10-31-2024

Resource Harvesting For Parallel Functions in Serverless Workflows

Peiman Fotouhi
Louisiana State University and Agricultural and Mechanical College

RESOURCE HARVESTING FOR PARALLEL FUNCTIONS IN SERVERLESS WORKFLOWS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science

in

The Division of Computer Science and Engineering

by
Peiman Fotouhi
B.S., University of Isfahan, 2021
December 2024

Acknowledgments

I would like to extend my deepest gratitude to my advisor, Dr. Hao Wang, for his invaluable guidance, patience, and support throughout my research journey. His expertise and encouragement have been instrumental in shaping this thesis, and I am grateful for the opportunities and knowledge he has provided me.

I would also like to express my sincere appreciation to my committee members, Dr. Ibrahim Baggili and Dr. Feng Chen, for their insightful feedback and constructive suggestions. Their expertise and thoughtful contributions have greatly improved my intuition for future directions of this research.

A special thanks to my parents and my siblings for their unwavering support, encouragement, and understanding. Their belief in me has been a constant source of motivation, and I am incredibly grateful for their love and strength.

Table of Contents

Abstract

In the rapidly evolving landscape of cloud computing, serverless architectures have gained attention for their scalability and cost-effectiveness. This thesis aims to introduce a novel approach to maximize resource utilization in serverless environments through the concept of harvesting idle resources within Directed Acyclic Graph (DAG)-based workloads. Our proposed solution targets resource harvesting at parallel stages by utilizing Machine Learning models to accurately harvest or accelerate serverless functions. Additionally, we present a scheduling algorithm specifically designed to address the unique requirements of DAG workloads in cloud environments.

The framework leverages dynamic resource allocation techniques to identify and exploit idle resources within the serverless environment. By intelligently redistributing resources among functions in a parallel stage, our approach minimizes the overall execution time and resource waste, thereby enhancing the overall efficiency of DAG-based serverless computing. Our findings indicate that we can improve CPU utilization by 200% and memory utilization by 30%. Additionally, our system reduces request response latency by 43% on average. This thesis provides a detailed exploration of the proposed solution's architecture, implementation, and performance evaluation using real-world workloads.

Chapter 1. Introduction

The expansion of serverless computing has revolutionized cloud-based application design, offering scalable and cost-effective execution models for various workloads. As serverless applications become increasingly complex, organizing a series of functions into a Directed Acyclic Graph (DAG) becomes crucial, particularly in managing data dependencies among these functions [5]. DAGs facilitate parallel execution, enhancing performance for latency-sensitive and resource-demanding tasks. Nevertheless, the inherently dynamic nature of serverless computing, especially within multi-tenant cloud environments, poses significant challenges in the efficient allocation of resources such as CPU and memory [6]. Unpredictable workloads and varying function execution patterns often result in resource under-utilization or over-provisioning, leading to performance degradation and higher costs.

In serverless DAGs, particularly during parallel execution stages, resource management becomes even more critical. These stages are characterized by the simultaneous execution of multiple functions, each with unique resource demands. The functions in the parallel stage tend to have high variability in terms of resource demand and execution time [14]. Current serverless platforms primarily rely on user-defined resource allocation, which could not capture the demand of functions under different scenarios [13]. As a result, inefficient resource allocation during parallel function execution can cause severe bottlenecks, impacting overall workflow performance.

To address these challenges, the concept of resource harvesting needs to be integrated with such workflows to ensure high resource utilization [22]. Within our system, resource harvesting aims to optimize resource allocation and utilization across parallel stages in DAG workflows by dynamically adjusting resource distribution based on predicted demand. By leveraging machine learning-based prediction models, resource harvesting ensures that resources are allocated efficiently, minimizing waste, and preventing contention between functions.

1.1. Problem Statement

Despite advances in serverless computing, existing resource allocation mechanisms in DAG-based workflows remain limited in their ability to handle the complexity of parallel function execution [12]. Traditional resource allocation strategies, which are often static, do not consider the dynamic and skewed resource demand in DAGs, leading to inefficient utilization of available resources. Furthermore, the lack of sophisticated orchestration mechanisms for parallel stages poses the issue of resource contention and performance degradation.

Furthermore, existing platforms [11, 5, 17] do not offer a scheduling mechanism adapted for serverless DAGs. By considering the data dependency between functions of a DAG, we aim to propose a scheduling algorithm that can reduce inter-invoker communication for DAG jobs, leading to lower latency.

This thesis addresses these limitations by proposing a resource harvesting framework for parallel function execution in serverless DAGs. The framework integrates machine learning-based inference pipelines to predict the CPU and memory demand of each function in the parallel stage, maintaining optimal resource allocation. The proposed system leverages the AOpenWhisk platform and introduces enhancements such as CPU-aware scheduling and harvesting validation checks, ensuring that resources are allocated based on real-time demand rather than static user-defined values.

1.2. Objectives

The primary objectives of this thesis are as follows.

- **Design and implement Serverless Workflows** that have a high degree of skew among parallel functions as observed by Cloud Providers.

- **Design and implement a resource harvesting framework** that optimizes resource allocation for parallel functions in serverless DAGs, with a focus on CPU and memory utilization.

- **Develop a machine learning-based inference pipeline** capable of predicting the resource demand (CPU and memory) of each function within the parallel stage of a DAG based on the input size and function characteristics.

- **Validate the feasibility of Harvesting Desicion** to ensure that resource reallocation occurs only within valid bounds, preventing resource starvation or over-provisioning.

- **Efficent Placement of Function with respect to DAG Structure** by considering the data dependency among functions within a serverless DAG

- **Evaluate the performance and efficiency** of the proposed framework through experimentation, measuring improvements in resource utilization and latency.

1.3. Contributions

This thesis makes the following contributions to the field of serverless computing and resource management:

- **Design and Implementaion of three DAG Workflows** that are in line with the real-world applications in terms of skew among parallel functions [14].

- **A resource harvesting framework** that optimally distributes resources during the parallel execution stages of serverless DAG workflows, addressing inefficiencies in current serverless platforms and user's configuration.

- **A machine learning-driven resource demand prediction model**, which accurately forecasts the CPU and memory requirements of individual functions based on input size and system conditions.

- **An extension of the Apache OpenWhisk orchestration model** to include DAG-based scheduling and dynamic resource reallocation for parallel functions.

- **The development of a composition-aware load balancing algorithm**, which reduces data communication overhead by placing functions within the same composition on the same invoker.

1.4. Thesis Organization

The remainder of this thesis is structured as follows:

- **Chapter 2** provides a review of related work in serverless computing, DAG-based workflows, and resource allocation strategies. It also demonstrates our motivation for investigating the problem

- **Chapter 3** introduces the proposed resource harvesting framework and outlines the system architecture.

- **Chapter 4** presents the experimental setup and evaluation of the framework, including performance benchmarks and resource utilization metrics.

- **Chapter 5** provides the limitation of this work and possible future directions.

- **Chapter 6** concludes the thesis with a summary of system design and findings.

Chapter 2. Background and Motivation

In this chapter, we introduce some of the concepts discussed in the thesis, as well as state-of-the-art research and their limitations. At last, we discuss the motivation for designing our system.

2.1. Serverless Computing

Serverless computing, also known as Function-as-a-Service (FaaS), represents a paradigm shift in cloud computing by abstracting infrastructure management from developers. In this model, applications are divided into independent, event-driven functions that are executed in response to specific events (e.g., data, API call, load). The serverless model provides several key benefits. These advantages have led to the widespread adoption of serverless computing across a variety of domains, from web services [16] to machine learning [9] and data processing [10].

One of the key benefits of serverless computing is its pay-per-use billing model. Unlike traditional cloud computing models, where users must provision and pay for dedicated resources regardless of actual utilization, serverless platforms charge only for the resources consumed during the execution of functions. This model allows developers to optimize costs, paying strictly for the execution time and memory consumed by their functions. Additionally, serverless platforms automatically scale functions in response to varying demand [3], eliminating the need for manual resource management and scaling configurations. This not only simplifies application deployment but also makes serverless computing an ideal choice for applications with unpredictable or bursty workloads [18]. These operational benefits, combined with the ease of development and deployment, have made serverless computing an attractive solution for developers and organizations looking to reduce costs and streamline their infrastructure.

However, as serverless computing continues to grow in popularity, several challenges have emerged that motivate ongoing research in this field. One key area of research focuses on resource management and performance optimization. Unlike traditional cloud models, serverless platforms offer limited control over resource allocation, often leading to suboptimal performance for complex applications such as workflows with parallel function execution. Researchers are exploring novel approaches for fine-grained resource management, including dynamic resource allocation strategies, function scheduling, and infrastructure optimizations to reduce latency and maximize resource utilization.

Another active area of research centers around stateful function execution. Serverless platforms are inherently stateless, meaning that functions must retrieve and store any required state in external storage systems between executions. This introduces additional overhead in terms of latency and cost, especially for applications that require frequent access to shared state or must maintain state across function invocations. Recent research efforts have focused on designing stateful serverless architectures that minimize these overheads and provide more efficient mechanisms for state management.

These areas of research, alongside others such as security, cold-start mitigation, and multi-cloud orchestration, highlight the ongoing evolution of serverless computing. As cloud providers and the research community continue to innovate, the potential applications and efficiency of serverless computing will expand, particularly in workflows that demand scalable, high-performance execution environments [7].

2.2. Resource Harvesting

Resource Harvesting refers to utilization techniques applied for reallocating computing resources(i.e. CPU, Memory) in the cloud in order to mitigate resource waste and increase the efficiency of a cloud-based service. The goal is to extract maximum value from the resources that are already allocated to users without interfering with the lifecycle of user requests. This method can help providers with cost savings and also help users with reducing the latency of their applications without having to worry about the underlying infrastructure.

Resource harvesting is a critical technique for improving resource utilization in serverless computing environments by reallocating unused resources from over-provisioned functions to under-provisioned ones. In serverless platforms, where workloads are dynamic and function invocations can have varying input sizes, resource misconfiguration often leads to inefficient resource allocation. Resource harvesting addresses this by ensuring that idle resources are captured and reassigned to functions that require more than their originally allocated resources.

A general work for harvesting resources in the cloud was introduced by Zhang et al.[24] which explores the integration of serverless platforms with Harvest VMs. This approach addresses the challenge of using ephemeral resources—VMs that can grow and shrink dynamically based on the availability of unused resources—while maintaining system stability and performance. The study contributes a load balancer specifically designed to handle the challenges of resource evictions and variability in Harvest VMs, providing a model for efficiently managing transient resources in serverless architectures.

SmartHarvest[20] proposes a method to safely harvest idle CPU resources in cloud environments, ensuring that critical tasks or latency-sensitive applications are not impacted. It emphasizes a mechanism to monitor CPU utilization and dynamically harvest resources only when they are truly idle, preventing resource contention.

Several significant contributions have advanced the field of resource harvesting in serverless platforms. One such contribution is Libra [21], a framework designed to enable safe and timely resource harvesting for serverless providers. Libra profiles both resource demands and availability based on the input size of functions, allowing it to make harvesting decisions that adapt to the fluctuating resource needs of serverless functions. Libra uses Machine Learning models and histograms to build its profile. Our work is directly influenced by Libra.

Building on these ideas, *Freyr* introduces a novel resource manager that applies deep reinforcement learning to dynamically manage resource harvesting. Freyr's key contribution lies in its ability to learn from real-time data and optimize resource allocations by detecting over-provisioned and under-provisioned functions. This system does not provide a mechanism for the timely release of resources.

Freyr$^+$ [23] extends the contributions of Freyr by incorporating advanced machine learning techniques, such as attention mechanisms, to improve the precision and efficiency of resource harvesting. The introduction of incremental learning in Freyr$^+$ further enables the system to continuously improve its harvesting strategies as new data becomes available. Furthermore, the issue of timeliness of harvesting is also addressed by Freyr$^+$.

Table 2.1 provides general information about the works discussed in this section.

Table 2.1. Existing Work on Resource Harvesting in the cloud

Paper/Feature	**Zhang et. al.**	**SmartHarvest**	**LIBRA**	**Freyr**	**Freyr$^+$**
Serverless	✗	✗	✓	✓	✓
Timeliness	✗	✗	✓	✗	✓
Safety	✗	✓	✓	✗	✓
Open Source	✓	✗	✓	✓	✗
Harvest VMs	✓	✓	✗	✗	✗

2.3. Serverless DAGs

Serverless DAGs are a collection of serverless functions that are chained together to serve an application logic. During the last few years, this paradigm has drawn the attention of serverless users due to its serverless design and its support for complex application logic. Serverless DAGs have been widely used for data processing and ETL pipelines, machine learning pipelines, image and video processing, event-driven microservices architectures, and real-time analytics. Figure 2.1 shows a basic MapReduce application as a serverless DAG.

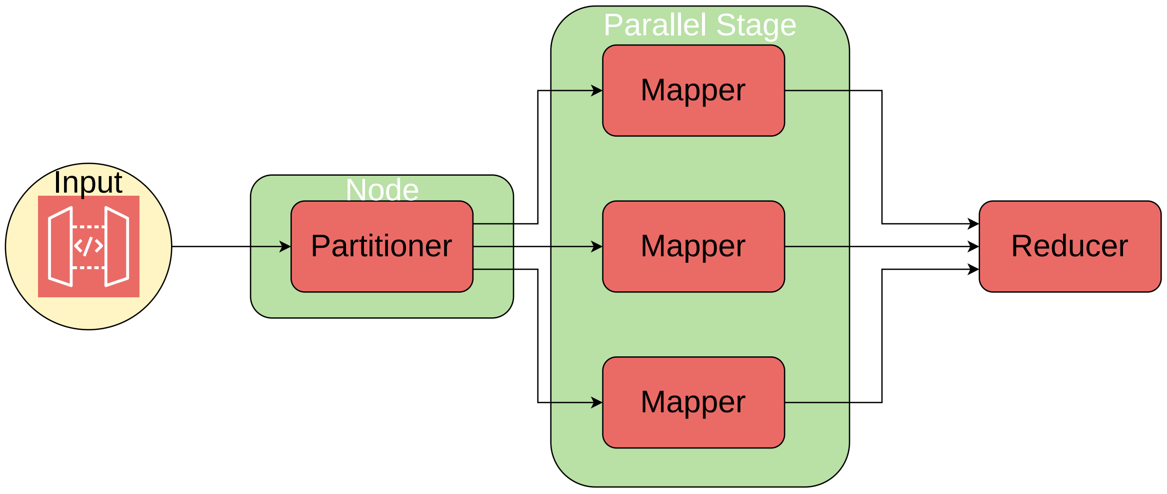


Figure 2.1. Simple DAG for a MapReduce Application

Nodes. Each node in the DAG is an individual serverless function. Such functions are stateless and do not have any information about the execution state of other nodes.

Edges. They represent the data-flow between nodes. A serverless function will only start execution after its predecessors have finished and passed their output to the current node.

Start Node. This is the first node in the execution flow of a serverless DAG. The input for this node is passed by the user through different methods(i.e. API endpoints, CLI, HTTP requests). Also, the last node will be outputting the final result of the DAG, which will be returned to the user through the same means.

Parallel Stages. Such stages are branched out from a single node into multiple branches. At the end of the branch, they may accumulate into a single node. This mechanism is often called a Fan-out/Fan-In scenario.

Execution Skew. If parallel branches within a parallel stage have different execution times due to different inputs or different code logic within a node, a skew scenario arises. Execution skew among multiple parallel functions in a DAG is the cornerstone of our motivation for harvesting opportunities.

There are multiple research works on DAGs, but it has not been integrated into Resource Harvesting research to the best of our knowledge. Table 2.2 the implementation setup of some of the systems that were investigated in our work.

Table 2.2. Implementation setup of existing work on serverless DAGs

Paper	**Programming Language**	**Evaluation Setup**
Orion	Python, C#	AWS Lambda
FaastLane	Python, Scala	OpenWhisk(Local Cluster)
Chiron	Python, C	OpenFaaS(Local Cluster)
UnFaaSener	Python, C++	Google Cloud Functions, GC E instances

ORION[14] proposed a framework as a solution for optimizing serverless DAGs.

Here authors aim to address three problems. The first problem is the most addressed issue in the serverless paradigm, which is the issue of cold start, where containers or microVMs need to load the runtimes and required services even before the invocation of the corresponding function. Also, this should be done in a timely manner since, according to this paper, even early pre-warming could introduce significant overheads. The second problem is that execution time for functions inside a serverless application is highly skewed, meaning that based on the functionality of some functions, they can take much longer to execute compared to the average execution time for that application. These functions can be recognized with online profiling and can be allocated more resources for execution to mitigate the straggler effect. The third problem is that given a fixed amount of resources and the current state of the application, there exists another skew in execution that corresponds to the application load depending on the other functions that are being executed when we are trying to invoke a function. This can be solved by bundling different functions inside a given container to achieve higher resource utilization and lower latency. ORION's findings are important in our system design as the authors directly analyzed the DAG invocations at Microsoft Azure. They found that almost 40% of DAGs used in Azure consist of parallel stages. They further indicated that 98.2% of DAGs used in Azure have a skew ratio of $\geq 2X$, which means there is a significant straggler effect present in serverless DAGs. Therefore, we propose to harvest the resources of the faster functions and accelerate the straggler functions in a parallel stage, which can significantly reduce the execution time of a serverless DAG.

FaastLane[11] first introduced the usage of both processes and threads inside a single container to execute functions in the DAG. It also is the first serverless paper to address the issue of concurrent access to data in interpreted languages (i.e. Python, JavaScript). The issue is that the language runtime does not allow the creation of multiple threads and their execution on different CPU cores. This leads to concurrency of threads on only one CPU core which does not guarantee real parallelism. The authors use Intel MPK to give key-based access to threads for their own data in the virtual address space of the running process.

Chiron[12] is inspired by FaastLane. It is the first to introduce the M-to-N deployment model. With this model the serverless workflow that consists of M distinct functions can be mapped to N containers. Each container can run multiple functions. Functions that are in sequence will be mapped to a multi-thread process to reduce the communication overhead by keeping the intermediate data inside the virtual address space of the process and allowing threads to access that. The parallel functions within the container need to be mapped into separate processes due to the GIL problem of interpreted languages, as discussed above. A major novelty of this paper over FaastLane is using a process pool inside the container to avoid keeping the processes required, and therefore avoid forking processes on the go, which can impose high latency.

UnFaaSener[17] offers a user-side solution. Such solutions offer a variety of configurations to user code since, in serverless computing, user code should not be visible to the platform for security reasons. UnFaaSener is inspired by the fact that some cloud users have unused cloud computing resources that are already paid for but might be idle. This platform aims to execute serverless workflows using the existing serverless platforms as well as the user resources. This enables the user to have more cost-saving compared to running everything on a serverless platform. This mechanism is called host offloading in the context of this paper. It should be noted that this paper does not change the architecture of existing serverless platforms but rather re-organizes the DAG so that host offloading can be done. Furthermore, the paper benefits from multiple components that enable cost-effective and SLO-aware host offloading.

The research on serverless DAGs has addressed multiple issues in serverless computing. Table 2.3 demonstrates the areas of study of such research. Despite its broad coverage, the research on serverless DAGs has not been integrated into Resource Harvesting techniques by the state-of-the-art. Our work aims to be the first to investigate the Resource Harvesting opportunities in the context of parallel functions.

Table 2.3. Existing Work on Serverless DAGs

Paper/Feature	**Orion**	**FaastLane**	**Chiron**	**UnFaaSener**
Serverside Solution	✓	✓	✓	✗
Resource Harvesting	✗	✗	✗	✗
Thread level scheduling	✗	✓	✓	✗
Prewarming and Coldstart mitigation	✓	✗	✗	✓
Function Bundling	✓	✓	✓	✗
Latency prediction	✗	✗	✓	✓

2.4. Existing DAG-Based Platforms in Production

DAG-based serverless computing has gained traction through various platforms provided by leading cloud service providers. These platforms offer robust tools and frameworks to orchestrate complex workflows using Directed Acyclic Graphs (DAGs), enabling efficient execution and management of serverless applications.

2.4.1. AWS Step Functions

Amazon Web Services (AWS) Step Functions(ASF) is a fully managed service that allows users to coordinate and orchestrate distributed applications using visual workflows.

It provides a graphical interface to design and execute workflows based on state machines represented as DAGs. AWS Step Functions supports a variety of integrations with AWS services, enabling developers to build scalable and fault-tolerant applications by defining workflows with different states and transitions.

2.4.2. OpenWhisk Composer

OpenWhisk, an open-source serverless platform, offers the Composer [2] tool for constructing complex workflows using DAG-based compositions. OpenWhisk Composer allows developers to create serverless applications by chaining together multiple functions through a declarative programming model. By defining sequences and dependencies among functions, users can design and execute sophisticated workflows efficiently.

2.4.3. Google Workflows

Google Cloud Workflows [8] is a fully managed service that enables the creation and execution of serverless workflows using a visual editor or code-based approach. Leveraging the power of DAGs, Google Workflows allows users to automate and orchestrate various Google Cloud services and external APIs. It supports defining workflow steps, dependencies, and conditional branching, enabling the creation of scalable and resilient applications.

2.4.4. Azure Durable Functions

Microsoft Azure offers Durable Functions [15], an extension of Azure Functions that enables the creation of stateful serverless workflows using DAGs. Azure Durable Functions allow developers to define orchestrator functions that coordinate the execution of multiple stateless functions in a durable and reliable manner. This platform facilitates building complex workflows with features for managing state, timeouts, and human interactions.

The most widely used platforms currently being used were listed in this section. Each tool can be integrated to other features within the provider. For instance ASF can be integrated with AWS CloudWatch or Amazon S3 intermediate data store to support the storage of data being passed between functions in a more reliable way.

2.5. Life-cycle of a Request within OpenWhisk

In this section, we briefly introduce major components of OpenWhisk as a system that we built our prototype on top of. Detailed modifications of the system can be found in Chapter 3

The internal processing of an action invocation in OpenWhisk involves several key components, each responsible for specific tasks to ensure efficient execution. The following steps outline the processing flow from the invocation request to the execution of the action. Figure 2.2 depicts a general overview of OpenWhisk internals.

Nginx - Entry Point. The first component that handles an incoming request is Nginx. OpenWhisk's user-facing API is entirely HTTP-based and follows a RESTful design. When an action is invoked, whether by the CLI or via an API call, the request is routed through Nginx. It acts as an HTTP server and reverse proxy, responsible for handling requests and forwarding the HTTP request to the next component, the Controller.

Controller - Request Handling. Once the request passes through Nginx, it reaches the Controller. The Controller serves as the core API handler for OpenWhisk. Implemented in Scala, it processes all incoming requests, including the creation, updating,

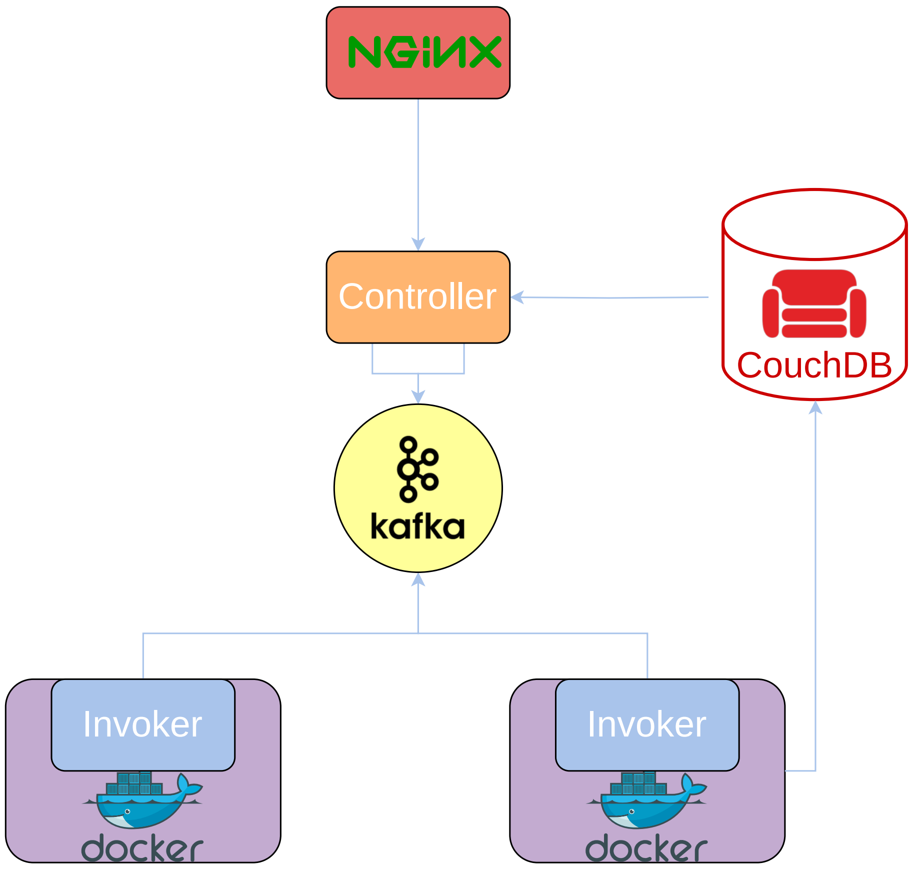


Figure 2.2. Overview of OpenWhisk

and invocation of actions. After receiving an invocation request, the Controller parses the request and determines the appropriate action to be executed.

Fetching the Action. Next, the Controller retrieves the actual code of the action from the CouchDB database. Each action is stored in CouchDB under the user's namespace and contains essential information such as the action code, any default parameters, and resource constraints (e.g., memory limits). This information is necessary to properly execute the action.

Load Balancer - Choosing an Invoker. Once the action is fetched, the Controller consults the Load Balancer. The Load Balancer's job is to assign the action to an available Invoker, which will execute the code. The Load Balancer keeps track of the health status and workload of all Invokers in the system, ensuring that actions are distributed efficiently.

Message Queuing with Kafka. To ensure the robustness and reliability of action execution, OpenWhisk uses Kafka as a messaging system. After the Controller selects an Invoker, the action invocation request is placed into a Kafka queue. This queue acts as a buffer, ensuring that no requests are lost if the system crashes or is under heavy load. Kafka's high-throughput, distributed nature guarantees that action invocation requests are handled reliably and in order.

Invoker - Executing the Action. The Invoker is responsible for executing the action. Each Invoker runs in a controlled environment using Docker containers. When the Invoker receives a request from Kafka, it spins up a Docker container, injects the action code, executes it with the provided parameters, and captures the result. After execution, the container is destroyed to ensure that each action runs in isolation and does not interfere with others. This containerized approach provides security and efficient resource utilization.

CouchDB. After the execution of a function, its results are posted into CouchDB. The user can then fetch the results from the database through its API. The database also keeps track of the status of the function. If function executions fail, it is reported to the user.

2.6. Motivation

Although most of the DAG applications with parallel stages in the cloud have a high degree of skew, users tend to over-provision or under-provision resources to functions in parallel stages due to their lack of knowledge. Although users would already be billed for the resources they are allocated, they will not efficiently utilize such resources. This would result in higher end-to-end time for user requests. While serverless providers have introduced monitoring mechanisms for users to find optimal resource allocations, such allocations can become suboptimal as soon as the user's input size changes. This motivates us to investigate the possibility of harvesting idle resources allocated to functions in parallel stages and allocating them to other functions in the same stage that have higher resource demand. To demonstrate this opportunity, we provide an illustrative example of how this works using one of our implemented workflows. Figure 2.3 depicts the structure of our Video Analytics workflow. This workflow consists of 4 functions, 2 of which are in a parallel stage.

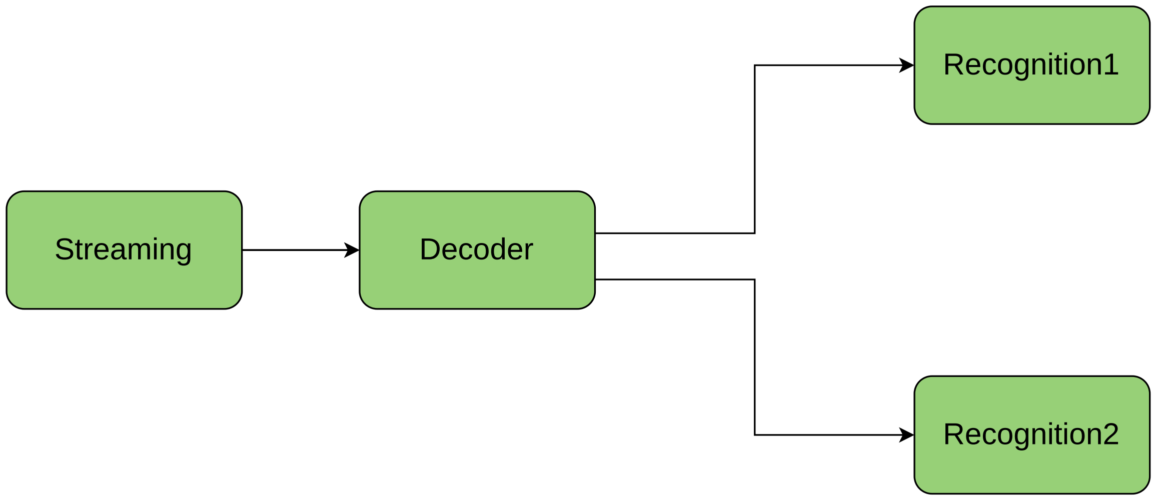


Figure 2.3. Workflow of Video Analytics DAG

We investigate an illustrative example of resource allocation for a Directed Acyclic Graph (DAG) to justify resource harvesting for parallel functions in serverless DAGs. Figure 2.4 shows the CPU and memory utilization of two functions in the parallel stage of the DAG. For the first function, ‘partitioner1‘, the user specified 2 CPU cores and 4 units of memory, while for the second function, the user specified 3 CPU cores and 4 units of memory. However, due to the actual resource demand of these functions, the first function only utilizes half of its allocated resources, whereas the second function fully utilizes its allocated resources.

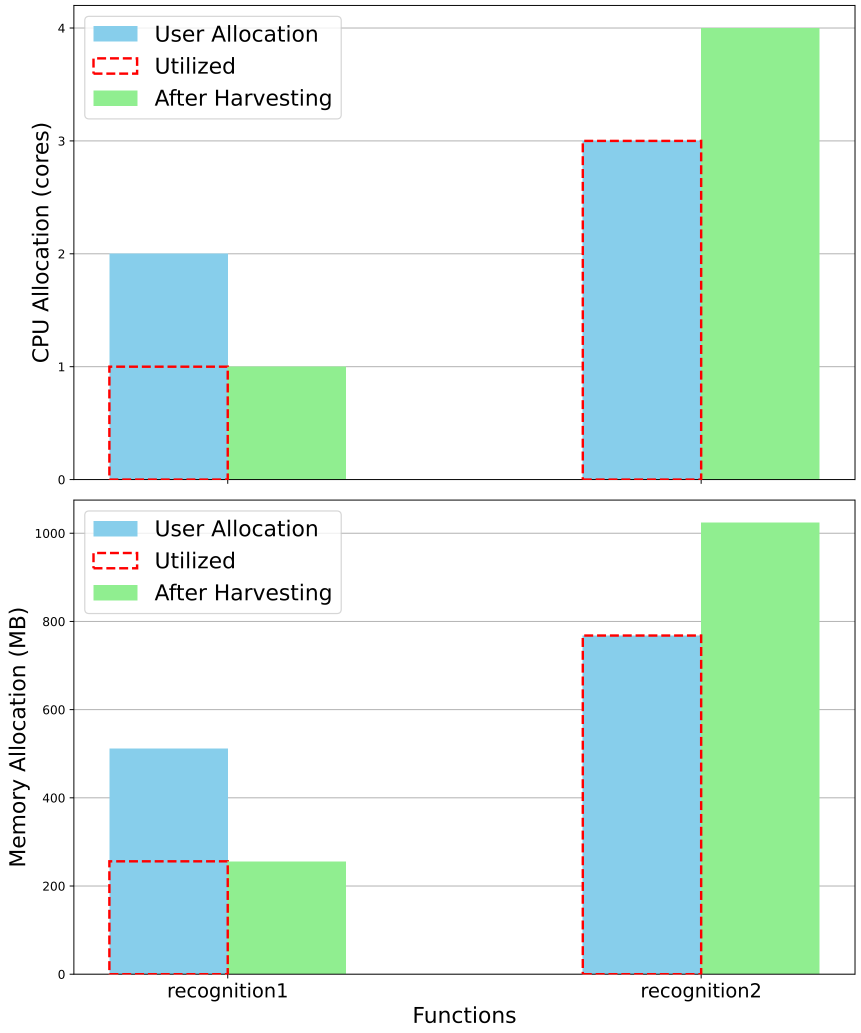


Figure 2.4. Memory and CPU Allocation Parallel Function in Video Analytics(30 frames)

By harvesting the underutilized resources from the first function and reallocating them to the second function, which still has unmet resource demand, we can reduce the execution time of the second function. Figure 2.5 presents the execution timeline for the DAG and its components with and without resource harvesting. Our analysis shows that by harvesting resources from ‘partitioner1‘ and reallocating them to ‘partitioner2‘, we achieve a reduction in the end-to-end runtime of the DAG by almost 25 percent.

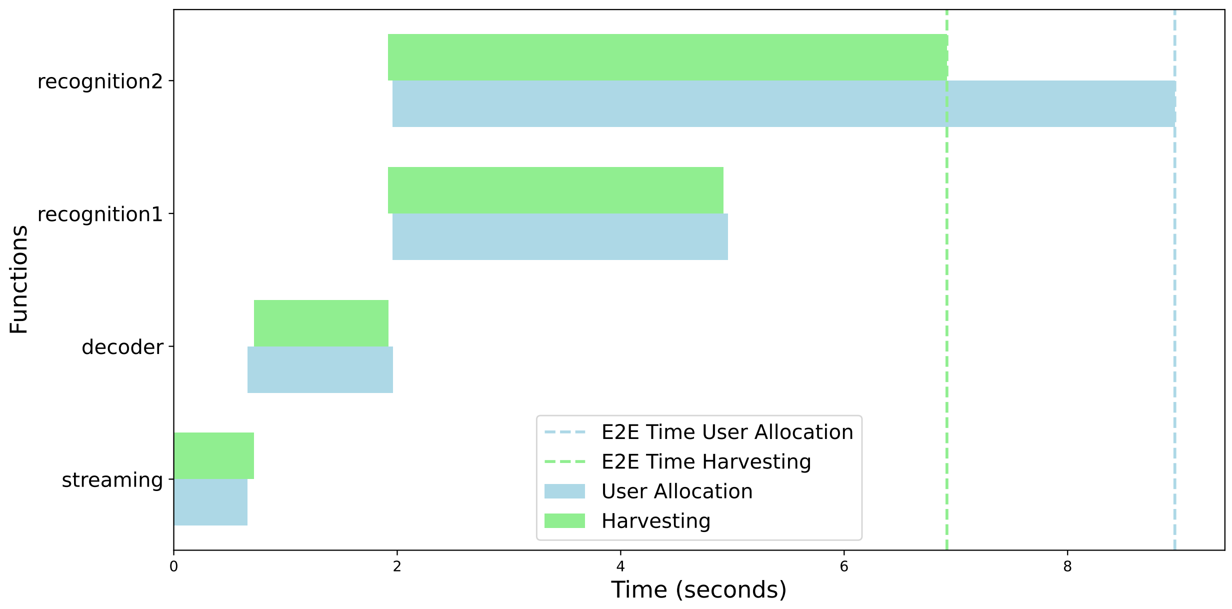


Figure 2.5. Execution Timeline for Video Analytics(30 frames)

Chapter 3. System Design

3.1. Compositions

In order to motivate our design tailored for harvesting idle resources among parallel functions within serverless DAGs, we need to articulate the concept compositions in our system design. Within the boundaries of our system, a composition is a high-level construct that wraps around a chain of serverless functions and, unlike the functions, can statefully monitor their execution and data flow. Figure 3.1 shows compositions generated in our MapReduce example to keep track of DAG execution. When a DAG is invoked in our system, a composition is generated to keep track of its execution. This composition passes the input to the Partitioner function; when the Partitioner's output is generated, it is sent back to the composition for the next step. Next, for each parallel branch, a composition is generated. This composition retrieves the input data from the parent composition and invokes its containing function, which is the Mapper; when a Mapper is done with its execution, it informs its composition and the composition stores the result in a Redis key-value cache. As soon as all the compositions in the parallel stage have posted their results to Redis, the parent composition retrieves the results and sends them to the last function, which is the Reducer. At last, Reducer sends its output to the parent composition. The parent composition now holds the results and will return the results to the user. It should be noted that the parent composition is the long-running action(serverless function) that is active throughout the execution of the DAG. The child compositions are also defined as actions that run throughout the timeline of the parallel stage.

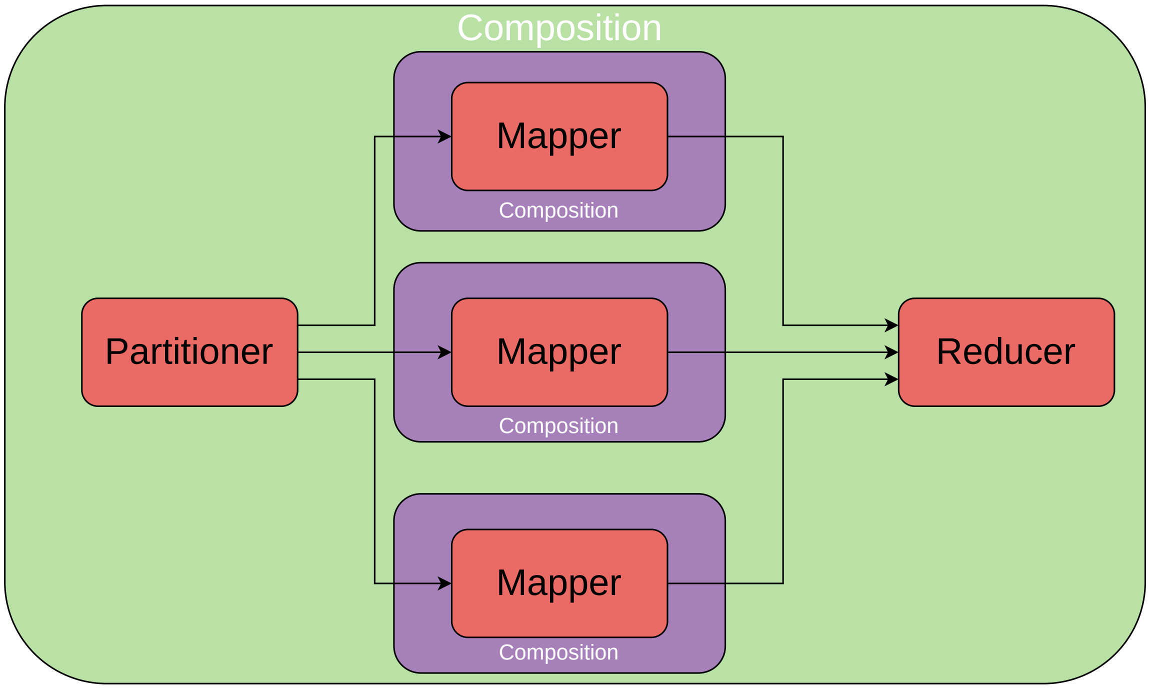


Figure 3.1. Compositions in a DAG

3.2. Impact of Input Size on E2E Latency

In serverless platforms, providers do not have direct access to user code or input content, which creates a significant challenge when estimating resource demands for function execution. This lack of visibility forces serverless systems to rely on general-purpose heuristics or static allocations, often leading to inefficiencies, especially in complex workflows such as Directed Acyclic Graphs (DAGs), where parallel stages can have varying resource requirements.

Also, different input sizes for a given DAG can lead to different E2E times. To show the effect of input size on E2E time for DAG execution, we consider one of our workflows. Figure 3.2 shows the structure of the ParallelAES workflow.

We then invoke this DAG in our system with 3 different input sizes and monitor

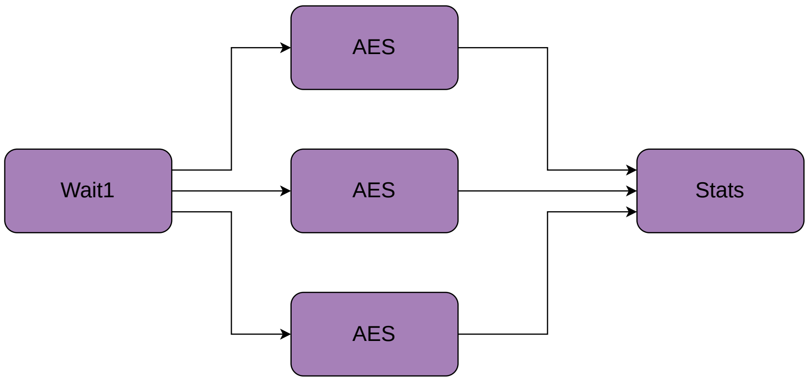


Figure 3.2. ParallelAES Structure

the execution time of the DAG. Figure 3.3 demonstrates the execution timeline of this DAG. By increasing the input size, we observe that the execution time for the functions in the parallel stage increases. This motivates us to use input size as the main parameter for harvesting decisions in our design.

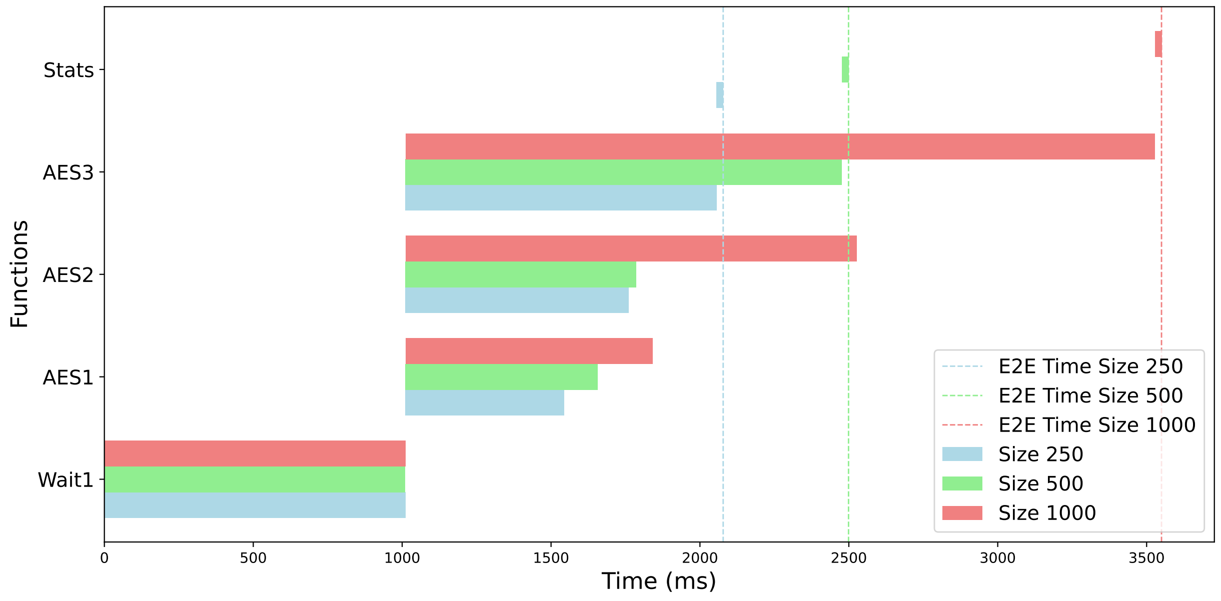


Figure 3.3. ParallelAES Execution Timeline For Different Input Size

If we can identify a consistent relationship between the DAG's input size and the input sizes of the functions within its parallel stage, it becomes possible to predict the resource demand for these functions with reasonable accuracy. This predictive capability forms the basis of our profiling mechanism, which uses input size as a key factor in resource demand estimation. This mechanism not only enables dynamic resource allocation but also improves the efficiency of serverless DAG execution by minimizing resource overprovisioning and underutilization.

The ability to predict resource demand based on input size is particularly crucial for optimizing the end-to-end execution time of DAGs. With accurate resource predictions, functions in the parallel stage can be better scheduled and allocated, ensuring that they have the necessary CPU and memory to execute efficiently. This leads to lower latencies and better overall performance of serverless workflows, especially for data-intensive applications that involve substantial parallel computation.

3.3. Designing Skew-Aware Workloads

One of the major observations in modern serverless computing is the phenomenon of execution skew in parallel functions within DAGs as highlighted by ORION[14] The authors identified that, in real-world DAG executions, the runtime behavior of parallel function invocations can exhibit significant variability. While some functions complete their executions quickly, others experience delays due to a variety of factors such as uneven data distribution, underlying resource constraints, or code logic. This execution skew results in prolonged execution times for the entire DAG and poses a threat to the efficiency of serverless workloads.

In the context of resource harvesting, this skew poses a unique challenge. If the execution patterns of DAGs are not accurately captured, resource harvesting strategies may either over-provision or under-provision resources, leading to inefficiencies. Current serverless DAG workloads and benchmarks tend to assume uniform execution times for parallel tasks, neglecting the execution skew observed in real-world scenarios. As a result, they fail to provide a realistic testbed for evaluating optimization strategies like resource harvesting.

To address this gap, we designed new skew-aware DAG workloads that intentionally incorporate execution variability across parallel functions. These new workloads capture the asymmetry in execution times, making them a better fit for evaluating resource harvesting strategies. By mimicking the execution skew, our workloads ensure that resource harvesting techniques are tested under conditions that more closely resemble real-world serverless function invocations. This design approach helps expose the limitations of existing harvesting techniques and motivates the development of more robust strategies that can adapt to the non-uniformity in execution patterns. Workflow characteristics are summarized in Table 3.1 below. More details about implementing and integrating the DAG Workflows with our system can be found in Appendix A.

Table 3.1. Overview of Designed Workloads

Name	**Type**	**Number of Functions**	**Depth**	**Width**
Video Analyitcs	Image Recognition	4	3	2
Parallel AES	Text Encryption	5	3	3
ML Pipeline	Model Training	6	3	4

3.4. Challenges

We design our system to address the following challenges:

How to predict the amount of resources required to minimize the execution time of DAGs with different input data? According to ORION [14] input data and user code are the main contributing factors to the execution time of functions in the parallel stage. However, due to security reasons, serverless providers cannot access the content of user input or their code. We propose an offline profiling mechanism that successfully captures this dependency by only inspecting the size of user input without inspecting the content of the input data or the user code.

How to simultaneously predict the resource demand of all functions within a parallel stage of DAG at DAG request arrival without interfering with the execution flow of the DAG? As there are multiple functions within a DAG, it is important to predict their resource demand after the arrival of the request. In this regard, we cannot wait to check the input size of each stage in the DAG during execution. To remedy this, we propose the Prediction Pipeline which can predict the resource demand of all functions in a parallel stage by only looking at the input size of the DAG.

How to make sure our harvesting mechanism only harvests the resources among functions in a parallel stage and does not claim the resources of other functions at execution? Since the motivation for our work is to only harvest resources among a parallel stage of a DAG to decrease its execution time, we need to make sure that we do not attempt to harvest resources that do not belong to functions in this stage. To do that, we design a Harvesting Validator which validates if a prediction would be in line with our constraint.

How to place functions into execution nodes in a way that the least amount of latency is imposed and we achieve high utilization? As there is a significant amount of data passed between functions in a DAG, especially for data-intensive workloads, we need to implement a strategy to efficiently place functions into invokers in order to reduce the latency. We design a Composition Cache to keep track of such dependencies. We also need to make sure that the load is distributed evenly among invokers. To address these two issues, we design a Composition-Aware Greedy Join-Shortest-Queue algorithm that is tailored for the scheduling of serverless DAGs.

3.5. Prediction Engine

We design a prediction engine for predicting maximum utilization of CPU and Memory for serverless functions. Our engine consists of two major components:

3.5.1. Profiler

Resource Usage Monitoring Thread

In our system, a resource usage monitoring thread implemented using cgroups V2 [19] runs alongside each serverless function to track the CPU and memory consumption during the function's lifetime. The thread is active only while the function is executing within its container and terminates when the function completes. The primary goal of this monitoring is to capture the peak resource utilization without introducing significant overhead. The thread stores its result as part of the function's activation message, which is then stored in a CouchDB database. We retrieve the results from this database to construct DAG profiles. Implement the thread using cgroups V2 [19]. Algorithm 1 shows the pseudo-code for this monitoring thread.

Algorithm 1 Resource Usage Monitoring Thread

```
 1: function MONITOR_CONTAINER(interval, queue_cpu, queue_mem, stop_signal)
 2:     while stop_signal is not set do
 3:         CPU monitoring
 4:         prev_cpu_usage ← read cgroup CPU usage
 5:         prev_system_usage ← read system CPU usage
 6:         sleep for interval
 7:         after_cpu_usage ← read cgroup CPU usage
 8:         after_system_usage ← read system CPU usage
 9:         delta_cpu_usage ← after_cpu_usage - prev_cpu_usage
10:         delta_system_usage ← (after_system_usage - prev_system_usage)
                    × conversion_factor
11:         cpu_busy ← delta_cpu_usage / delta_system_usage × online CPUs
12:         Memory monitoring
13:         mem_total ← read cgroup memory usage
14:         mem_cache ← read memory used by cache
15:         mem_busy ← (mem_total - mem_cache) / megabytes_per_byte
16:         Add results to queues
17:         add (cpu_timestamp, cpu_busy) to queue_cpu
18:         add (mem_timestamp, mem_busy) to queue_mem
19:     end while
20: end function
```

CPU Monitoring. The thread calculates the CPU utilization of the function by comparing the CPU time consumed by the container against the system-wide CPU usage. The result is normalized over the number of available CPUs in the container's cgroup.

Memory Monitoring. The memory usage is monitored by measuring the total memory consumed by the container and subtracting the memory used by the cache. **Termination:** The thread is terminated once the function finishes execution, as indicated by the `stop_signal` event, thereby avoiding unnecessary resource consumption after the function has ended.

This thread ensures that resource usage is monitored in real-time during function execution, facilitating informed resource management decisions.

We explain each component within our profiler and how they are connected in detail. Figure 3.4 shows the overall design of our offline profiling mechanism.

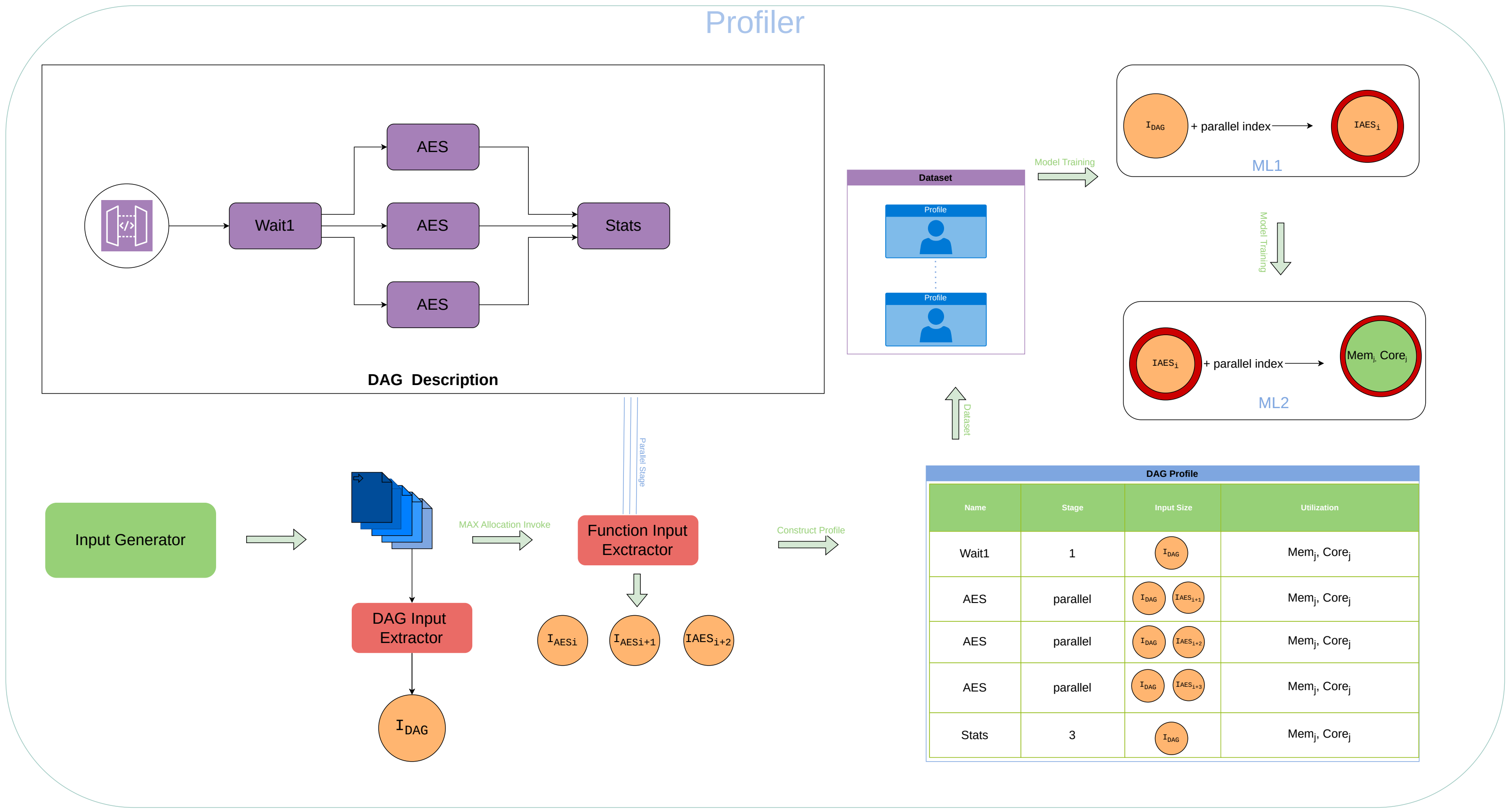


Figure 3.4. DAG Profiler

DAG Description. This component contains details about the functions that constitute the DAG. It includes critical metadata such as the function name, execution runtime, the location of the compressed source code, the stage to which the function belongs, the user's default resource allocation, and the JSON structure representing the DAG.

Input Generator. This module is inspired by the Workload Generator from LIBRA [21]. Its role is to produce a set of input files, but unlike LIBRA, it generates input files specifically for DAG workloads. During our profiling phase, we generate 100 input files of varying sizes for each DAG workload.

DAG Input Extractor. This component is responsible for extracting the size of the DAG's input and recording it in its profile. It is important to note that the size of the input at the DAG level differs from the input size of the individual functions within the DAG.

Max Allocation Invocation. At this stage, all DAG invocations are invoked sequentially, with each function being allocated the maximum possible resources in our system. This ensures that all functions receive more resources than their anticipated demand, allowing us to monitor their true peak resource usage during execution. Table 3.2 presents the allocation settings of our system. In our system, resource allocation for functions can be performed either through direct user allocation or through a harvesting/acceleration mechanism. In the case of user allocation, the user specifies the desired CPU and memory resources for each function, with the allocation ranging from a defined minimum to a maximum in fixed increments. For harvesting/acceleration, the system dynamically allocates resources based on current conditions, while adhering to the same predefined increment and range as user allocations.

Function Input Extractor. During the profiling phase, we capture the input size of each function within the DAG. This is done by retrieving the outputs of all stages in the DAG from CouchDB after the completion of all invocations. The output of each stage serves as the input for the subsequent function in the next stage. The input size of each function is then analyzed and stored in a separate profile for each DAG invocation.

DAG Profile. This component stores the data of all functions within the DAG. Each DAG profile includes the function name, its stage information, and its input size. For functions that are not part of a parallel stage, we record the input size of the entire DAG. Additionally, we store the CPU and memory utilization for each function as captured by our resource monitoring agent. All DAG profiles are then aggregated into a single dataset, which serves as input for our machine learning prediction models.

ML Models (ML1). This refers to a set of machine learning models trained on the collected dataset. The objective of these models is to predict the input size of each function based on the input size of the entire DAG. The features used for training include the DAG input size and the function name mapping (represented as the index of the function in the parallel stage), while the target label is the function's input size. We create and train these models for each DAG type, resulting in three distinct ML1 models corresponding to the three different DAG types within our profiler.

ML Models (ML2). These machine learning models are responsible for predicting CPU and memory utilization. The input to these models is the predicted function input size, which is obtained from the ML1 models. The output of ML2 models reflects the monitored CPU and memory utilization for each function. In addition to being specific to the DAG type, we train separate models for CPU and memory utilization, resulting in six versions of such models.

The rationale behind this two-step prediction approach stems from the challenge of simultaneously predicting the resource demand of all functions within a parallel stage of a DAG at the time of the DAG request arrival, without interfering with its execution flow. By using a two-step process, we first predict the input size of each function (via ML1 models), which is more directly influenced by the input size of the entire DAG. This prediction can be made quickly and efficiently without delaying the execution.

Once the function input sizes are predicted, the second step (ML2 models) uses this information to estimate the resource utilization (CPU and memory) of each function. This method allows us to avoid real-time interference with DAG execution since the resource predictions are pre-computed based on input sizes, thereby allowing efficient resource allocation at the moment of DAG request arrival. The separation of input size prediction from resource utilization ensures that the models are optimized for each specific task, leading to more accurate predictions and minimizing disruption to the DAG execution flow.

Table 3.2. Resource Allocation: CPU Cores and Memory Settings

Resource	**Minimum**	**Maximum**	**Increment**
CPU Cores	1 core	8 cores	1 core
Memory	128 MB	8192 MB (8 GB)	128 MB

Why Regression and Not Classification?

We formulate our resource prediction problem as a regression task rather than a classification problem due to the continuous nature of resource demands and the observed behavior of function executions. Our resource monitoring agent has shown that functions often do not fully utilize the total reported demand. For example, a function may report Maximum CPU usage as 2.78 cores, indicating that it did not fully consume the available CPU time for 3 cores during its execution. This is effectively captured by regression models.

A classification model would necessitate the creation of a large number of resource categories to represent all possible CPU and memory allocations. This would increase the model's complexity and reduce its ability to generalize. Additionally, predefined classes would impose artificial boundaries that do not align with the actual, varying resource utilization observed during function executions. On the other hand, a regression model is better suited to predict continuous resource values, allowing for greater flexibility and precision in capturing the true resource needs of each function. We formulate the regression problem as described in Appendix B.

To ensure that the predictions from the regression model conform to valid system allocations, we apply a rounding mechanism (Allocation Normalizer) that adjusts the predicted values to the nearest allowed resource allocation interval. This approach combines the flexibility of regression with the practical constraints of the system, leading to more accurate and feasible resource allocations.

3.5.2. Inference Pipeline

This component is part of our prediction engine that will be used in real-time invocations for your system. Its role is to determine the actual demand of each function within a DAG by only peeking into the size of the input. Figure 3.5 an overall execution flow of this component. In this section, we explain the flow in more detail.

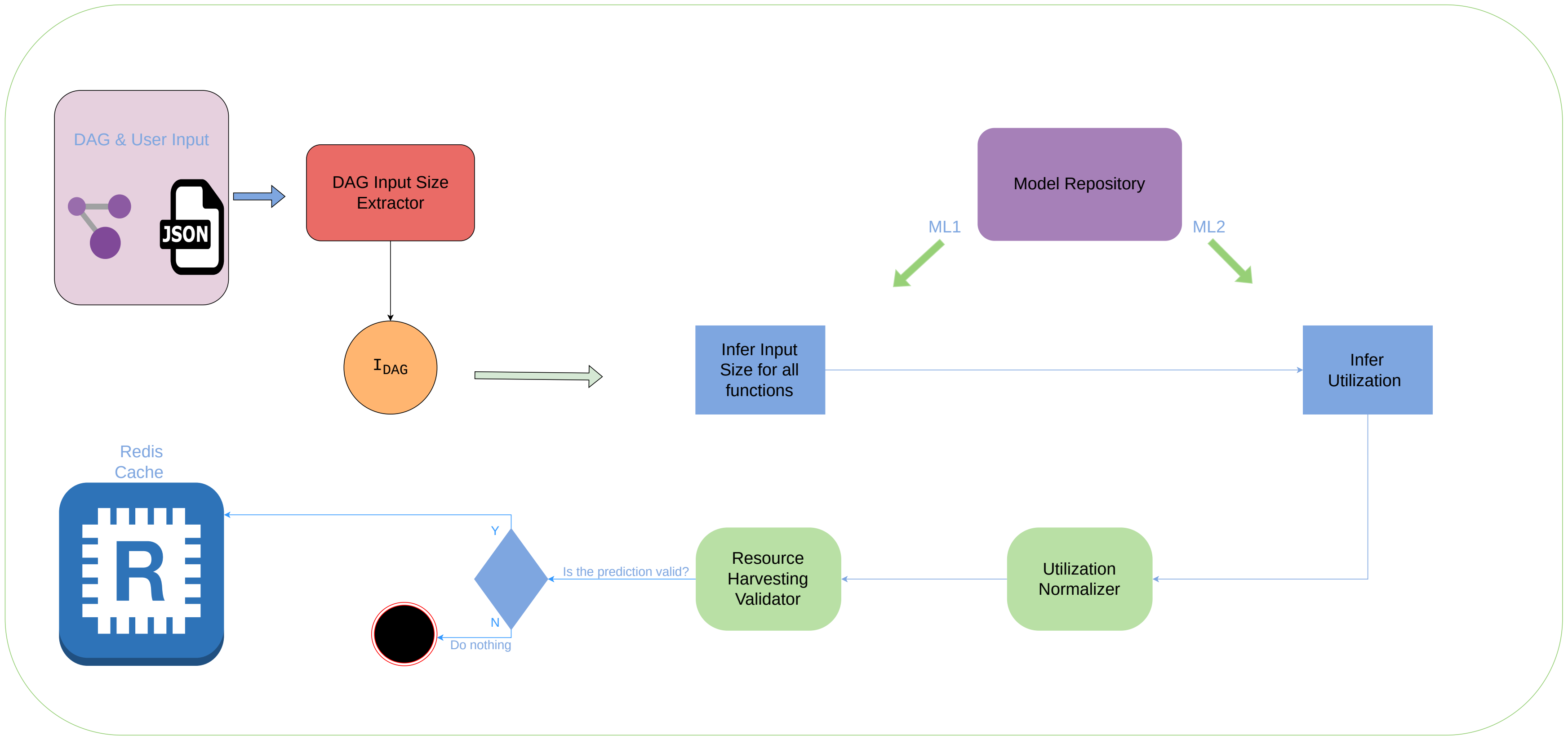


Figure 3.5. Resource Prediction Pipeline

Upon the arrival of a user request, the pipeline begins by extracting the size of the user-provided input for the entire DAG. This input size is a critical feature that is immediately forwarded to the first set of machine learning models, denoted as ML1. The ML1 models are trained to predict the input size for each individual function within the DAG based on the overall input size of the DAG. This prediction process takes into account the structure of the DAG, as well as the specific roles of the functions, including whether they are part of parallel stages or not.

The output from the ML1 models, which consists of the predicted input sizes for all functions, is then passed to the second set of machine learning models, referred to as ML2. These models are responsible for predicting the CPU and memory demand of each function. The ML2 models use the function input sizes as key features, along with additional function-specific characteristics such as stage information.

In this two-step pipeline, the resource demands of all functions within a parallel stage can be predicted simultaneously at the time of DAG request arrival. This design is highly efficient, as it avoids the need to gather real-time resource usage data during execution, which would interfere with the DAG's execution flow. Instead, the predictions are made based on pre-trained models, allowing resource allocation decisions to be made quickly and accurately. After prediction, the output of the ML2 models is rounded to the nearest valid allocation by the Utilization Normalizer component to ensure compliance with the system's resource allocation constraints. This process allows the system to optimize resource usage while respecting the varying demand patterns of individual functions.

Utilization Normalizer. In our inference pipeline, this component ensures that predicted resource values conform to the predefined allocation intervals. Since the predictions generated by our regression models can yield non-discrete values that do not match the allowed resource allocations, the utilization Normalizer rounds each prediction to the nearest valid allocation higher than the predicted value. For instance, if the predicted CPU requirement is 2.78 cores, it will be rounded up to 3 cores, and if the predicted memory requirement is 234.32 MB, it will be rounded up to 256 MB. This process ensures that all predictions are feasible and aligned with the system's resource allocation rules, avoiding the assignment of invalid resource configurations.

Harvesting Validator. The Harvesting Validator is a critical component in the resource allocation process for parallel stages of a DAG. Its primary function is to ensure that resource harvesting is performed within the constraints of the available resources allocated to a parallel stage. This component operates by validating the predicted CPU and memory demands of each function within a parallel stage against the total number of resources allocated for that stage.

In a typical scenario, the number of available resource units (e.g., CPU cores) for a parallel stage is predefined, and each function in the stage is assigned a portion of these resources based on the predictions from the ML2 models. For instance, consider a parallel stage with 4 available resource units and two functions to execute. If the predicted resource allocation for these two functions is a 3-1 split (i.e., 3 units for one function and 1 unit for the other), the Harvesting Validator deems this a valid allocation, as the total allocation fits within the 4 available units. However, if the prediction results in a 5-1 split, this allocation exceeds the available 4 units and is therefore deemed invalid.

An additional aspect of the validation process involves determining whether the predicted resource distribution represents an optimal harvesting strategy. For example, if the prediction suggests a 6-2 split for two functions but only 4 units are available, the validator will calculate the ratio between the resource demands. If this ratio aligns with the overall structure of the DAG and the observed function behavior, the validator can still determine that this represents an effective harvesting strategy. The goal is to ensure that resource harvesting maximizes utilization without exceeding available resources or disrupting the execution of other functions.

Once the Harvesting Validator confirms a valid harvesting opportunity, it writes the CPU and memory demands of each function to a Redis key-value store. This key-value store is accessed by the container proxy inside the invoker, which dynamically reassigns the appropriate resources to each function during execution. By caching these resource demands, the system enables efficient and dynamic resource management, allowing for optimized execution of the parallel stage. This would yield a significant reduction in the execution time of the whole parallel stage, leading to a reduction in the latency of the user's request.

If the validation fails, meaning the predicted resource allocation exceeds the available resources or is otherwise deemed suboptimal, the demands of the affected functions are not written to Redis. In such cases, the system either falls back to the user-specified allocation to ensure that the DAG proceeds without interruptions. This mechanism ensures that resource harvesting is carried out in a controlled manner, maintaining the balance between efficient resource use and the integrity of the DAG's execution flow.

3.6. System Overview

To address the challenges discussed in this section, we design a system for serving serverless DAG workloads. Figure 3.6 depicts the overall architecture of our system. Our solution is built on top of an existing serverless system called OpenWhisk.

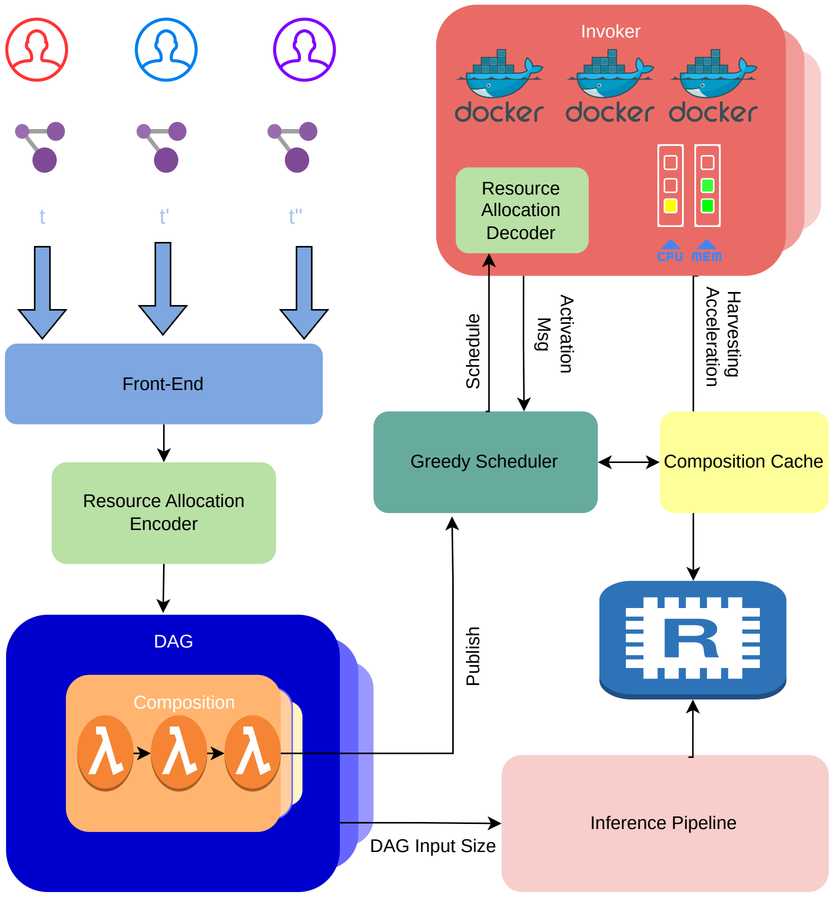


Figure 3.6. System Architecture

Job Submission. In our system, users submit DAG job requests that define the structure and resource requirements of their DAG workloads. Each job submission includes the DAG structure, user-defined CPU and memory allocations, an input file for the DAG, and the code for each function within the DAG in a compressed zip format. This flexible interface allows users to submit jobs at various times, supporting different DAG configurations and input data. The system is designed to handle multiple, diverse requests concurrently, ensuring that each DAG is executed efficiently on the serverless back-end according to the specified resource requirements. This dynamic job submission process enables scalable execution of complex workloads with varying computational demands.

Front End. The front end of our system is implemented in Python and serves as the interface for managing incoming DAG job requests. Upon receiving a request, the front end concurrently handles the life-cycle of each job, tracking the status of the DAG functions throughout execution. It monitors resource usage, such as CPU and memory, for every function within the DAG, ensuring that resource consumption is recorded for later stages of the workflow. Additionally, this component is responsible for formatting the user-defined DAG descriptions, resource allocations, and input data so that they seamlessly integrate with the system's prediction and invocation components. Given that our system leverages OpenWhisk for serverless execution, the front end also communicates with the OpenWhisk backend via its CLI, facilitating both the monitoring and execution of jobs. This component plays a significant role in the dynamic management and orchestration of serverless DAG workflows. **Resource Allocation Encoder.** In our system, OpenWhisk natively supports memory specification for functions but does not allow users to define the number of CPU cores directly. To address this limitation, we introduce a resource allocation encoding mechanism that captures both the user's desired memory and CPU allocations. This encoding allows us to interact with OpenWhisk while preserving the user-defined CPU and memory requirements.

The encoding is defined as follows:

$$\text{encoded\_allocation} = (\text{userDefinedCpu} - 1) \times \text{MaxMemoryAllocation} + \text{userDefinedMemory}$$

where MaxMemoryAllocation is set to 8 GB, which corresponds to the maximum memory allocation permitted in our system. This encoding mechanism ensures that both CPU and memory specifications are passed to OpenWhisk, enabling efficient resource management for each function within the DAG. By incorporating both resource dimensions, the system can more effectively allocate resources based on user-defined constraints, optimizing the execution of serverless functions.

DAG Orchestration. In our system, we orchestrate user-defined DAGs by leveraging the Composer model, which integrates seamlessly with our OpenWhisk back-end. Each DAG can consist of multiple compositions, depending on its structure and the width of the DAG. A composition is a sequence of serverless functions that represent the user logic for each node in the DAG. This orchestration approach allows us to utilize the data dependencies between functions, optimizing their placement across different invokers.

Before any function within the DAG is submitted to the load balancer for placement, a request is sent to the Inference Pipeline to generate a harvesting decision for all the functions in the parallel stage of the DAG. This ensures that the resources are allocated efficiently based on predicted function demands. Once the harvesting decision is made, functions are published to the load balancer according to the composition structure of the DAG. This process ensures that the functions are executed with proper resource allocation while adhering to the dependencies defined by the user's DAG.

Inference Pipeline. This component, as depicted in Figure 3.5, is responsible for predicting the resource demand of all functions within the parallel stage of a DAG. After predicting the resource demand, it applies a harvesting check to determine if the reallocation of resources is valid within the constraints of the parallel stage.

If the reallocation is deemed valid, the computed CPU and memory demands are written to a Redis cache. These cached values are then used during the generation of Docker containers in the invoker, enabling dynamic reallocation of resources. However, if the harvesting decision is found to be invalid, the system will revert to the user-specified resource allocation for function execution, ensuring the correct and efficient use of system resources during the execution of the DAG.

Composition Aware Greedy Scheduler. In our work, we leverage the Composer model [2] provided by Apache OpenWhisk. The Composer model enables developers to define workflows in which multiple functions are linked together, representing both the control flow and data dependencies between them. Although the Composer model offers flexibility in building these compositions, current implementations often fail to optimize the natural grouping of functions within a single composition.

To address this gap, we introduce a composition-aware scheduling algorithm that optimizes the placement of functions in a serverless environment. Our algorithm is the first of its kind to consider the structural properties of a DAG composition when assigning function invocations to compute resources (invokers). Specifically, the algorithm greedily assigns functions that belong to the same composition to the same invoker whenever possible. This placement strategy minimizes data communication overhead between functions within a composition, as intra-composition data transfers are significantly faster when the functions are co-located on the same invoker, compared to when data must be transferred across invokers.

The primary motivation for this approach arises from the high cost of cross-invoker communication, particularly in data-intensive workflows. When parallel functions in a composition exchange large datasets or maintain frequent data dependencies, network communication between separate invokers introduces latency and additional resource contention. By co-locating these functions on the same invoker, our algorithm reduces the number of network-bound data transfers, leading to faster execution times and improved resource utilization. Next we introduce our proposed algorithm for scheduling.

3.6.1. GreedyJSQ Load Balancer Design

Our scheduling algorithm is designed for efficiently placing functions from the same composition on a single invoker in our system. The primary motivation behind this design is to minimize the communication overhead and reduce latency between functions in the same composition by ensuring that they execute on the same invoker, reducing the data transfer cost.

In traditional load balancers, functions from the same sequence might be placed on different invokers, causing significant latency as data needs to be transferred between the invokers for each step in the composition. Our approach, however, is *composition-aware*, meaning that for functions belonging to the same composition, we greedily place them on the same invoker. This minimizes the data transmission latency by eliminating the need for inter-invoker communication during function execution within the same composition. Below are the main concepts in our scheduler:

1. **JSQ for First Function Placement**: For the first function in a composition, the load balancer selects the invoker with the shortest queue using the JSQ algorithm.

This ensures that the initial function is placed optimally based on the current load across the available invokers.

2. **Greedy Caching for Subsequent Functions**: Once the first function is placed, the invoker is cached and all subsequent functions within the same composition are placed on the same invoker. This reduces the data communication overhead between functions and improves overall execution latency.

Algorithm2 shows the pseudo-code for our invoker selection mechanism. We implement this algorithm in the load balancer component within OpenWhisk.

Algorithm 2 GreedyJSQ Load Balancer Algorithm

```
1: Input: Set of invokers $I$, Action $a$, Activation Message $m$
2: Output: Assigned invoker $invoker$
3: Initialize CompositionCache ← empty map
4: Initialize invokerQueues ← list of current invoker loads
5: $cachedInvoker \leftarrow$ LOOKUP(CompositionCache, $m.parentComposition$)
6: if $cachedInvoker$ is not None and $cachedInvoker$ is healthy then
7:     return $cachedInvoker$
8: else
9:     $invoker \leftarrow$ JSQ($I$, invokerQueues)
10:    if $invoker$ is found then
11:        UPDATE(CompositionCache, $m.parentComposition$, $invoker$)
12:        return $invoker$
13:    else
14:        Wait for NEEDWORKSIGNAL(invoker)
15:        $invoker \leftarrow$ FIRSTINVOKERTOSENDSIGNAL($I$)
16:        if $invoker$ is found then
17:            UPDATE(CompositionCache, $m.parentComposition$, $invoker$)
18:            return $invoker$
19:        else
20:            Throw Error:  No available invokers
21:        end if
22:    end if
23: end if
```

Detailed Explanation:

- **Input**: The algorithm takes as input the set of available invokers I, the action a that is being invoked, and the activation message m containing the invocation details.

- **Caching Mechanism**: The algorithm first checks the composition cache, which holds mappings from a composition's cause to the previously assigned invoker. If the cause is already present and the invoker is still healthy, the cached invoker is used.

- **JSQ for First Placement**: If the cause is not cached or the cached invoker is unavailable, the JSQ logic is applied. The invoker with the shortest queue is selected based on the current system load, and it is stored in the cache for future requests with the same cause.

- **Greedy Placement**: Once the invoker is selected for the first function in the composition, all subsequent functions are placed on the same invoker, ensuring minimal data transfer latency between functions.

- **Error Handling**: If no invokers are available, the algorithm raises an error indicating that no invokers are capable of handling the request.

The composition-aware nature of this algorithm ensures that the latency between function executions is minimized by eliminating inter-invoker communication. By placing functions from the same composition on the same invoker, the algorithm significantly reduces the time spent in data transmission between functions, thus improving the overall performance and responsiveness of the system.

Resource Allocation Decoder. The Resource Allocation Decoder is inside the Container Proxy, which wraps Docker containers to monitor and manage their execution in OpenWhisk. This decoder is responsible for interpreting the encoded resource allocations (CPU and memory) that were previously set by the Resource Allocation Encoder.

To support both CPU and memory allocations during container initialization, we extend the Docker API within OpenWhisk to accept these values. When the encoded resource allocation is decoded, or the CPU and memory values are retrieved from Redis, these values are passed to the Docker container. The Container Proxy then initializes the container with the specified resource constraints, ensuring that the user's code and runtime are properly injected into the container for execution. This modification enables more granular control over resource allocation, allowing both CPU and memory to be dynamically adjusted based on the predicted or user-specified resource requirements.

Modifying the Semaphore Lock for CPU and Memory Permits

In the original OpenWhisk implementation, the resource management system was primarily designed to manage memory resources, allowing for the allocation and tracking of memory usage across various function invocations. However, serverless environments often involve significant CPU demands, making it essential to introduce CPU resource management alongside memory management. To address this need, we extended the existing semaphore mechanism to manage both CPU and memory resources, ensuring more comprehensive and efficient resource allocation.

Original Semaphore Design. The initial design of the semaphore in OpenWhisk focused solely on managing memory resources. The semaphore was responsible for maintaining the available memory permits, and when a function required memory, the system would allocate the necessary resources. If sufficient memory permits were not available, function execution would be delayed until enough memory was freed. While effective for managing memory, this design lacked the capacity to track and allocate CPU resources, an increasingly critical aspect of serverless computing.

CPU Resource Management. To enable CPU resource tracking, the semaphore design was extended to accommodate CPU resources alongside memory permits. This enhancement allows the system to maintain independent control over both types of resources, enforcing allocation limits in a more granular and efficient manner. The system now tracks available CPU and memory units simultaneously, ensuring that both resources are allocated based on demand while avoiding over-allocation.

Concurrent Resource Allocation. In our design, when a function requests resources, both CPU and memory demands are assessed simultaneously. The system verifies whether the available resources meet the requested demand. If both CPU and memory resources are available, the function is allocated the required resources; otherwise, the request is delayed until resources become available.

Resource Release and Reallocation. Once a function completes its execution, both CPU and memory resources are released and made available for subsequent functions. The system efficiently tracks the release of resources, ensuring that they are promptly reallocated to other functions that are awaiting execution.

3.6.2. Function Execution

Once an action(functions in OpenWhisk) is being allocated a container to execute its code, we retrieve its resource demand from the Redis cache. This happens in the container proxy component which monitors the lifetime of docker containers. The retrieved value is then used as arguments for the Docker API to allocate resources to the function. If there was no value retrieved from Redis, meaning that there is no harvesting opportunity, the user's default allocation is used for the invocation.

Chapter 4. Evaluation

We implement our system prototype with 3k lines of code in Python for the front-end and our prediction engine. We also implement 3k lines of code in Scala to modify the internals of existing OpenWhisk. We also integrate OpenWhisk's Composer model with our system to orchestrate DAGs.

4.1. Evaluation Metrics

We utilize metrics previously used by the community to demonstrate the benefits of our system and its potential weaknesses.

4.1.1. Prediction Error

Prediction error is defined as the difference between the predicted value and the actual demand for the CPU or RAM. It is calculated as follows:

$$\text{Prediction Error} = |\text{Predicted Value} - \text{Actual Value}|$$

Where:

- Predicted Value is the predicted CPU or RAM demand.
- Actual Value is the actual CPU or RAM demand.

4.1.2. Model Accuracy

Since our models are regression-based, we define a prediction as accurate if the sum of the predicted value and its corresponding prediction error falls below the smallest possible allocation that is greater than or equal to the predicted value in our system.

Formally, let:

- P be the predicted CPU or RAM demand,
- ΔP be the prediction error,

- $A(P)$ be the smallest possible allocation that is greater than or equal to P,

The prediction is considered accurate if:

$$P + \Delta P < A(P)$$

For example, if the predicted CPU usage is 2.6 and the prediction error is 0.3, the sum $2.6 + 0.3 = 2.9$ is smaller than the next available allocation of 3 cores, making it an accurate prediction. However, if the prediction error is 0.41, the sum $2.6 + 0.41 = 3.01$ falls into the range of 4 cores, which is not accurate according to our system's allocation policy.

4.1.3. Speedup

The speedup metric quantifies the reduction in end-to-end (E2E) latency for each invocation or for each parallel stage of a DAG due to our harvesting mechanism. It is formally defined as:

$$\text{Speedup} = \frac{L_{\text{baseline}} - L_{\text{ours}}}{L_{\text{baseline}}}$$

Where:

- L_{baseline} is the latency for each invocation in the baseline system (OpenWhisk) without harvesting.
- L_{ours} is the latency for each invocation in our system with harvesting.

This metric highlights the effectiveness of our harvesting mechanism. It is inspired by Libra [21].

4.1.4. Resource Utilization

The resource utilization metric measures how efficiently our system uses the available resources. It is defined as:

$$\text{Utilization} = \frac{R_{\text{utilized}}}{R_{\text{total}}}$$

Where:

- R_{utilized} is the total amount of resources utilized by the system during the experiment.

- R_{total} is the total amount of available resources during the experiment.

This metric indicates the improvement in resource utilization provided by our system. Like the speedup metric, it is inspired by Libra [21].

4.2. Evaluation Setup

We evaluate our system on a single-node cluster. The cluster has three nodes, including a single node for hosting both the controller and Python client for invocation, and 2 nodes that host an invoker with 8GB of memory and 8 CPU cores each. **Workload.** As there are no specific DAG traces published by major serverless providers, we utilize the 2021 Azure Function traces [4] for a single workflow consisting of a total of 180 DAG invocations. We utilize the trace through grouping the functions by their `<application_id>` in the trace and construct workloads for the corresponding 180 applications in Azure traces. Throughout the timeline of these invocations, over 900 individual serverless functions are executed. To evaluate our system under varying load conditions, we classify the traces into three categories: low, medium, and high arrival patterns. Table 4.1 demonstrates the arrival pattern for each load setting. To determine the input file for each invocation, we randomly select files with different input sizes for the first time and preserve the order to be consistent for all evaluation settings.

Table 4.1. Arrival patterns for different load settings

Arrival Pattern	Number of DAG Invocations	Requests Per Minute
Low	180	15
Medium	180	30
High	180	60

4.3. Model Analysis

We evaluate 6 ML models for regression tasks regarding input size prediction, CPU utilization prediction, and memory utilization. We build separate models for each of our DAG workflows. Each DAG workflow is evaluated across 200 different input sizes, and we apply an 80-20 train-test split to assess model performance. Models used for evaluation are listed as below:

- **RandomForest**: A tree-based ensemble method that builds multiple decision trees and averages their predictions to improve accuracy and reduce overfitting.
- **XGBoost**: An efficient gradient boosting algorithm that iteratively builds trees by focusing on correcting previous errors, commonly used for its high performance.
- **SVR (Support Vector Regression)**: A regression model based on the support vector machine algorithm, which finds the optimal hyperplane in high-dimensional space to predict continuous values.
- **NN (Neural Network)**: A model inspired by biological neural networks, consisting of layers of interconnected nodes that can capture complex patterns in data through non-linear transformations. We use a 4-layer fully connected network for this model.
- **DecisionTree**: A simple, interpretable model that splits the data into subsets based on feature values, following a tree-like structure for predictions.
- **LinearRegression**: A basic regression model that assumes a linear relationship between input features and the target variable, making it easy to interpret but limited in handling non-linearity.

To be concise, we present the performance evaluation of our models for the VideoAnalytics DAG workflow. Evaluation figures regarding the two other workflows can be found in

Appendix C.

Figure 4.1 shows the error rates for input size prediction across all models. The goal of this figure is to demonstrate the accuracy of each model in predicting the size of input data for different functions within the VideoAnalytics DAG. We observe that the DecisionTree model and RandomForest model result in the lowest prediction error. SVR and LinearRegression are omitted from this figure due to their relatively high prediction error.

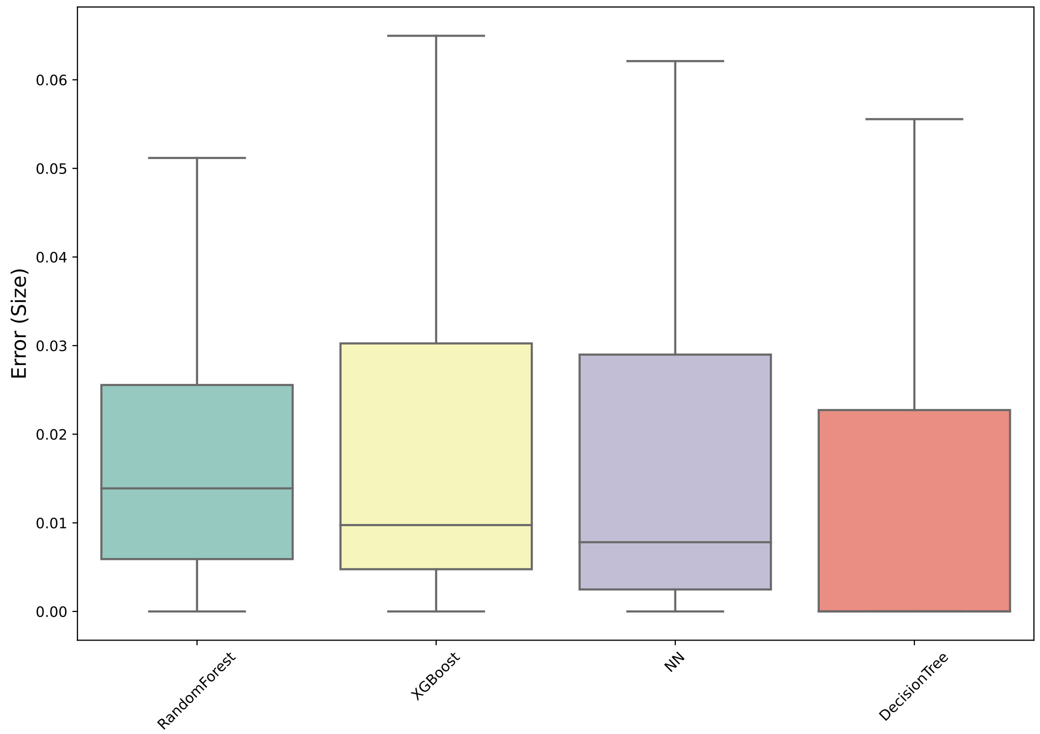


Figure 4.1. Input Size Models for VideoAnalytics

Figure 4.2 presents the CPU utilization prediction error for the same models. This figure evaluates how well each model predicts CPU demand based on the input size. The results highlight that the RandomForest model is performing the most accurately in predicting CPU usage. The SVR model is again demonstrating high prediction accuracy and is left out from the figure.

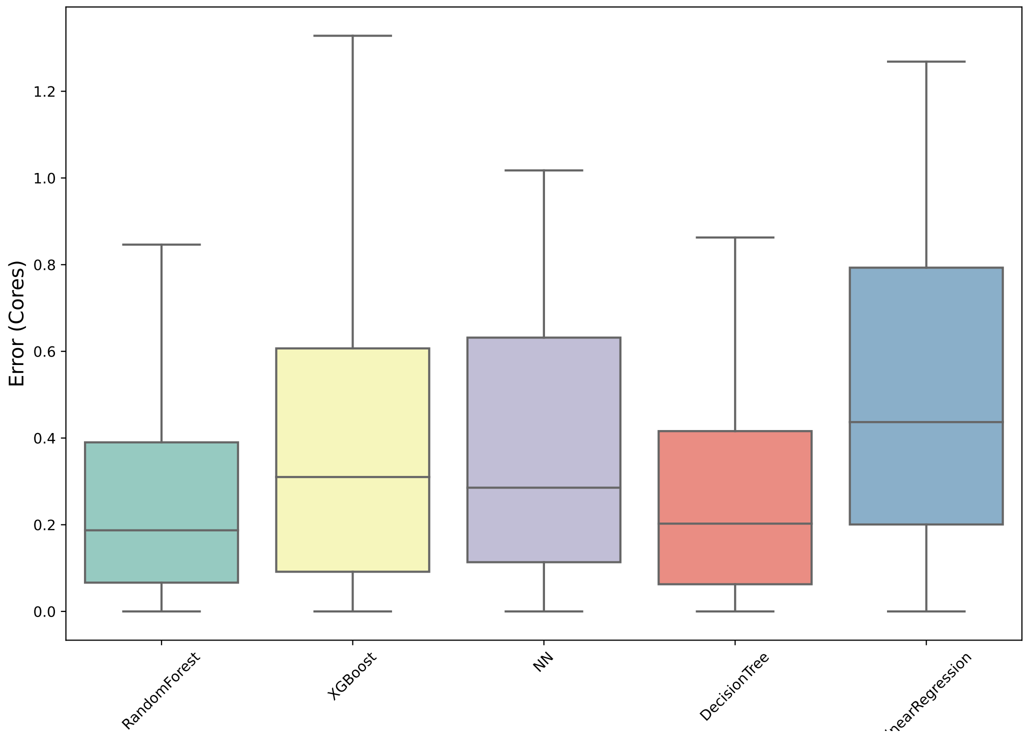


Figure 4.2. CPU Utilization Models for VideoAnalytics

In Figure 4.3, we show the memory utilization prediction errors. Similar to CPU utilization, this figure focuses on how accurately the models can estimate the memory demand for each function in the DAG. The DecisionTree model can be observed, with the lowest prediction error trend, although the average prediction error for RandomForest is still the lowest.

Lastly, Figure 4.4 provides an overall view of the prediction accuracy for each model across all three tasks (input size, CPU utilization, and memory utilization). This figure serves as a summary, illustrating the general performance trends, and showing that

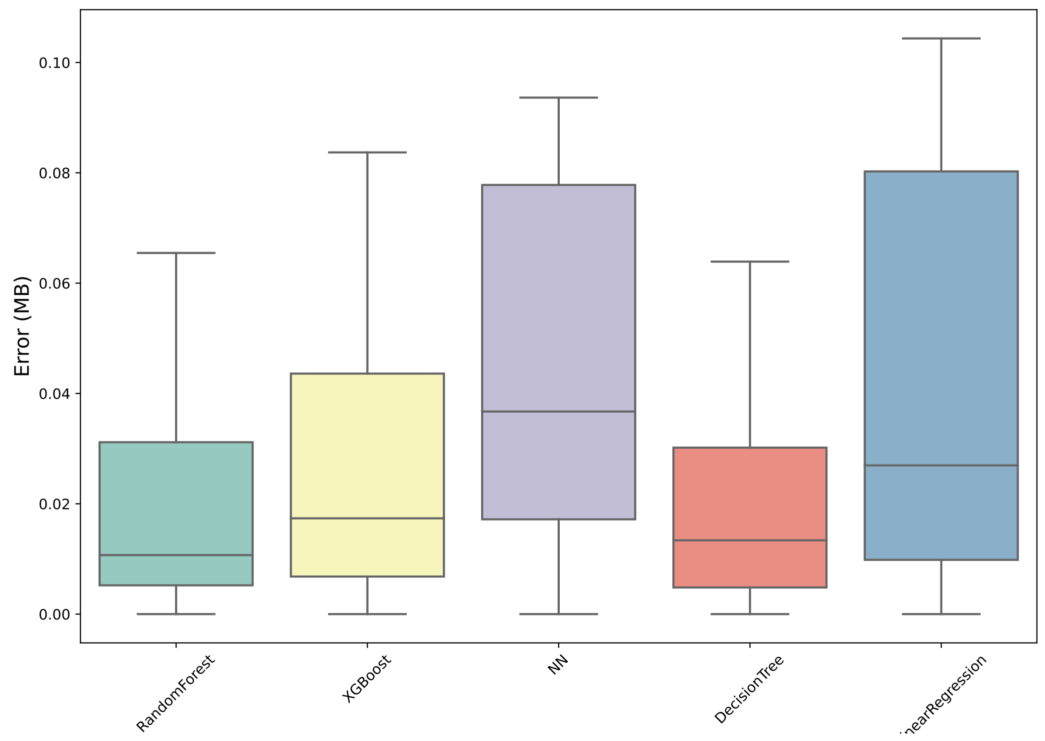


Figure 4.3. Memory Utilization Models for VideoAnalytics

the RandomForest model has the highest accuracy amongst our models with almost 99 percent accuracy in the prediction of CPU and memory utilization.

We opt for this model in our prediction pipeline due to its consistent performance across all workflows. We build our prediction pipeline with this model as the ML model for all input size, CPU utilization, and Memory utilization models across all workflows.

4.4. Harvesting Evaluation

Although there have been multiple works proposed to apply resource harvesting for serverless functions, none of them can be applied to complex DAG workflows. This makes our work the first to propose such a mechanism for DAG workflows. Therefore, we only compare the efficiency of our system with a variant of our own which disables harvesting.

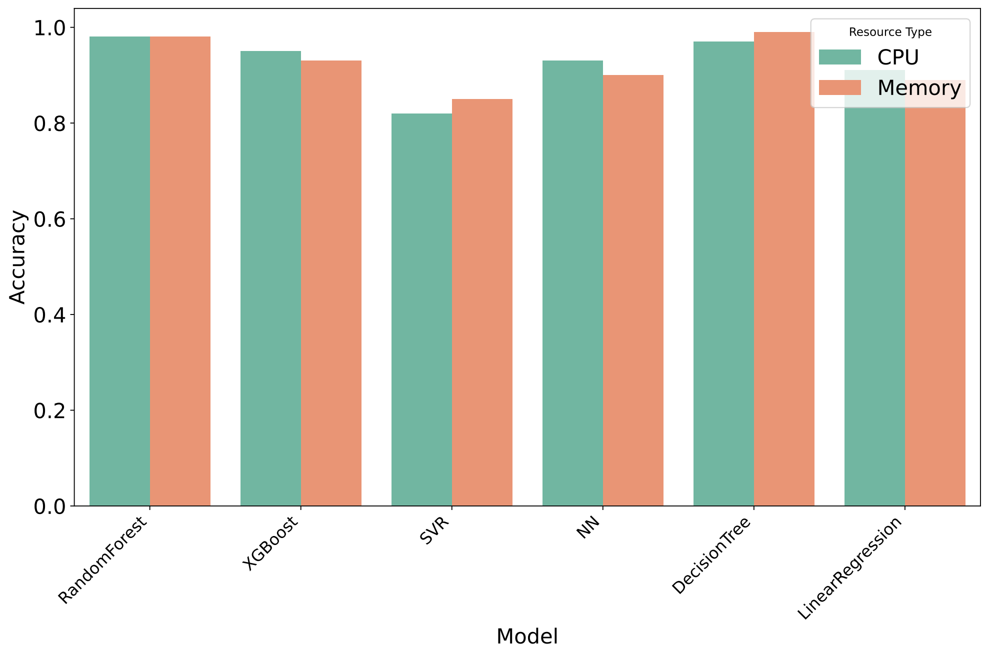
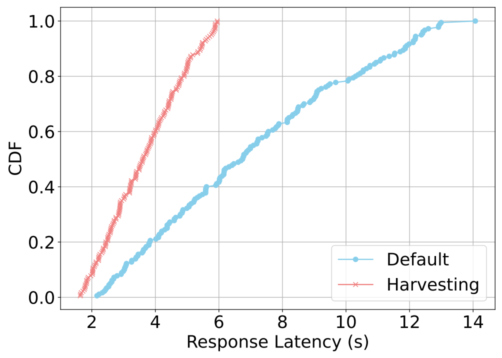
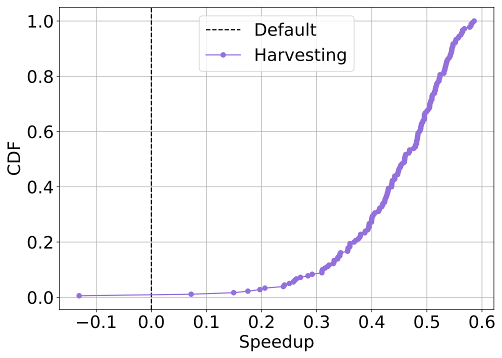


Figure 4.4. Prediction Accuracy for VideoAnalytics

We refer to this variant as default in our experiments. Figure 4.5a shows that the P_{99} latency of requests in our workload is reduced by 57% compared to the default scenario. This significant reduction in execution time is achieved by harvesting resources within a single DAG request, demonstrating the importance of such harvesting opportunities. The P_{50} speed up under low workload is 50% according to our experiments.



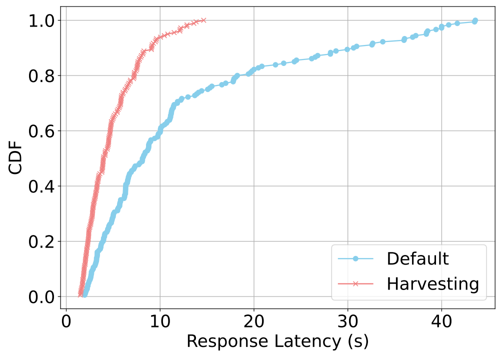
(a) CDF of DAG response latency



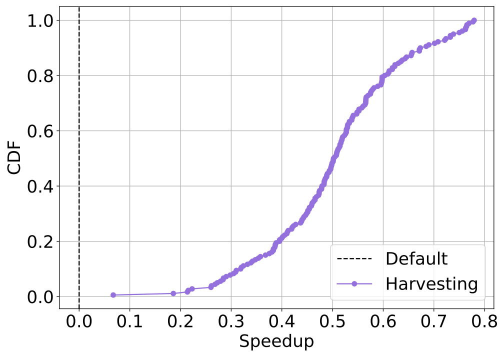
(b) CDF of Speedup

Figure 4.5. Latency and Speedup CDF for Low trace

We also evaluate the performance of our system under the medium workload. As shown in Figure 4.6, the improvement under the medium workload is even more pronounced, with a 75% reduction in P_{99} latency. This greater reduction can be attributed to the increased number of parallel function executions in the medium workload, which amplifies the effect of resource harvesting. As the workload intensity rises, the system's ability to dynamically reallocate resources becomes more critical, optimizing resource usage across parallel functions. This reduces contention and shortens the execution time for high-latency requests. We also observe during the medium trace, we can achieve up to 80% speedup for certain DAG requests, which is 25% higher than our low arrival workload.



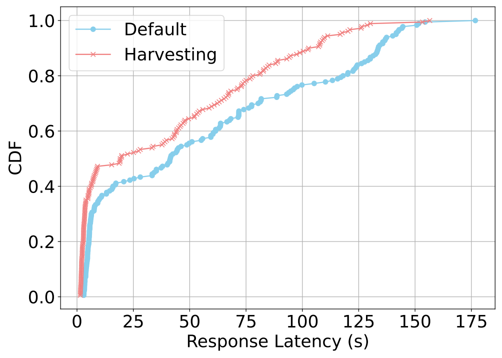
(a) CDF of DAG response latency



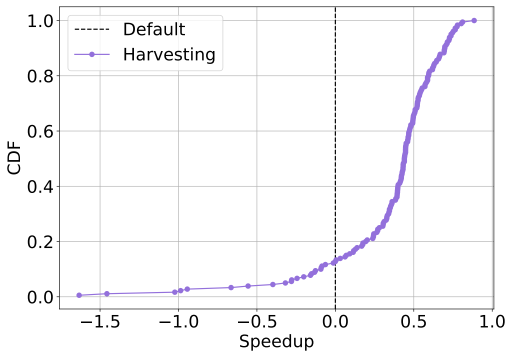
(b) CDF of Speedup

Figure 4.6. Latency and Speedup CDF for Medium trace

By applying the high workload to our system, we find out that the improvement immediately starts to diminish. Figure 4.7a shows the response latency and speed up during the high workload. We observe that there is less than 25% improvement for the P_{99} of the requests. We also do not observe any form of speed up for almost 10% of requests. This reduced improvement can be attributed to the system reaching a saturation point in resource availability. Under high workload conditions, the number of concurrent DAG invocations increases significantly, leading to high contention for CPU and memory resources. As a result, the system's ability to efficiently harvest and reallocate resources is constrained.



(a) CDF of DAG response latency



(b) CDF of Speedup

Figure 4.7. Latency and Speedup CDF for High trace

To attribute the observed latency improvement to functions in the parallel stage of the DAG, we monitor the E2E time for the parallel stage of each request. We observe the same pattern observed in response latency, which means the improvement is solely due to our optimizations in the parallel stage. To prevent repetition, we have included the results for the parallel stage in Appendix C. We also analyze the speed-up of request latency under different scenarios to evaluate the performance under different workloads. Table 4.2 reports a summary of speed-up in all scenarios. Our system achieves 44% speed-up on average by harvesting resources among parallel functions in DAG.

We monitor and report the resource utilization of our system under the low workload to make sure we capture the improvement in resource utilization of our system without the effect of the saturation point. This helps us to solely demonstrate the maximum

Table 4.2. Summary of Speedup for Different Workloads

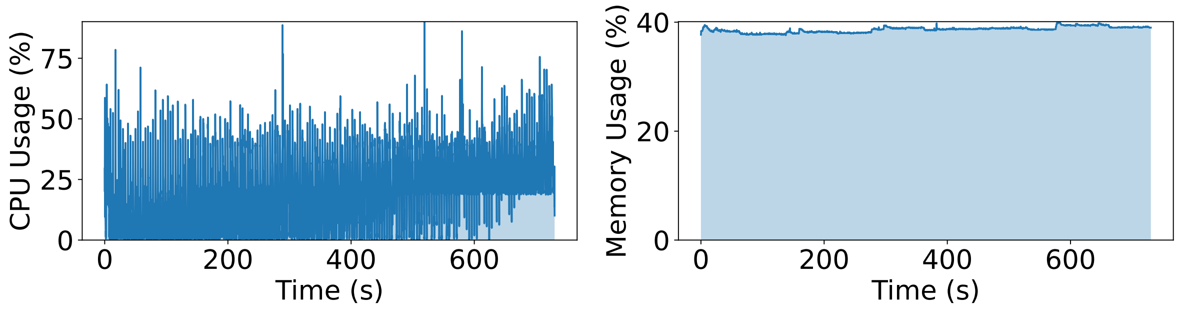
Workload	**Max Speedup**	**Min Speedup**	**Average Speedup**	**Median Speedup**
Low	0.5858	-0.1310	0.4411	0.4599
Medium	0.7795	0.0673	0.5033	0.5033
High	0.8848	-1.6333	0.3529	0.4404

improvement of resource utilization. Figure 4.8a shows the CPU utilization in default mode. We observe that during the execution of workload the CPU utilization fluctuates between 0% to 60% with an average utilization of 20%. Figure 4.8a also reports the memory utilization below 40% with minimal fluctuation. This is because we do not modify the default container initialization of OpenWhisk and by the arrival of requests as well as the end of the keep-alive period for previous actions, the memory does not change substantially. Figure 4.8b shows the same information in the case of running the workload with harvesting enabled. We observe a 200% increment in CPU utilization as well as a 30% increment in memory utilization compared to the default scenario. This increase in resource utilization is only due to harvesting. in the parallel stage of DAG requests.

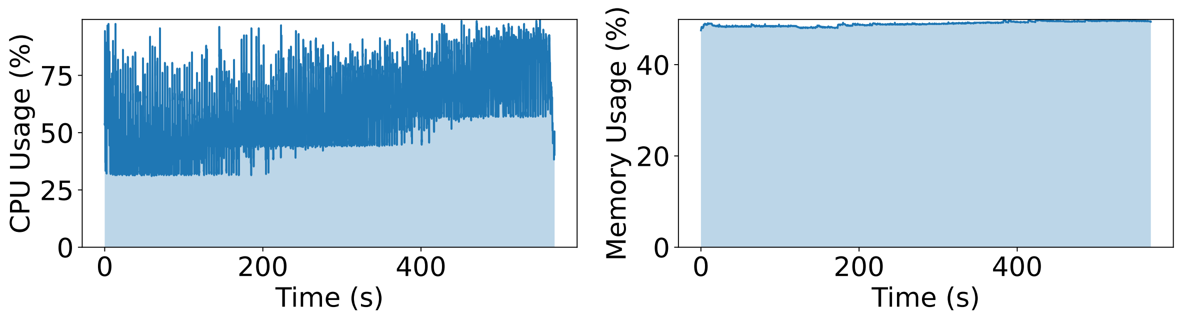
We report the average CPU and Memory utilization in Table 4.3. The memory utilization increase is significantly lower than CPU utilization increase in our experiments. This happens because we only evaluate one of our workflows(ParallelAES) during our workload evaluation due to resource limitation. This workload is more CPU-intensive and improvement is more evident in terms of CPU usage.

Table 4.3. Summary of Average CPU and Memory Utilization

Scenario	**Average CPU Utilization (%)**	**Average Memory Utilization (%)**
Harvesting	59.94	48.96
Default	19.60	38.66



(a) Resource utilization (Default)



(b) Resource Utilization (Harvesting)

Figure 4.8. Resource Utilization

4.5. Scheduling Evaluation

Existing serverless platforms typically treat function invocations as independent units, disregarding the interdependencies present within compositions. To provide solid baselines for comparison with our composition-aware algorithm. The scheduling algorithms we integrate and evaluate are as follows: **Join-Shortest-Queue (JSQ).** The JSQ algorithm selects the invoker with the fewest queued functions at the time of invocation. By distributing functions based on the current load of each invoker, JSQ aims to balance the workload evenly across the available resources, minimizing queue lengths and reducing potential bottlenecks in function execution. This algorithm is particularly effective in scenarios where invoker load is highly variable, as it dynamically adapts to the state of the system. We implement this by monitoring the in-flight invocation across all invokers and constructing a mapping of such invocations in the load balancer. For each invocation, the load balancer selects the invoker with the smallest queue in our mapping.

Round Robin. The Round Robin (RR) scheduling algorithm assigns incoming function invocations to invokers in a cyclic manner. This ensures that all invokers receive an equal number of function invocations over time, without considering the current load or state of the invokers. Round Robin is simple and efficient, especially in environments where function execution times are relatively uniform. However, it may lead to suboptimal resource utilization in cases where invoker workloads are imbalanced. We implement this algorithm by tracking which invoker was chosen for the previous invocation and iterating through healthy invokers in our system.

OpenWhisk's Default Hash-Based Placement. OpenWhisk's default scheduling algorithm uses a hash-based function placement strategy. This method maps function invocations to invokers based on a hash value derived from the function identifier. While this approach is efficient in distributing functions uniformly across invokers, it does not take into account the current load or resource availability, which can result in uneven distribution of workloads across invokers.

Greedy Round Robin (Greedy RR). We also introduce a variant of the Round Robin algorithm, called Greedy RR, which optimizes function placement within compositions. Greedy RR first uses the Round Robin approach to place the initial function of a composition on an invoker. For all subsequent functions within the same composition, our greedy scheduling strategy is applied, placing these functions on the same invoker as the first function whenever possible. This ensures that data-dependent functions are co-located, minimizing cross-invoker communication while still maintaining a simple and effective initial placement strategy through Round Robin.

We evaluate all three workloads while enabling different scheduling algorithms for function placement. Figure 4.9 shows the workload completion time for each scenario. We observe that our proposed GreedyJSQ algorithm results in lower completion time among all algorithms studied, with a 10% improvement over the default scheduling algorithm of OpenWhisk under the medium workload. The Round Robin variant of our algorithm comes in second due to its composition-aware scheduling process.

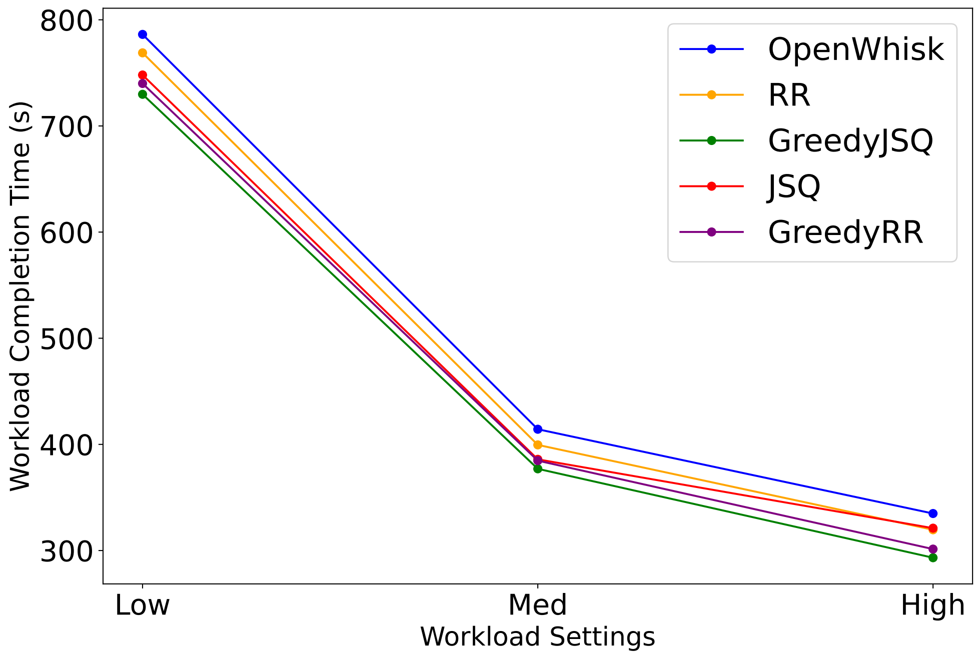


Figure 4.9. Workload Completion Time

We also analyze the P_{99} latency under all workloads to determine which scheduling algorithm can yield in lower end-to-end time for DAG requests. Figure 4.10 reports our findings. We find out that by increasing the workload, our proposed algorithm outperforms the traditional scheduling algorithms. This is because our algorithm is able to make more efficient function placement decisions when there are multiple DAG requests at the system. The composition cache helps our system quickly and efficiently place the function on an invoker where its input data is likely coming from, further reducing the latency. This is done by grouping functions from the same composition onto the same invoker, which leads to reduced communication delays. These optimizations allow our algorithm to maintain lower latencies, even under heavy workloads, compared to JSQ or RR approaches, which do not account for the composition structure of DAGs. We observe some abnormal behavior for the JSQ algorithm while increasing the workload. We argue that this abnormal increase in latency of JSQ could be due to network traffic on our cluster as we could not find any logic behind such behavior.

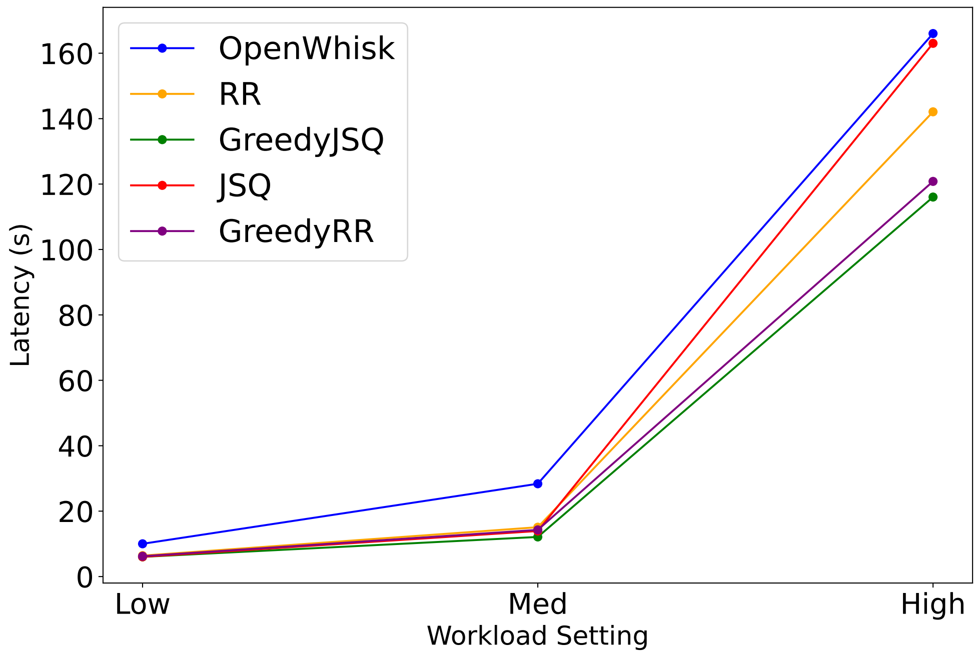


Figure 4.10. P_{99} Latency of Requests

Chapter 5. Limitations and Future Work

5.1. Limitations

We list the limitations of our system as below:

- **Scope of Resource Harvesting**: Our system is designed to harvest resources only among parallel functions within a single DAG request. It does not explore harvesting opportunities across different stages of a DAG or between functions from different DAGs running concurrently on the serverless back-end.

- **Focus on Input Size**: The current system exclusively investigates the effect of input size on function resource demands, as serverless platforms are restricted from accessing user code or input content for security reasons. However, other factors, such as the complexity of the user's code and the nature of the input content, may also influence resource demand.

- **Reliance on Machine Learning Predictions**: Our solution relies entirely on machine learning models to predict resource demand, with a fallback to user-defined allocations in case of inaccurate predictions. The inherent uncertainty of these models could reduce the system's reliability in production environments.

- **Lack of Scalability Testing**: Due to resource constraints, we were unable to conduct scalability experiments to evaluate the system's performance in large-scale, production-like environments, leaving its behavior under heavy workloads untested.

- **Experimental Setup**: Due to limitation of resources we only evaluate the ParallelAES DAG on real-world traces. The performance of other workflows during high load is not tested in our system.

- **Applicability Limited to Serverless DAGs**: The resource harvesting approach is tailored specifically for serverless DAGs and is not applicable to individual, non-composed serverless functions, as it relies on a Composer-aware architecture and prediction pipeline.

5.2. Future Work

There are several directions for future work to enhance the current system. One potential extension is to incorporate the timeliness of harvesting across different DAGs. By considering when resources can be harvested from functions across multiple DAG workflows, the system could achieve more efficient resource utilization across the serverless back-end.

One possible area of improvement could be to apply histogram models to capture the resource demand of DAGs that are not input-size dependent. Such workflows do not determine the resource demand of functions based on the input size. Our current implementation only considers input-size dependent workflows.

Another avenue for improvement is designing an encoding mechanism that captures user input content and code to better recognize harvesting opportunities. This encoding would need to respect security concerns while enabling more granular resource allocation based on the complexity and characteristics of the user's workload.

Future work could also explore applying both intra-DAG and inter-DAG prewarming and keep-alive strategies for hosting functions within a DAG before execution begins. By prewarming functions based on the DAG structure, the system could reduce cold start latencies and improve overall performance.

Chapter 6. Conclusion

In this thesis, we presented an approach to resource harvesting during parallel stages in serverless DAG workflows, addressing key challenges related to execution skew, resource allocation, and efficient scheduling. Our design leverages a predictive inference pipeline that models the relationship between input size and resource demands for functions within parallel stages of a DAG. By integrating these predictions into our scheduling design, we were able to optimize resource utilization, minimize latency, and improve overall system performance.

The proposed system also introduced a composition-aware scheduling algorithm, which places functions from the same composition on the same invoker to reduce communication overhead. Additionally, our resource harvesting mechanism also ensures that resource allocations are dynamically adjusted based on real-time predictions, avoiding over-provisioning and under-provisioning issues while respecting user-defined requirements.

Through our evaluations, we demonstrated that our system can achieve 200% more CPU utilization, 30% more memory utilization, and decrease the request latency by 44% on average. Our scheduling algorithm designed for serving serverless DAGs in the cloud achieves a 10% improvement compared to traditional scheduling algorithms.

Appendix A. Design and Implementation of DAG Workflows

A.1. Video Analytics

The *Video Analytics* workload focuses on image recognition tasks, where a video stream is processed to identify and classify objects. The DAG for this workload consists of four functions arranged in a depth of three and a width of two. The pipeline includes stages for video decoding and object recognition. The video decoder extracts frames from a video file, which are then processed in parallel by multiple object recognition functions using a pre-trained SqueezeNet model. The implementation leverages libraries such as OpenCV (`cv2`) for frame extraction, CouchDB for storing video frames and recognition results, TorchVision for model loading. Figure A.1 illustrates the structure of the DAG for this workload.

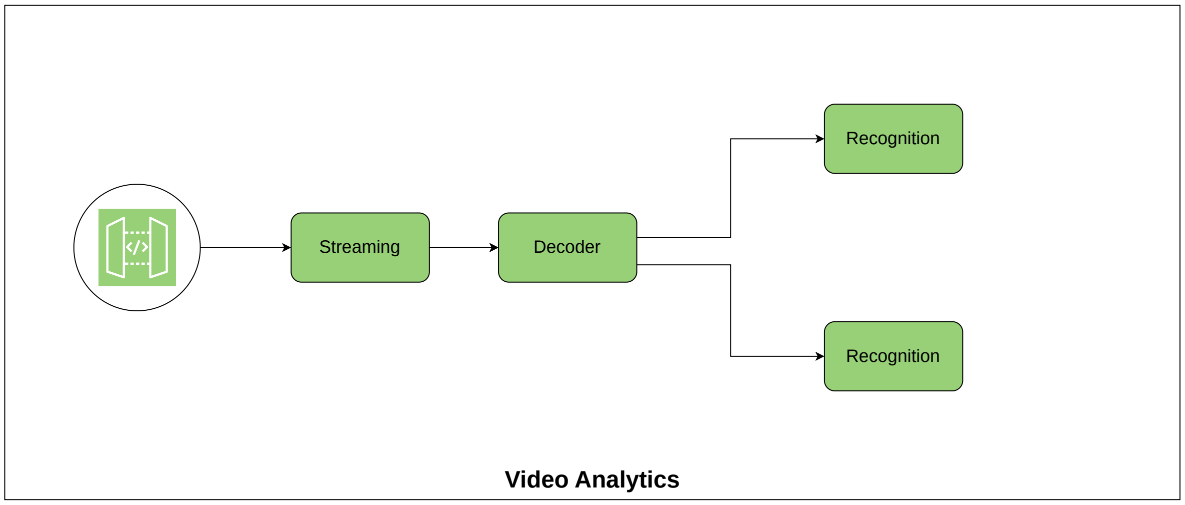


Figure A.1. DAG structure of the Video Analytics workload.

A.2. Parallel AES

The *Parallel AES* workload is a text encryption pipeline using the AES encryption algorithm. The DAG for this workload includes five functions with a depth of three and a width of three. Each AES function operates in parallel to iteratively encrypt and decrypt different segments of the input text, enabling efficient processing of large datasets. The implementation leverages several Python libraries, including `pyaes` for encryption, `multiprocessing` and `threading` to handle parallel processing of text within each function. Figure A.2 shows the structure of this workload's DAG.

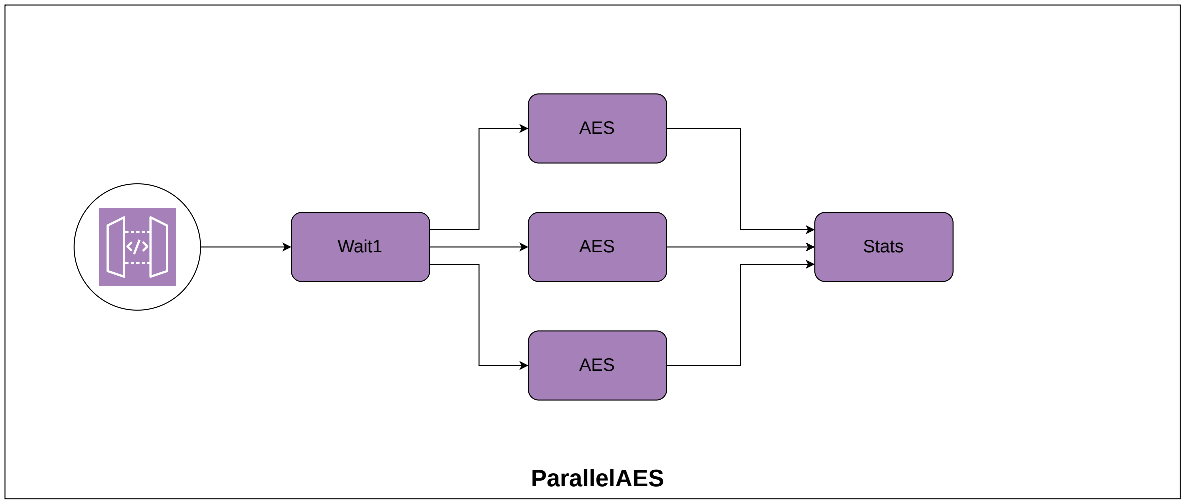


Figure A.2. DAG structure of the Parallel AES workload.

A.3. ML Pipeline

The *ML Pipeline* workload simulates a machine learning model training pipeline, where the dataset is processed through multiple stages, including data preprocessing, model training, and evaluation. During the parallel stage of the ML Pipeline, the workload trains machine learning models using LightGBM in a distributed manner. This stage of the pipeline is designed to run multiple processes in parallel, each process training a tree with different hyper-parameters, such as the number of trees, maximum depth, and feature fraction.

Figure A.3 provides a visual representation of the DAG structure for this workload.

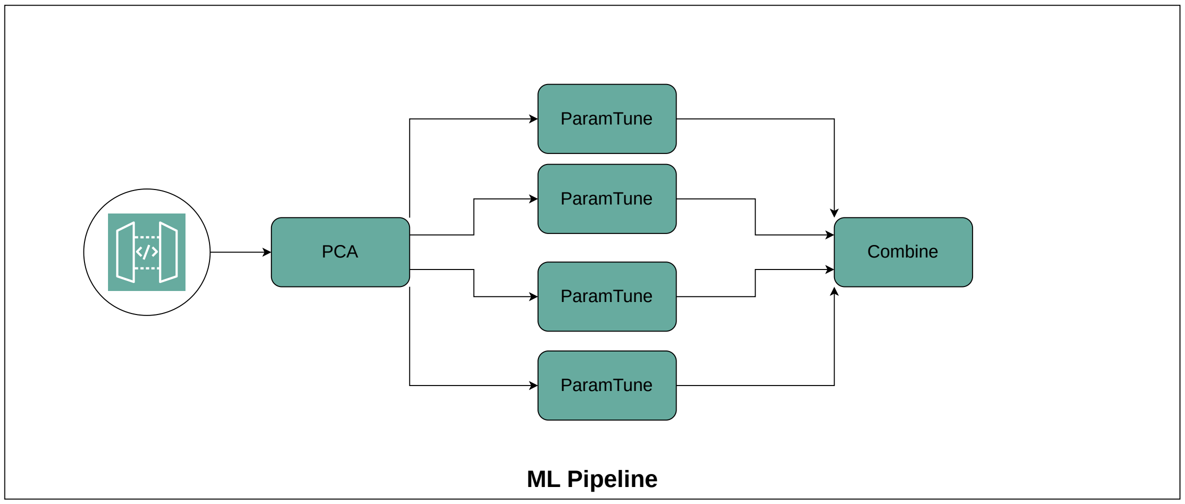


Figure A.3. DAG structure of the ML Pipeline workload.

A.4. DAG Composition

In this section, we describe the process of orchestrating a DAG workflow in our customized version of Apache OpenWhisk. The orchestration involves two main stages: first, each function is compiled as an independent action within OpenWhisk; then, a DAG composition is created using the Composer model to define the overall workflow structure.

A.4.1. Function Compilation

Each function in the workflow is packaged and deployed as an OpenWhisk action. The following example outlines the process for the *Video Analytics* workload, where functions like video streaming, decoding, and object recognition are compiled into separate actions. For each function, the source code is bundled into a zip file and uploaded as an OpenWhisk action. The shell script below demonstrates the steps to compile and upload each function for the *Video Analytics* workload:

Listing A.1. Function Compilation for Video Analytics Workload

```
#!/bin/bash
```

```
cd video-streaming
rm -rf build; mkdir build
cp -R src/* build
cd build; zip -r index.zip *
wsk -i action update streaming --kind python:3.10 --main main --memory \$memo
cd ../../decoder
rm -rf build; mkdir build
cp -R src/* build
cd build; zip -r index.zip *
wsk -i action update decoder --kind python:3.10 --main main --memory \$memory
cd ../../image-recognition
rm -rf build; mkdir build
cp -R src/* build
cd build; zip -r index.zip *
wsk -i action update recognition1 --kind python:3.10 --main main --memory \$m
cd ../../image-recognition
rm -rf build; mkdir build
cp -R src/* build
cd build; zip -r index.zip *
wsk -i action update recognition2 --kind python:3.10 --main main --memory \$m
```

This process compiles the following actions for the *Video Analytics* workload:

- `streaming`: Handles video streaming.

- `decoder`: Extracts frames from video files.
- `recognition1` and `recognition2`: Perform object recognition on different image frames.

A.4.2. Creating the DAG Composition

Once each function is compiled and deployed as an OpenWhisk action, we create the DAG composition using the Java Script implementation of the OpenWhisk Composer model. The composition defines the sequence and parallelism of the function invocations within the DAG. In the case of the *Video Analytics* workload, we invoke the `streaming` function first, followed by the `decoder` function, and then two object recognition functions in parallel.

The following JavaScript file demonstrates how the composition is created using the Composer model:

Listing A.2. DAG Composition for Video Analytics Workload

```
const composer = require("openwhisk-composer");
module.exports = composer.sequence(
  composer.action("streaming"),
  composer.action("decoder"),
  composer.parallel(
    composer.action("recognition1"),
    composer.action("recognition2")
  )
);
```

A.4.3. Deploying and Invoking the Composition

After creating the composition, we deploy the DAG using a shell script. The script constructs the DAG by compiling the composition into JSON format and deploying it as an OpenWhisk sequence. It also handles invoking the DAG workflow by sending an input payload to the OpenWhisk invoker:

Listing A.3. Deploying and Invoking Video Analytics DAG

```
#!/bin/bash
cd video-analytics
compose DAG.js > DAG.json
deploy vid DAG.json -w -i -m \$memory -t \$timeout
wsk -i action invoke vid -P input.json
echo -e "End-of-Invocation"
```

A.4.4. Custom Runtimes for Python Workloads

For workloads that use Python runtimes, we upload each DAG runtime to Docker-Hub and modify OpenWhisk to use these custom runtimes during container initialization. This ensures that each function in the DAG is executed in its appropriate runtime environment, supporting custom requirements and configurations needed by the system.

Appendix B. Mathematical Formulation of the Regression Problem

Our prediction pipeline involves a two-step regression process to estimate the resource demands of each function in a DAG.

Step 1: Predicting Function Input Sizes (ML1) The first set of models, ML1, predicts the input size for each function f_i in the DAG based on the total input size of the DAG I_{DAG}:

$$\hat{I}_{f_i} = \text{ML1}_{f_i}(I_{\text{DAG}}, \text{stage info}, \text{function index}) \quad \forall f_i \in \mathcal{F}_k, \forall D_k \in \mathcal{D}$$

Where $\hat{I}_{f_i}$ is the predicted input size for function f_i, based on the overall DAG input size and function-specific metadata.

Step 2: Predicting CPU and Memory Demands (ML2) The second set of models, ML2, takes the predicted function input size $\hat{I}_{f_i}$ as input and predicts the CPU and memory demand for each function:

$$\hat{C}_{f_i} = \text{ML2}_C(\hat{I}_{f_i})$$

$$\hat{M}_{f_i} = \text{ML2}_M(\hat{I}_{f_i})$$

Where $\hat{C}_{f_i}$ and $\hat{M}_{f_i}$ are the predicted CPU and memory demands for function f_i.

Objective Function The objective for both ML1 and ML2 models is to minimize the mean squared error (MSE) between predicted and actual values. The loss functions for the models are as follows:

For ML1 (input size prediction):

$$\mathcal{L}_{\text{ML1}} = \frac{1}{N}\sum_{i=1}^{N}(I_{f_i} - \hat{I}_{f_i})^2$$

For ML2 (CPU and memory prediction):

$$\mathcal{L}_{\text{ML2},C} = \frac{1}{N}\sum_{i=1}^{N}(C_{f_i} - \hat{C}_{f_i})^2$$

$$\mathcal{L}_{\text{ML2},M} = \frac{1}{N}\sum_{i=1}^{N}(M_{f_i} - \hat{M}_{f_i})^2$$

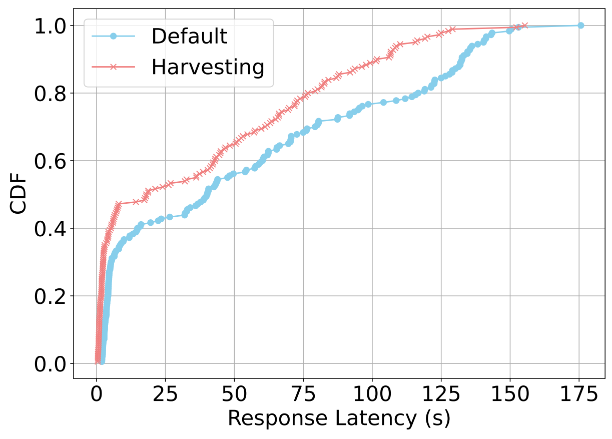
Where N is the number of training samples, and $I_{f_i}, C_{f_i}, M_{f_i}$ represents the actual input size, CPU demand, and memory demand for function f_i, respectively.

Appendix C. Additional Plots

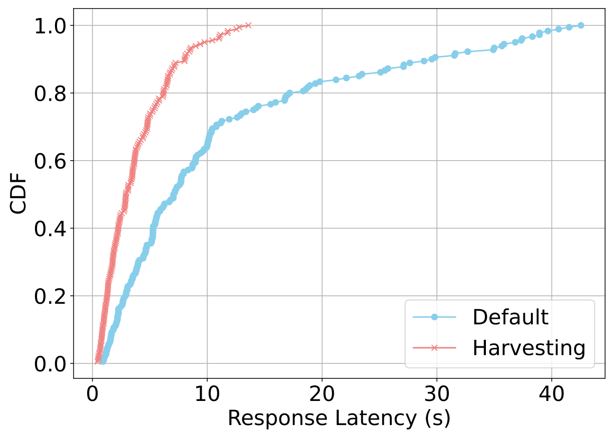
Figure C.1 shows the CDF (Cumulative Distribution Function) of the parallel stage latency during different workload scenarios. Sub-figure (a) represents the latency under a low workload, while sub-figures (b) and (c) correspond to medium and high workloads, respectively. The CDF helps in understanding the distribution of latencies and identifying any performance bottlenecks across different workload intensities. We observe that the improvement in the parallel stage is almost identical to request latency improvement, which means our harvesting strategy is responsible for such improvement.

Figure C.2 shows the performance models for the MLPipeline workload. It presents the error models for input size, CPU utilization, and memory utilization. These models are used to predict resource demands during different stages of the MLPipeline execution and assess the accuracy of these predictions across the specified metrics. Similarly, figure C.3 displays the performance models for the ParallelAES workload, covering input size, CPU utilization, and memory utilization. Models with relatively low performance are omitted from the figures for readability.

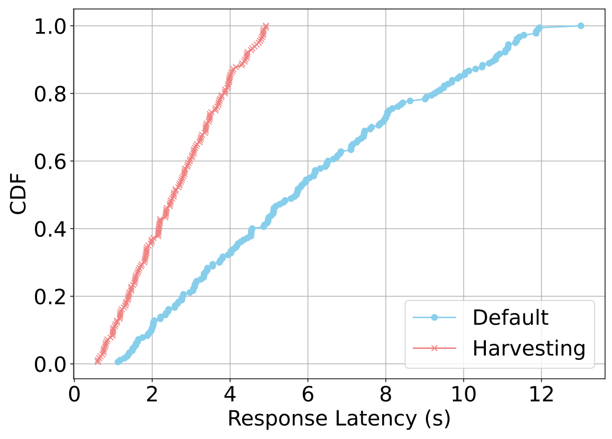
Figure C.4 compares the prediction accuracy for the MLPipeline and ParallelAES workloads. The accuracy of predictions across these workloads is important for assessing the effectiveness of the system's harvesting and its ability to handle different types of workflows efficiently. We use the RandomForest model across all workflows due to its superior performance.



(a) Low Workload

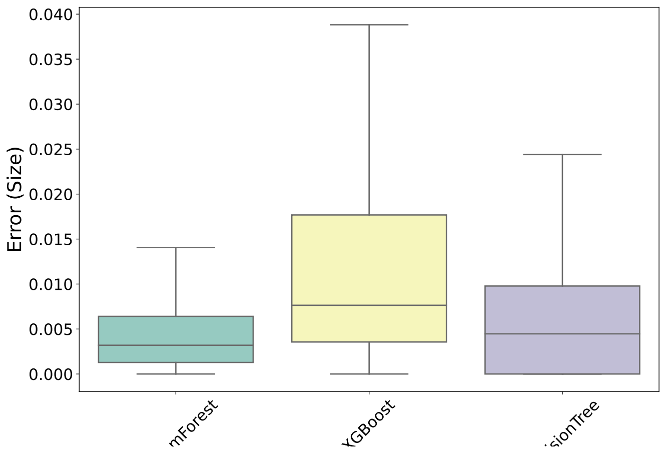


(b) Medium Workload

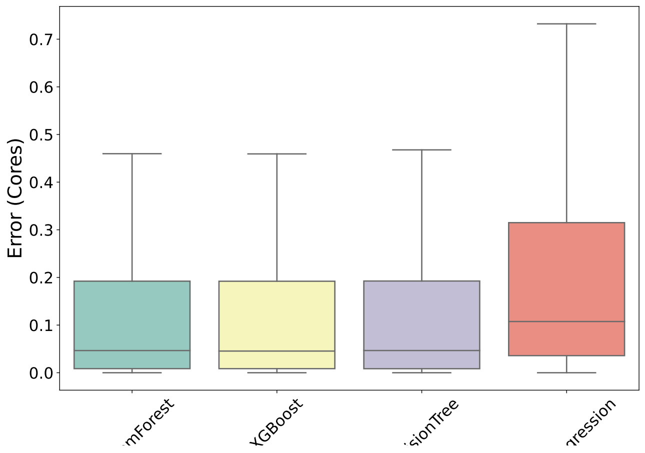


(c) High Workload

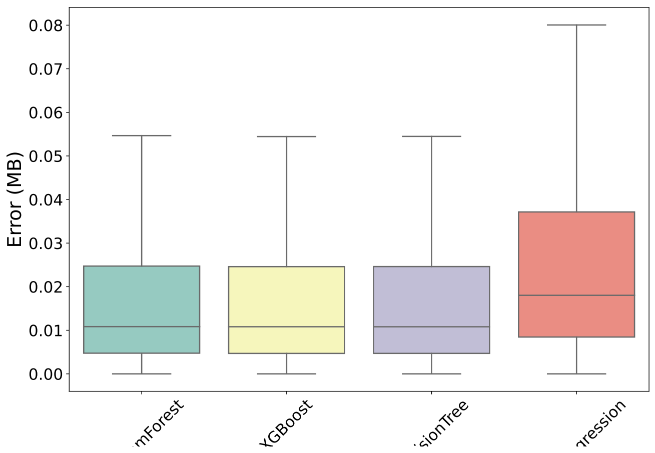
Figure C.1. CDF of parallel stage latency during different workloads



(a) Input Size

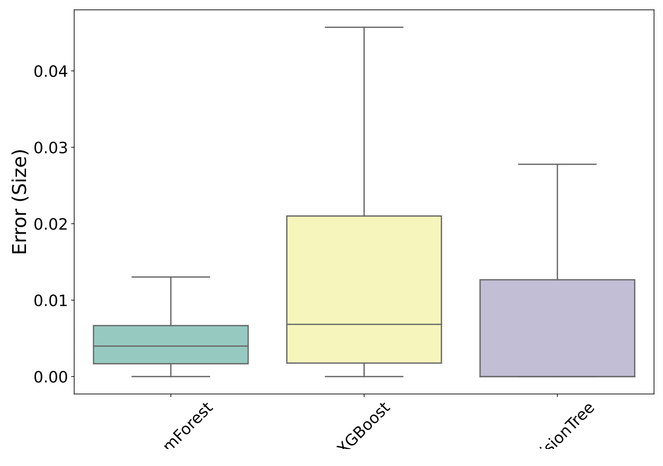


(b) CPU

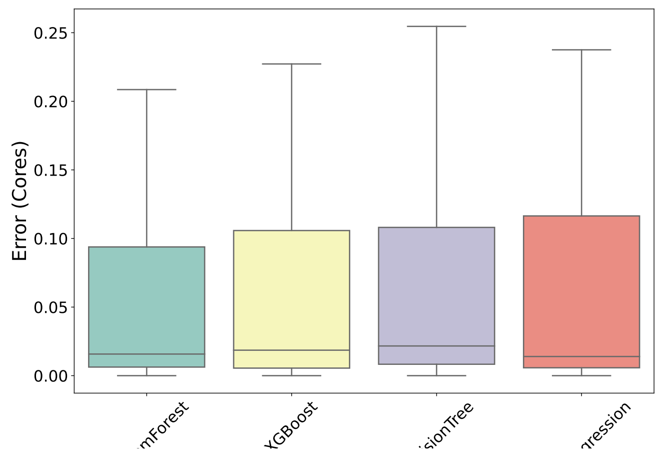


(c) Memory

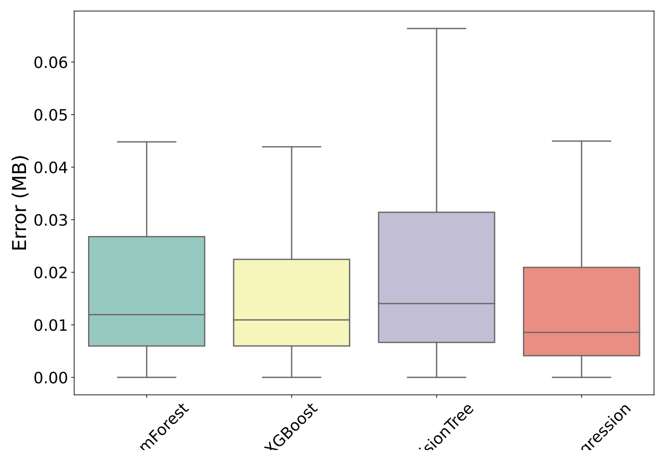
Figure C.2. Performance models across different tasks for MLPipeline: Input Size, CPU, Memory.



(a) Input Size

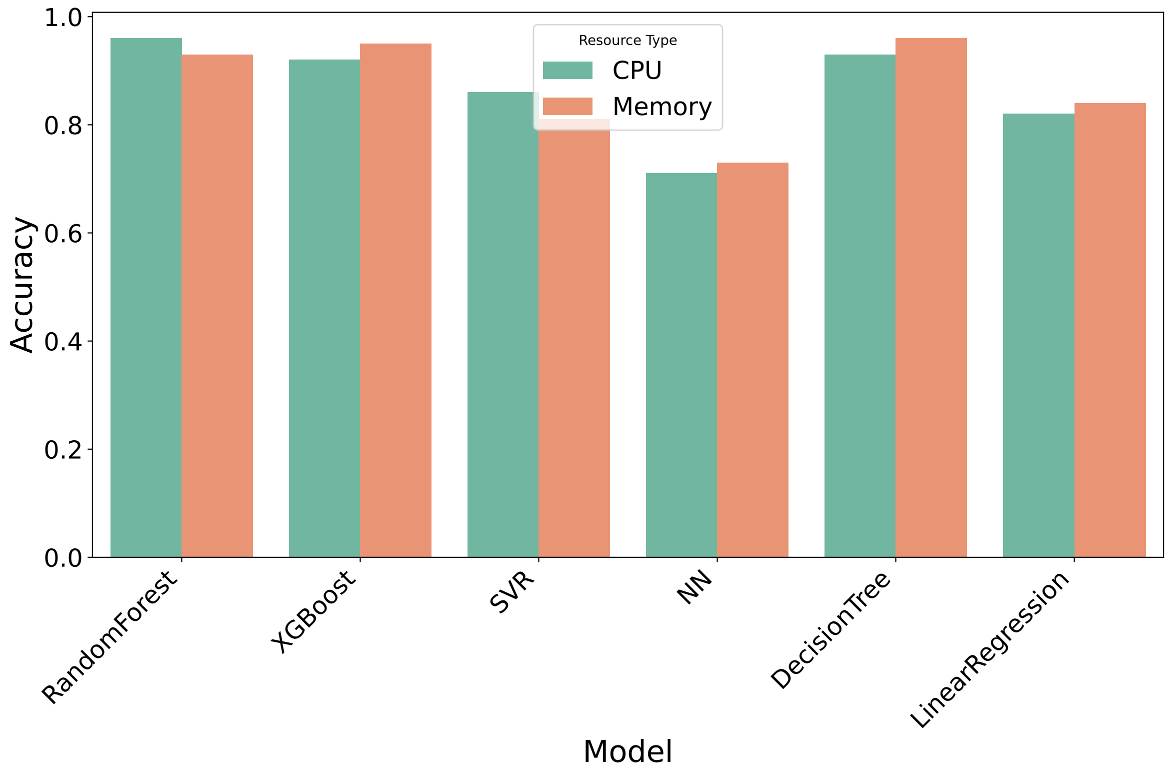


(b) CPU

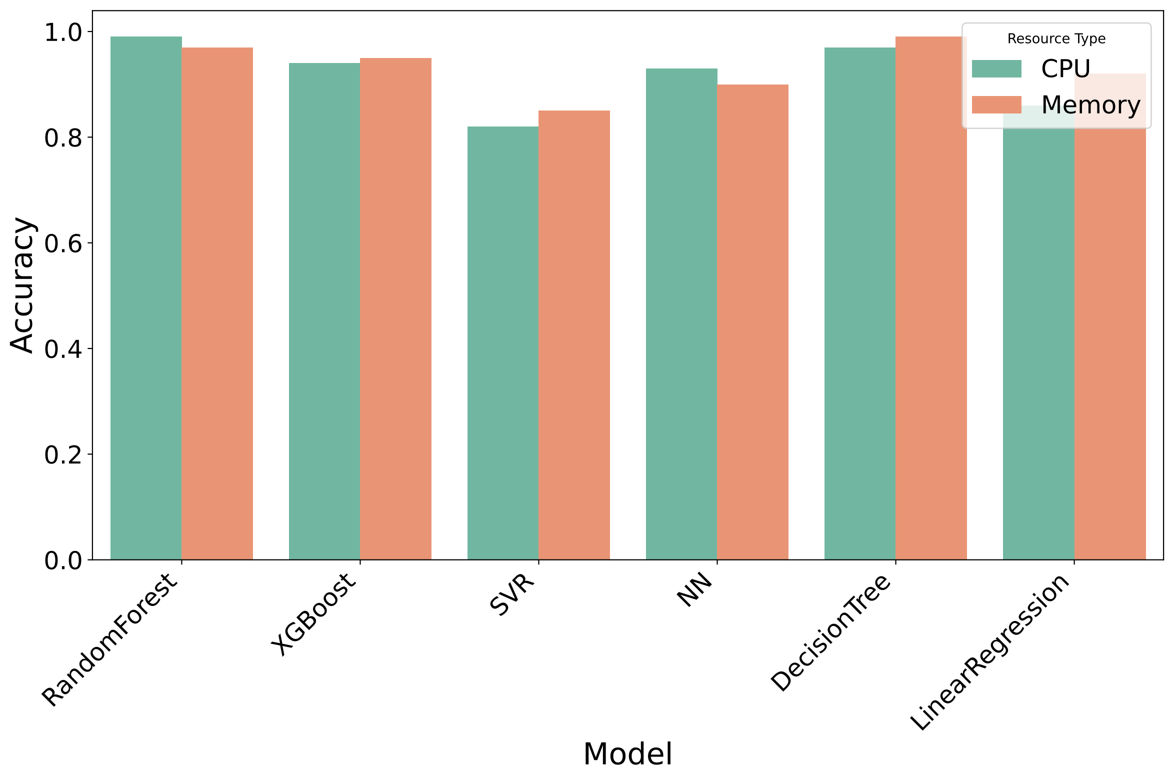


(c) Memory

Figure C.3. Performance models across different tasks: Input Size, CPU, Memory.



(a) MLPipeline



(b) ParallelAES

Figure C.4. Prediction accuracy for MLPipeline and ParallelAES.

Bibliography

[1] Amazon Web Services. *AWS Step Functions*, 2023. Accessed: 2024-10-24.

[2] Apache Software Foundation. Apache openwhisk composer, 2024. Accessed: 2024-10-10.

[3] AWS Documentation. *Lambda function scaling*, 2023. Accessed: 2024-10-24.

[4] Azure Public Dataset. *Azure Functions Blob Dataset 2020*, 2020. Accessed: 2024-10-15.

[5] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 153–167, New York, NY, USA, 2021. Association for Computing Machinery.

[6] Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. With great freedom comes great opportunity: Rethinking resource allocation for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 381–397, New York, NY, USA, 2023. Association for Computing Machinery.

[7] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand container loading in AWS lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 315–328, Boston, MA, July 2023. USENIX Association.

[8] Google Cloud. *Google Cloud Workflows*, 2023. Accessed: 2024-10-24.

[9] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. Elasticflow: An elastic serverless training platform for distributed deep learning. 2023.

[10] Eric Jonas, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. *CoRR*, abs/1702.04024, 2017.

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

[12] Yiming Li, Laiping Zhao, Yanan Yang, and Wenyu Qu. Rethinking deployment for serverless functions: A performance-first perspective. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[13] Qingyuan Liu, Yanning Yang, Dong Du, Yubin Xia, Ping Zhang, Jia Feng, James R. Larus, and Haibo Chen. Harmonizing efficiency and practicability: Optimizing resource utilization in serverless computing with jiagu. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 1–17, Santa Clara, CA, July 2024. USENIX Association.

[14] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, Carlsbad, CA, July 2022. USENIX Association.

[15] Microsoft Documentation. *Durable Functions Overview*, 2023. Accessed: 2024-10-24.

[16] Alfonso Pérez, Sebastián Risco, Diana María Naranjo, Miguel Caballer, and Germán Moltó. On-premises serverless computing for event-driven data processing applications. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 414–421. IEEE.

[17] Ghazal Sadeghian, Mohamed Elsakhawy, Mohanna Shahrad, Joe Hattori, and Mohammad Shahrad. UnFaaSener: Latency and cost aware offloading of functions from serverless platforms. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 879–896, Boston, MA, July 2023. USENIX Association.

[18] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo 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)*, pages 205–218. USENIX Association, July 2020.

[19] The Linux Kernel Organization. *Linux Kernel Documentation: Control Group v2*, 2024. Accessed: 2024-10-19.

[20] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. Smartharvest: harvesting idle cpus safely and efficiently in the cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 1–16, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Hanfei Yu, Christian Fontenot, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. Libra: Harvesting idle resources safely and timely in serverless clusters. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '23, page 181–194, New York, NY, USA, 2023. Association for Computing Machinery.

[22] Hanfei Yu, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. Accelerating serverless computing by harvesting idle resources. In *Proceedings of the ACM Web Conference 2022*, WWW '22, page 1741–1751, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Hanfei Yu, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. Freyr $^{+}+$: Harvesting idle resources in serverless computing via deep reinforcement learning. *IEEE Transactions on Parallel and Distributed Systems*, 35(11):2254–2269, 2024.

[24] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 724–739, New York, NY, USA, 2021. Association for Computing Machinery.

Vita

Peiman Fotouhi was born in December 1997 in the city of Isfahan, Iran, where he spent his childhood. He pursued his undergraduate studies in Computer Engineering at the University of Isfahan, focusing primarily on applications of machine learning. During this time, he developed a strong foundation in the field, which led to an internship with EEPACO, a mojor IT company of that city, allowing him to expand his technical skills.

Peiman later applied for the PhD program in Computer Science at Louisiana State University (LSU), where he served as both a Teaching Assistant (TA) and a Research Assistant (RA). His research during this period concentrated on optimizations in serverless computing. Due to a conflict of interest in his research focus, Peiman transitioned to the Master's program after his first year in the PhD track.

Peiman's current research interests lie at the intersection of deep learning and computing optimizations, and he intends to pursue PhD studies in the field of deep learning in the near future.