle (IPC) value. It then derives the time taken by the I stage of a function ($init\_time$) using Equation 1. Therefore, the platform can derive the $init\_time$ regardless of the system configuration. The Styx workflow engine utilizes the $init\_time$ value to decide at which point in time a function is needed to be triggered.

$$init\_time = \frac{init\_insts}{IPC \times CPUfrequency} \tag{1}$$

D. Function Request

In a serverless platform, users initiate function execution by sending an HTTP request containing input parameters and their values [44]–[46]. After authentication, the platform assigns the request to the workflow engine of a suitable node. The Styx workflow engine maps the request to a container for execution. It analyzes both the request body and the workflow definition file to identify any required input files from a remote data store (RDS). If such files are needed, the engine signals the Styx prefetcher via a TCP socket to initiate the fetch process (see ① in Figure 8).

E. Function Data Extraction

The Styx prefetcher receives a request from the workflow engine. It predicts the fetch latencies of input files and sends them to the Styx workflow engine. Predicting the fetch latency of an input file is a challenging task due to the influence of numerous dynamic platform-wide factors beyond just node-specific attributes. The node typically lacks visibility into key variables such as the size of the object to be fetched and system conditions at the remote storage server. As a result, the design of an accurate and reliable fetch latency predictor is significantly complex and nontrivial. Subsequently, the Styx prefetcher starts fetching input files to an in-memory buffer. Let us elaborate.

1) Input Features of the Predictor: To obtain a comprehensive platform-wide view, the predictor collects resource statistics from both the node and the remote storage server, along with the input file size in megabytes ($filesize\ (MB)$). On the node side, it gathers the following features: the number of active threads handling upload and download operations to MinIO ($\#active\_upload\_reqs$, $\#active\_download\_reqs$), and system metrics including the CPU utilization, the disk utilization, the memory usage, and the network utilization (cpu_util, $disk\_util$, mem_util, net_util) (refer to Table III). On the remote storage server, it collects the number of currently executing requests in MinIO ($\#active\_MinIO\_reqs$), and the same set of system metrics (refer to Table III). To get these features on the node, we modify the `get_stat_object` API handler of MinIO. Before predicting the fetch latency, the Styx prefetcher invokes the `get_stat_object` API to get the input file's size and the system state of the remote storage server. It then augments these features with the node-specific features and sends the complete input feature set to the predictor.

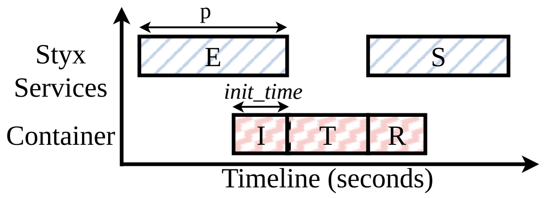


Fig. 9: The fine granular view of function execution in Styx.

2) Design of the Predictor: To predict the fetch time of a file, we use an XGBoost-based analytical model trained on the input feature set described in the previous section. XGBoost, being a tree-based ensemble method, effectively captures complex nonlinear relationships and adapts well to varying data distributions – allowing it to outperform other analytical models on our synthetic dataset (refer to Section V-A1). Once trained, this model is used by the Styx prefetcher to predict the fetch time (p) of a given file. The predicted value is then forwarded to the Styx workflow engine (see ②). Subsequently, the prefetcher initiates file download operations.

3) Input File Downloader: The input file downloader is responsible for fetching files from the RDS. Initially, it parses the HTTP request body to obtain the list of required input files for download. It then maps the request fields with the workflow definition file to find the bucket name, access token, and object names of the files. Finally, the downloader starts actual data transfer from the remote data store (RDS) to an in-memory buffer (see ③).

4) Remote Storage Feature Collector: The remote storage feature collector is responsible for collecting the remote-side features that is necessary for the fetch latency predictor. To prevent interference with storage performance, the feature collector operates outside of the API request's critical path. A dedicated background thread periodically collects remote-side features and maintains them in an in-memory global state, independent of incoming requests. When the predictor queries the `get_stat_object` API handler of MinIO API, the handler reads the global state and appends them to the response, incurring only a few microseconds of overhead. As a result, the overhead of a prediction query remains limited to lightweight global data accesses and does not prohibitively increase with the storage load..

F. Function Initialization and Execution

After receiving the fetch time of input files (p) from the prefetcher, the Styx workflow engine delays function initialization by $p - init\_time$ seconds, then starts executing the function (see ④ in Figure 8 and Figure 9). This ensures that the data becomes available precisely when it is needed by the function, thereby reducing the waiting time of a request. After the T stage completion, the engine receives a soft acknowledgment from the function's container (see ⑥). Subsequently, it schedules the next request on the container. Simultaneously, the Styx data service receives a signal from the function to start the S stage of a function's request through a TCP socket (see ⑤ in Figure 8).

G. Function Data Storage

The Styx data service handles the upload of output data from a container to the remote data store (RDS) (see ⑦ in Figure 8). Since the output is generated inside the container, a key challenge is enabling effective file sharing between the container and the Styx data service running on the host. This shared data must persist even if the container is stopped or removed. Docker provides two mechanisms to support persistent and accessible storage: volumes and bind mounts. Volumes are managed by Docker, whereas bind mounts give direct access to files on the host system. Given these characteristics, Docker bind mounts are well-suited for sharing data between the container and the host [47].

From a security standpoint, Docker bind mounts grant the container access to part of the host filesystem. To ensure isolation, Styx assigns a unique, initially empty host directory, $cntr\_dir$, to each container, which is mounted exclusively within the container at the time of creation. The bind mount exists only within the container's isolated mount namespace, preventing access to other host paths or mount points belonging to different containers. In addition, containers run with a non-root UID and a minimized capability set, ensuring they cannot perform privileged filesystem operations such as mounting, pivoting the root filesystem, or traversing beyond the bind-mounted directory. To enforce safe file access semantics within $cntr\_dir$, the host-side data service constrains all file I/O to the container's assigned directory. Specifically, it disables symlink traversal (e.g., via no-follow semantics) and validates canonical paths to ensure that all resolved paths remain within the $cntr\_dir$ boundary.

To facilitate data communication, the Styx workflow engine instructs Docker to mount $cntr\_dir$ on the container during the container creation. Upon receiving a signal from the container, the data service reads data directly from the container's $cntr\_dir$ and uploads it to the RDS following at-least-once delivery semantics. Upon successfully completing the S stage, the Styx data service sends a hard acknowledgment to the workflow engine marking the request as complete and allowing the execution of workflows' downstream functions (see ⑧ in Figure 8). In the case of a failure, the data service treats the data as uncommitted and notifies the workflow engine of the failure. The workflow engine then re-executes the function. Similarly, if a node failure occurs after a soft acknowledgment, the absence of a corresponding hard acknowledgment prevents downstream functions from being triggered.

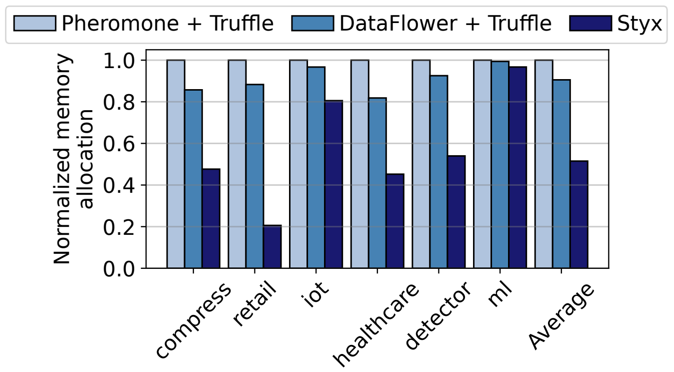


Fig. 10: The overall memory allocated by a serverless platform while executing the serverless workflows listed in Table I. Takeaway: Styx improves the memory allocation by 43.1%.

V. Evaluation

In this section, we discuss the effectiveness of Styx by executing real-world serverless workflows. We evaluate its performance by comparing the memory allocation, the mean latency and the tail latency against two state-of-the-art serverless platforms: Dataflower [27] and Pheromone [1], both integrated with Truffle [26]. For a fair comparison, the memory allocation in Styx includes the memory consumed by the Styx data service and containers. Furthermore, we measure the latency of a workflow in Styx up to the point at which the workflow engine receives the hard acknowledgment from the sink function of a workflow.

To account for the memory footprint of the Styx data service in the memory allocation metric, we record its memory usage as a time series throughout the workflow execution using a 10 ms sampling interval. In particular, we measure the resident set size (RSS) of the data service process, which represents the portion of memory resident in physical RAM (similar to [48], [49]). At each sampling point, we add the data service's RSS into the total memory allocated to active containers on the host, resulting in a unified time series that reflects platform-level memory allocation. We then compute the area under this combined curve to derive the final memory allocation metric.

A. Evaluation Setup

Our serverless setup has been described in Table II (see Section III). To host the MinIO and Apache CouchDB, we use a machine with an Intel Core i5-4590 CPU, 4 CPU cores, a 1 TB Seagate hard disk, and 12 GB of memory. The average ping latency between the node and the registry is 214 μsec. To evaluate our scheme, we use a set of real-world serverless applications with diverse workflow structures described in Table I. For each of these workflows, we randomly pick an input file out of 50 choices following a normal distribution (similar to Beldi [50]). The input file size ranges between 1 KB and 10 MB because 70% of serverless function process data less than 10 MB [39]. We use MinIO as a remote data store and Redis as a local cache.

To simulate real-world traffic patterns, we evaluate our system using the asynchronous invocation mode. In the asynchronous mode, we generate requests with a random input size in an open-loop fashion, with the arrival rate set to 80% of a workflow's maximum load. To evaluate the scalability of Styx, we perform a colocation experiment by executing all workflows from Table I with random input sizes simultaneously. In addition, we also evaluate the impact of a workflow's input size, inaccurate $init\_insts$ estimates and a bursty load on the evaluated metrics.

TABLE IV: Comparison of the fetch latency predictor with different models.

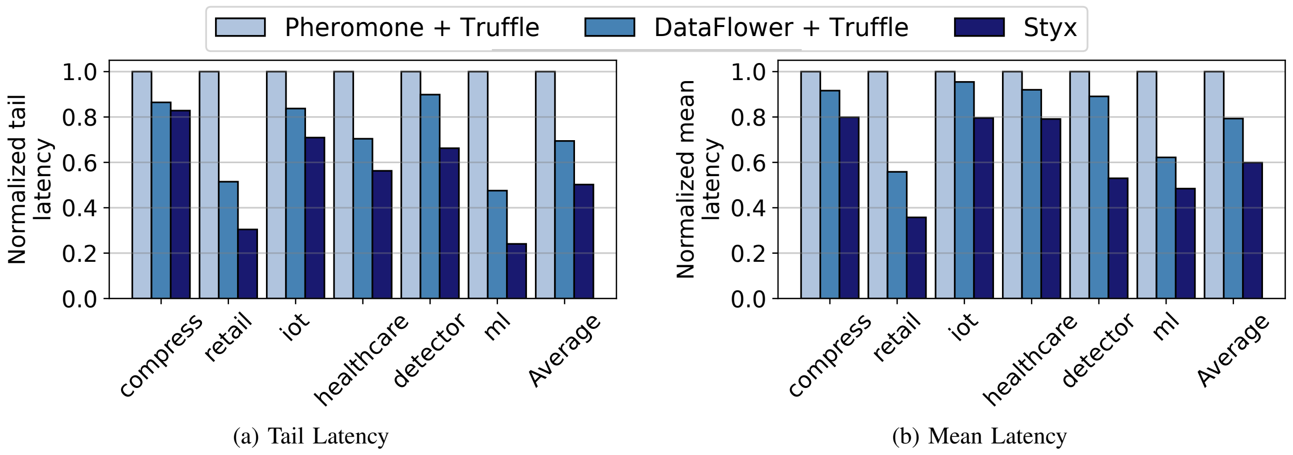
Model	MSE (sec^2)	R^2
XGBoost Regressor	0.1028	0.8942
LightGBM Regressor	0.1091	0.8889
Random Forest Regressor	0.1224	0.8759
Gradient Boosting Regressor	0.1366	0.8616
K-Nearest Neighbors Regressor	0.1897	0.8094
Decision Tree Regressor	0.1944	0.7997
Support Vector Regressor (SVR)	0.2187	0.7802
Ridge Regression	0.4621	0.5303
Linear Regression	0.4621	0.5303

1) Selection of a Model for the Fetch Latency Predictor: In this section, we find an appropriate analytical model that can accurately predict the fetch latency of a file with the features described in Section IV-E1. For this analysis, we create a synthetic dataset by running workflows with different input sizes. In Table IV, we show that the XGBoost-based model has better mean squared error and R^2 score compared to other regression models across five folds. Therefore, we use the same model in Styx. Note that the input sizes used for generating the synthetic dataset are different than those used for evaluation.

2) Hyperparameter Setting of the Fetch Latency Predictor: The Styx prefetcher uses an XGBoost-based analytical model to predict the fetch latency of input files. To optimize its performance, we apply one-at-a-time (OAT) hyperparameter tuning, where each hyperparameter is adjusted individually while keeping others fixed to assess its effect on model accuracy [4]. Using this method, we determine the optimal values for the following hyperparameters: $n\_estimators = 200$, $learning\_rate = 0.05$, $max\_depth = 6$, $eta = 0.3$, and $subsample = 0.7$.

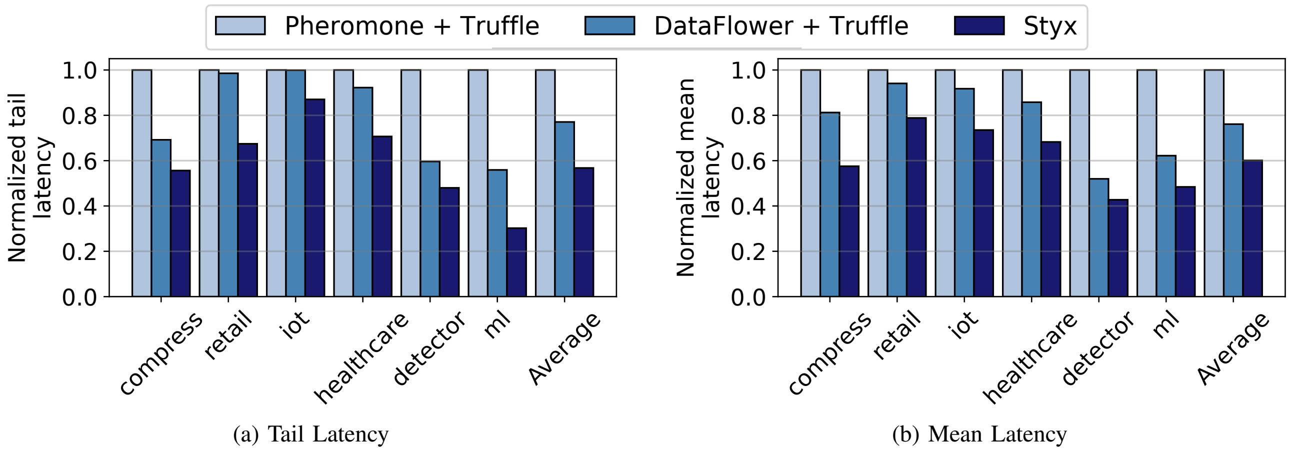
B. Asynchronous Invocations

In this section, we evaluate the workflows' performance in the asynchronous invocation mode. In Figures 10 and 11, we show that Styx reduces the total memory allocation, the tail latency, and the mean latency by 43.1%, 27.7%, and 24.5% compared to state-of-the-art schemes, respectively. The improvement in the memory allocation is because we offload



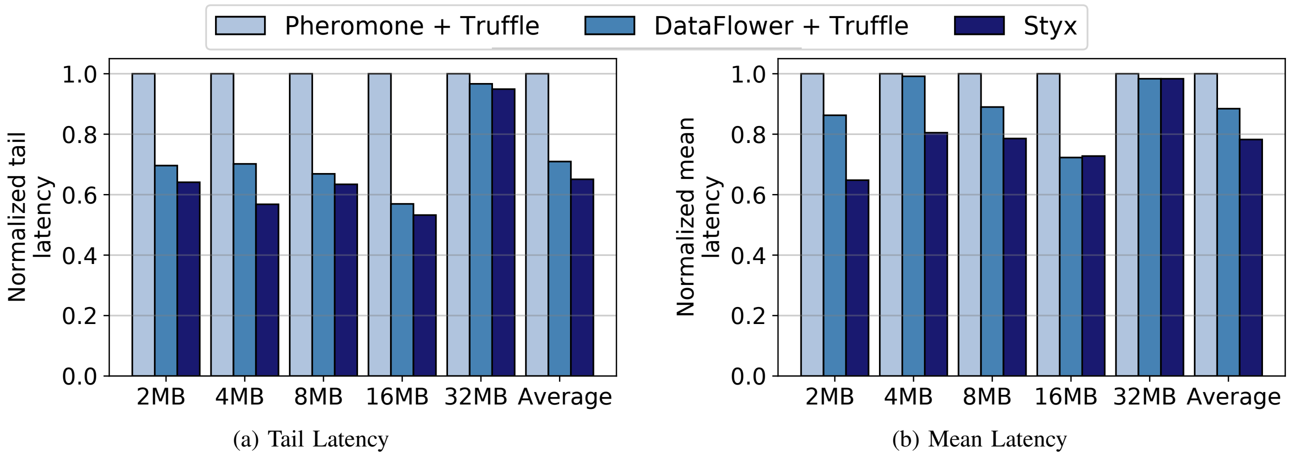
(a) Tail Latency (b) Mean Latency

Fig. 11: The tail latency and the mean latency of serverless workflows when executed on Styx and the state-of-the-art solutions (normalized to that of Pheromone + Truffle). Takeaway: Styx improves the tail latency and the mean latency of applications by 27.7% and 24.5%, respectively.



(a) Tail Latency (b) Mean Latency

Fig. 12: The tail latency and the mean latency of serverless workflows when executed concurrently on Styx and the state-of-the-art solutions (normalized to that of Pheromone + Truffle). Takeaway: Styx improves the tail latency and the mean latency of applications by 26.3% and 21%, respectively.



(a) Tail Latency (b) Mean Latency

Fig. 13: The tail latency and the mean latency of serverless workflows when the *iot* workflow executed with different input sizes on Styx and the state-of-the-art solutions (normalized to that of Pheromone + Truffle).

the data storage stage from containers to the Styx data service. This reduces the container allocation by 68.46% compared to state-of-the-art schemes. During this experiment, we observe that the Styx data service consumes about 146.6 MB, on average. The improvement in the latency of workflows is because the Styx storage service smartly overlaps the initialization stage with the data extraction stage, thereby reducing the waiting time of a request by 20.5%.

We observe that the memory allocation, the tail latency, and the mean latency of Dataflower improve by 9.5%, 30.6%

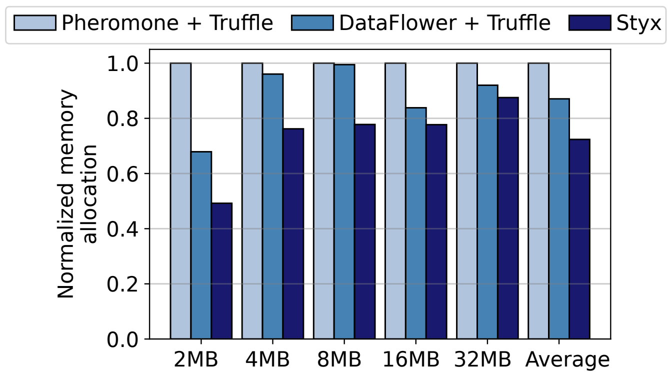


Fig. 14: The overall memory allocated by a serverless platform while executing the *iot* workflow with different input sizes

and 20.7% when compared against Pheromone, respectively. This is because Dataflower uses the asynchronous threads mechanism that enables concurrent invocation of a function's execution and data storage threads. On the other hand, Pheremone waits for the completion of the previous function request before starting to execute the next function request. As a result, the container allocation reduces by 12.7% and the waiting time of a request reduces by 22.3% in Dataflower.

Disabling *fdatasync* in the *S* Stage: In Section III, we demonstrate that the *fdatasync* operation constitutes a major contributor to the S stage latency in MinIO. To evaluate only Styx 's offloading architecture, we measure the workflows' performance in the asynchronous invocation mode with *fdatasync* disabled in MinIO, thereby eliminating storage system–specific bottlenecks in Styx and baseline systems. Disabling *fdatasync* weakens durability semantics; accordingly, this configuration serves only as a diagnostic test and does not reflect a production setting. Under this configuration, the relative contribution of the I stage to execution latency increases (to 14.4%), while the contribution of the S stage decreases (to 30.8%). Styx reduces the total memory allocation, the tail latency, and the mean latency by 22.8%, 32.1%, and 35.6% compared to state-of-the-art systems, respectively. These improvements stem from a 49.2% reduction in the container allocation and a 30.2% decrease in the request's waiting time. The increased contribution of the I stage suggests improved overlap between the E and I stages in Styx, which can help reduce response latency. Additionally, offloading the S stage to the Styx data service and handling uploads asynchronously may reduce queueing delays associated with serialized storage operations in the baseline, contributing to improvements in both mean and tail latency.

C. Colocated Invocations

In this section, we run all the workflows simultaneously. To ensure a reasonable load in the system, we execute each of the workflows with a moderate load, i.e. 10%-15% of their respective maximum load. In Figure 12, we show that Styx reduces the tail latency and the mean latency by 26.3% and 21% compared to state-of-the-art schemes, respectively. The improvement in the latency of workflows is because the Styx reduces the waiting time of a request by 19.6%. In addition, Styx reduces the total memory allocation by 32.6% due to offloading of the data storage stage to Styx data service. This offloading reduces the container allocation by 40.7%. During this experiment, we observe that the Styx storage service consumes about 227 MB, on an average.

D. Impact of an Input File's Size

In this section, we study the impact of a workflow's input size on the memory allocation and its latency. For this study, we select the *iot* workflow as a representative example since it is the largest and spends less amount of time in the storage stage. In Figure 14, we observe that as the input size increases from 2 MB to 32 MB, the improvement in the memory allocation reduces from 27.5% to 4.9%, respectively. This trend emerges because, for larger input sizes, the container spends about 76% of its time in the transform stage, limiting the potential for memory optimization. Similarly, the benefit in latency reduction diminishes with increasing input sizes (see Figure 13). This is largely due to the fact that, for higher input sizes, the file fetch latency begins to dominate and overshadows the time spent in the initialization stage of the workflow's source function.

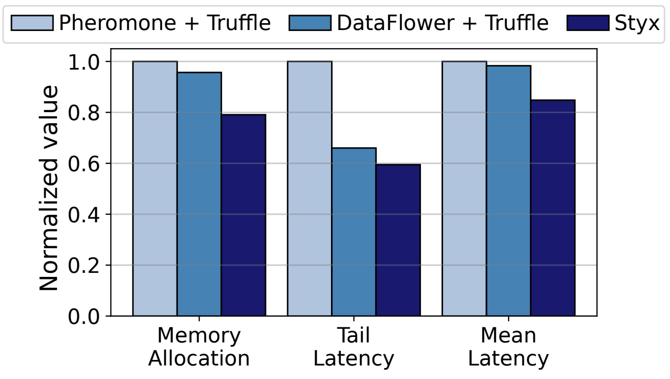


Fig. 15: The memory allocation and the latency of a serverless platform while executing the *iot* workflow with a bursty load (normalized to that of Pheromone + Truffle).

E. Impact of a Bursty Load

Given that serverless platforms are particularly well-suited for handling bursty workloads, we investigate how such sudden load spikes impact memory allocation and request latency. To analyze this, we design an experiment where a serverless workflow experiences a rapid increase in load – from 15% to 80% of the maximum arrival rate – executed in the asynchronous invocation mode (similar to Dataflower [27]). For this evaluation, we select the *iot* workflow as a representative example (discussed in Section V-D).

As shown in Figure 15, Styx shows performance improvements under this bursty load scenario. The memory allocation improves by 17.3%, the tail latency by 9.87%, and the mean latency by 13.74% compared to the state-of-the-art solutions. The observed gain in memory allocation is primarily from offloading the data storage stage from the container to a host-side service, which reduces the container allocation by 25.6%. Additionally, the improvements in both mean and tail latency can be attributed to a reduction in request waiting time by

6.8% – enabled by coinciding the end times of the function initialization and the prefetching operation.

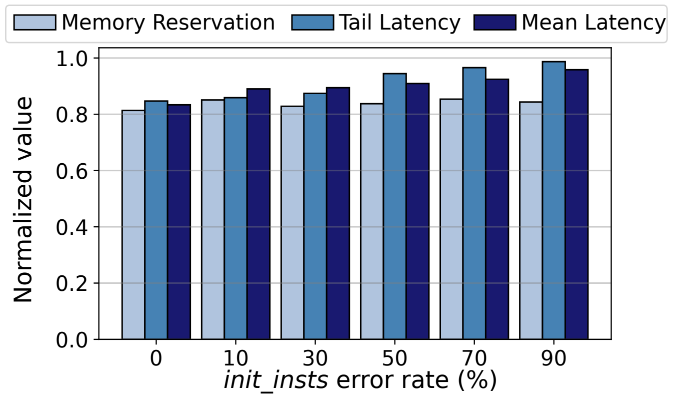


Fig. 16: The memory allocation and the latency of Styx while executing the *iot* workflow with different error rates in the $init\_insts$ estimates under the underestimation scenario (normalized to that of Dataflower + Truffle).

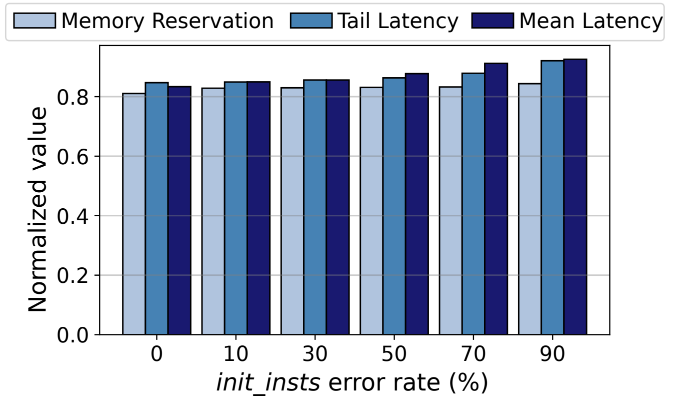


Fig. 17: The memory allocation and the latency of Styx while executing the *iot* workflow with different error rates in the $init\_insts$ estimates under the overestimation scenario (normalized to that of Dataflower + Truffle).

F. Impact of Inaccurate $init\_insts$ Estimates

Styx relies on accurate estimates of $init\_insts$ to coincide the end times of the E and I stages. Since this estimate is provided by the developer, a certain degree of inaccuracy is possible. To study the impact of such inaccuracies, we execute a workflow under varying error rates in the $init\_insts$ ranging from 0% to 90%. The error rate is defined as the relative difference between the provided and the actual $init\_insts$ value. For this study, we select the *iot* workflow as a representative example (discussed in Section V-D). We further consider two error scenarios: underestimation, where the provided $init\_insts$ is smaller than the actual value, and overestimation, where it exceeds the actual value.

1) Underestimation: As show in Figure 16, the improvement in the latency metrics diminish as the error rates increases in Styx. This is because higher error rate in the $init\_insts$ estimate prevents Styx from effectively coinciding the end times of the E and I stages. However, we observe that the improvement in the memory allocation is at least 15.5% across all error rates. When the error rate reaches 50%, tail latency and mean latency improve by 5.55% and 9.04%, respectively, compared to 15.44% and 16.62% when no error is present. These results indicate that while inaccurate $init\_insts$ estimates reduce latency improvements, the improvement in the memory allocation remains largely stable.

2) Overestimation: Figure 17 shows a similar trend, where latency improvements decrease as the error rate increases, but at a slower rate compared to underestimation. Overestimating $init\_insts$ causes the I stage to be triggered earlier, potentially leading to idle waiting until the required data becomes available; however, this does not prevent overlap entirely. As a result, memory allocation improvements remain consistently above 15.6% across all error rates. At a 50% error rate, tail and mean latency improve by 13.68% and 12.26%, respectively, compared to 15.44% and 16.62% with no estimation error. These results indicate that Styx is more tolerant to overestimation than underestimation, as early triggering of the I stage still allows partial overlap and preserves a fraction of the latency benefits.

VI. Related Work

In serverless platforms, users provide their applications as a directed acyclic graph (DAG) of functions. Platforms execute these functions inside an ephemeral and stateless container, and use remote data stores (RDSs) for data communication between functions. To improve resource utilization of platforms, prior work has explored efficient resource scheduling, container sharing, lightweight virtualization, and optimized function execution models. This paper focuses on the last category, which aims to reduce the time containers spend on reading or writing data to an RDS – since such operations mainly utilize network resources, leaving memory resources underutilized. Prior work proposed the following mechanisms to tackle this problem: efficient data communication and asynchronous data upload mechanisms.

A. Data Communication

In conventional serverless platforms, the data required for a function's execution may reside on RDSs, leading to wastage of allocated resources while fetching input data. Lambdata [51], HydroCache [52], and Cloudburst [53] proposed adding a caching layer to minimize the number of remote storage accesses. While effective, this approach requires cloud providers to manually allocate dedicated memory resources, which can result in inefficient resource utilization. OFC [2] and Faa$t [5] proposed autoscaling cache mechanisms that utilize the unused memory of function containers to store object caches. Even though these solutions decrease the remote accesses, but they still need to fetch the input data and store the output data in a remote storage machine.

Truffle [26] proposed a smart prefetching mechanism that parses the request body of a function to identify the necessary input files for execution. Subsequently, the prefetcher engine downloads these files into a local buffer, from which containers retrieve them. However, the container accesses the prefetched data after a delay, which leads to an increase in the request waiting time. To reduce this delay, Styx intelligently overlaps the I and E stages of a function such that data becomes available precisely when needed.

B. Asynchronous Data Upload

In conventional serverless platforms, containers primarily utilize either network resources for data communication or compute resources for data processing. But containers rarely use both resources at peak capacity simultaneously, resulting in underutilized compute resources. OFC [2] proposed an execution model where output data storage occurs asynchronously in a separate thread. However, as the number of requests executed per container increases, many requests may end up waiting to store their data, leading to higher response latency. In addition, functions must wait for preceding functions to complete before execution, even if their required data is already available, causing unnecessary delays. To address delays, Dataflower [27] and Pheromone [1] proposed a dataflow-based orchestrator that triggers a function execution based on its data availability. To reduce request queuing for data storage, Dataflower proposed a function scaling mechanism that increases the number of containers. However, scaling up containers increases memory pressure on the system. Moreover, using a container solely for storing results in a remote data store leads to provisioned memory underutilization.

To address the memory pressure issue and the wastage of memory resources, Styx offloads the store stage of a function from a container to the Styx data service. The service uses Docker bind mounts to read the output data of a function from the container, and then uploads the data to the RDS. This separation reduces the lifespan of a container.

VII. Future Work

A. Cluster Scale Deployment

In a multi-node cluster deployment, Styx retains its core architecture, but its components are distributed across the cluster. Each node runs a Styx workflow engine and a Styx data service locally, while the request scheduling decisions, data extraction, and prefetch predictions are managed by cluster-wide services. When a function request arrives, the coordinator first retrieves the predicted fetch latency from the Styx prefetcher running cluster-wide. It then forwards this estimate to the Styx workflow engine running on the selected node. In parallel, the Styx prefetcher fetches the required object and places it into a distributed in-memory store – such as a Redis Cluster – that is accessible from all nodes, thereby eliminating cross-node data-access complications.

Using the predicted latency, the node's workflow engine attempts to coincide the end times of the I and E stages. After the T stage completes, the container sends a soft acknowledgment to the local workflow engine to initiate execution of the next request on the same container, and notifies the node's data service to perform the S stage. Once the S stage finishes, the data service returns a hard acknowledgment to the local workflow engine. The workflow engine then sends the response to the coordinator to trigger the next function in the workflow. Although this design centralizes prediction, data extraction, and scheduling for simplicity and consistency, it also highlights an opportunity for future work to decentralize these services to reduce bottlenecks and enhance scalability as cluster sizes grow.

B. Impact of In-Network Congestion on Prediction Accuracy

When estimating the file fetch latency of a file, the Styx prefetcher incorporates features derived from both the worker node and the remote storage server (Section IV-E), but does not explicitly model the state of the network fabric between nodes. Under uncongested conditions, packets typically incur only microsecond-scale delays at network switches; however, in the presence of sustained congestion, switch buffering can introduce significant queueing delays that dominate end-to-end transfer time [54]. Prior work has proposed transport-level and in-network mechanisms to mitigate such congestion, and Styx is designed to operate on top of these mechanisms without introducing additional traffic patterns that exacerbate in-network contention [55]–[58]. Our design targets the common operating regime of serverless platforms, where compute and storage nodes experience low to moderate congestion for the majority of executions. As a result, we do not explicitly include network-fabric characteristics as features in the prediction model.

VIII. Conclusion

This paper introduces Styx, a novel workflow engine designed to enhance the resource utilization and response latency in serverless platforms. Our approach coincides the completion of the function initialization and data extraction stages by using a fetch latency predictor, thereby improving the request waiting time. Furthermore, it offloads the data storage stage of a function from a container to the Styx data service running on the host, thereby improving the resource utilization. Experimental evaluations demonstrate that Styx improves overall memory allocation by 32.6% and reduces the tail latency and the mean latency of real-world serverless workflows by an average of 26.3% and 21% when executing all workflows concurrently, respectively.

References

[1] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 1489–1504.

[2] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont *et al.*, "Ofc: an opportunistic caching system for faas platforms," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 228–244.

[3] R. Basu Roy, T. Patel, R. Garg, and D. Tiwari, "Codecrunch: Improving serverless performance via function compression and cost-aware warmup location optimization," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2024, pp. 85–101.

[4] A. Panda and S. R. Sarangi, "Faasctrl: A comprehensive-latency controller for serverless platforms," *IEEE Transactions on Cloud Computing*, 2024.

[5] F. Romero, G. I. Chaudhry, Í. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "Faa$t: A transparent auto-scaling cache for serverless applications," in *Proceedings of the ACM symposium on cloud computing*, 2021, pp. 122–137.

[6] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 419–433.

[7] A. Lambda. (2023) Serverless computing - amazon web services. [Online]. Available: https://aws.amazon.com/lambda/#:~:text=AWS%20Lambda%20is%20a%20serverless,pay%20for%20what%20you%20use

[8] M. Azure. (2023) Azure functions – serverless functions in computing. [Online]. Available: https://azure.microsoft.com/en-us/products/functions

[9] G. Cloud. (2023) Cloud functions. [Online]. Available: https://cloud.google.com/functions

[10] I. Cloud. (2023) Ibm cloud functions. [Online]. Available: https://cloud.ibm.com/functions

[11] S. Poojara, C. K. Dehury, P. Jakovits, and S. N. Srirama, "Serverless data pipelines for iot data analytics: A cloud vendors perspective and solutions," in *Predictive Analytics in Cloud, Fog, and Edge Computing: Perspectives and Practices of Blockchain, IoT, and 5G*. Springer, 2022, pp. 107–132.

[12] R. K. Deka, A. Ghosh, S. Nanda, R. K. Barik, and M. J. Saikia, "Smart healthcare system in server-less environment: Concepts, architecture, challenges, future directions," *Computers*, vol. 13, no. 4, p. 105, 2024.

[13] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 263–274.

[14] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 502–504.

[15] J. McKim, "Serverless event sourcing at nordstrom — a cloud guru," 2017. [Online]. Available: https://web.archive.org/web/20210119044741/https://acloudguru.com/blog/engineering/serverless-event-sourcing-at-nordstrom-ea69bd8fb7cc

[16] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX annual technical conference (USENIX ATC 20)*, 2020, pp. 205–218.

[17] B. Wang, A. Ali-Eldin, and P. Shenoy, "Lass: Running latency sensitive serverless computations at the edge," in *Proceedings of the 30th international symposium on high-performance parallel and distributed computing*, 2021, pp. 239–251.

[18] M. Stein, "The serverless scheduling problem and noah," *arXiv preprint arXiv:1809.06100*, 2018.

[19] G. Somma, C. Ayimba, P. Casari, S. P. Romano, and V. Mancuso, "When less is more: Core-restricted container provisioning for serverless computing," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2020, pp. 1153–1159.

[20] A. U. Gias and G. Casale, "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)*. IEEE, 2020, pp. 1–8.

[21] L. Schuler, S. Jamil, and N. Kühl, "Ai-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments," in *2021 IEEE/ACM 21st international symposium on cluster, cloud and internet computing (CCGrid)*. IEEE, 2021, pp. 804–811.

[22] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li *et al.*, "Help rather than recycle: Alleviating cold startup in serverless computing through {Inter-Function} container sharing," in *2022 USENIX annual technical conference (USENIX ATC 22)*, 2022, pp. 69–84.

[23] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "{SAND}: towards {High-Performance} serverless computing," in *2018 USENIX annual technical conference (USENIX ATC 18)*, 2018, pp. 923–935.

[24] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, 2020, pp. 419–434.

[25] B. Tan, H. Liu, J. Rao, X. Liao, H. Jin, and Y. Zhang, "Towards lightweight serverless computing via unikernel as a function," in *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. IEEE, 2020, pp. 1–10.

[26] C. Marcelino and S. Nastic, "Truffle: Efficient data passing for data-intensive serverless workflows in the edge-cloud continuum," 2024. [Online]. Available: https://arxiv.org/abs/2411.16451

[27] Z. Li, C. Xu, Q. Chen, J. Zhao, C. Chen, and M. Guo, "Dataflower: Exploiting the data-flow paradigm for serverless workflow orchestration," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ser. ASPLOS '23. New York, NY, USA: Association for Computing Machinery, 2024, p. 57–72. [Online]. Available: https://doi.org/10.1145/3623278.3624755

[28] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 symposium on cloud computing*, 2017, pp. 445–451.

[29] I. Minio, "Minio — s3 compatible storage for ai," 2025. [Online]. Available: https://min.io/

[30] Ceph, "Ceph.io — home," 2025. [Online]. Available: https://ceph.io/en/

[31] Amazon, "Amazon s3 - cloud object storage - aws," 2025. [Online]. Available: https://aws.amazon.com/s3/

[32] Microsoft, "Azure blob storage — microsoft azure," 2025. [Online]. Available: https://azure.microsoft.com/en-us/products/storage/blobs

[33] Google, "Cloud storage — google cloud," 2025. [Online]. Available: https://cloud.google.com/storage?hl=en

[34] I. Minio, "Minio — s3 compatible storage for ai," 2025. [Online]. Available: https://min.io/

[35] ——, "Minio object storage for kubernetes — minio object storage for kubernetes," 2025. [Online]. Available: https://min.io/docs/minio/kubernetes/upstream/index.html

[36] S. R. Poojara, C. K. Dehury, P. Jakovits, and S. N. Srirama, "Serverless data pipeline approaches for iot data in fog and cloud computing," *Future Generation Computer Systems*, vol. 130, pp. 91–105, 2022.

[37] A. Pérez, S. Risco, D. M. Naranjo, M. Caballer, and G. Moltó, "On-premises serverless computing for event-driven data processing applications," in *2019 IEEE 12th International conference on cloud computing (CLOUD)*. IEEE, 2019, pp. 414–421.

[38] A. Merenstein, V. Tarasov, A. Anwar, S. Guthridge, and E. Zadok, "F3: Serving files efficiently in serverless computing," in *Proceedings of the 16th ACM International Conference on Systems and Storage*, 2023, pp. 8–21.

[39] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4152–4166, 2021.

[40] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 427–444.

[41] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *16th USENIX symposium on networked systems design and implementation (NSDI 19)*, 2019, pp. 193–206.

[42] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 107–120.

[43] T. Patel and D. Tiwari, "Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 193–206.

[44] Microsoft, "Azure functions http trigger — learn.microsoft.com," https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-http-webhook-trigger?tabs=python-v2%2Cisolated-process%2Cnodejs-v4%2Cfunctionsv2&pivots=programming-language-csharp, [Accessed 21-05-2025].

[45] Serverless, "Serverless framework - aws lambda events - http api gateway v2) — serverless.com," https://www.serverless.com/framework/docs/providers/aws/events/http-api, [Accessed 21-05-2025].

[46] Amazon, "Invoking lambda function urls - aws lambda — docs.aws.amazon.com," https://docs.aws.amazon.com/lambda/latest/dg/urls-invocation.html, [Accessed 21-05-2025].

[47] Docker, "Docker docs — volumes," 2025. [Online]. Available: https://docs.docker.com/engine/storage/volumes/

[48] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 467–481. [Online]. Available: https://doi.org/10.1145/3373376.3378512

[49] Z. Wu, Y. Deng, J. Huang, Q. Yang, P. Zhou, and G. Min, "Chrono: Efficient serverless analytics with adaptive fine-grained partitioning and shadow execution," *IEEE Transactions on Cloud Computing*, 2025.

[50] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 1187–1204.

[51] Y. Tang and J. Yang, "Lambdata: Optimizing serverless computing by making data intents explicit," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, 2020, pp. 294–303.

[52] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Transactional causal consistency for serverless computing," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 83–97.

[53] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," *arXiv preprint arXiv:2001.04592*, 2020.

[54] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable low latency for data center applications," in *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012, pp. 1–14.

[55] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, 2010, pp. 63–74.

[56] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. Pan, B. Prabhakar, and M. Seaman, "Data center transport mechanisms: Congestion control theory and ieee standardization," in *2008 46th Annual Allerton Conference on Communication, Control, and Computing*. IEEE, 2008, pp. 1270–1277.

[57] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is more: Trading a little bandwidth for {Ultra-Low} latency in the data center," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 253–266.

[58] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "Detail: Reducing the flow completion time tail in datacenter networks," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, 2012, pp. 139–150.

Abhisek Panda received a bachelor's degree in computer science and engineering from Odisha University of Technology and Research, Bhubaneswar, Odisha.

He is currently a Research Scholar with the computer science and engineering deparment, Indian Institute of Technology Delhi, New Delhi, India. His current research interests include operating system, Intel SGX, and distributed computing.

Smruti R. Sarangi received the Ph.D in computer science from the University of Illinois at Urbana Champaign(UIUC), USA in 2006, and a B.Tech in computer science from IIT Kharagpur in 2002. After completing his Ph.D he has worked in Synopsys Research, and IBM Research Labs.

He has filed five US patents, seven Indian patents, and has published 130 papers in reputed international conferences and journals. He has published two popular books on computer architecture: (1) ”Computer Organisation and Architecture”, with McGrawHill in 2014 and later with WhiteFalcon in 2021 (”Basic Computer Architecture”) , (2) and ”Advanced Computer Architecture”, with McGrawHill in 2021. He is currently the Associate Editor of the Elsevier Journal of Systems Architecture. He takes an active interest in teaching and technology-enhanced learning. He is currently the HoD of the Educational Technology Services Center. He has gotten the teaching excellence award in IIT Delhi in 2014 and is the latest recipient of the ACM Outstanding Contributions to Computing Education Award (OCCE, 2022). He got the Qualcomm Faculty Award in 2021 and has numerous best paper awards and nominations to his credit.