Serverless Dataflows: ...

Diogo Alexandre Ferreira de Jesus

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Luis Manuel Antunes Veiga

Examination Committee

Chairperson: Prof. Name of the Chairperson
Supervisor: Luis Manuel Antunes Veiga
Member of the Committee: Prof. Name of First Committee Member

October 2025

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

Abstract

Serverless computing has become a suitable cloud paradigm for many applications, prized for its operational ease, automatic scalability, and fine-grained pay-per-use pricing model. However, executing workflows, which are compositions of multiple tasks, in Function-as-a-Service (FaaS) environments remains inefficient. This inefficiency stems from the stateless nature of functions, and a heavy reliance on external services for intermediate data transfers and inter-function communication.

In this document, we introduce a decentralized DAG engine that leverages historical metadata to plan and influence task scheduling. Our solution encompasses metadata management, static workflow planning, and a worker-level scheduling strategy designed to drive workflow execution with minimal synchronization. We compare our scheduling approach against WUKONG, another decentralized serverless DAG engine. Our evaluation demonstrates that utilizing historical information significantly improves performance and reduces resource utilization for workflows running on serverless platforms.

Keywords

Cloud Computing; Serverless; FaaS; Serverless Workflows; Serverless DAGs; Metadata; Workflow Prediction

Resumo

A computação serverless tornou-se um paradigma de nuvem adequado para muitas aplicações, valorizado pela sua facilidade operacional, escalabilidade automática e modelo de preços granular baseado na utilização. Contudo, a execução de workflows, que são composições de múltiplas tarefas, em ambientes Function-as-a-Service (FaaS) permanece ineficiente. Esta ineficiência resulta da natureza *stateless* (sem estado) destas funções e de uma forte dependência de serviços externos para transferências de dados intermédios e comunicação entre funções.

Neste documento, apresentamos um motor de workflows serverless descentralizado que utiliza métricas recolhidas durante a execução para planear e influenciar o *scheduling* de tarefas. A nossa solução abrange a gestão de metadados, o planeamento estático de workflows e uma estratégia de *scheduling* ao nível dos workers concebida para conduzir a execução de workflows de uma forma descentralizada e com sincronização mínima. Comparamos a nossa abordagem com o WUKONG, outro motor de workflows serverless descentralizado. A nossa avaliação demonstra que a utilização de informação histórica melhora significativamente o desempenho e reduz a utilização de recursos para workflows executados em plataformas serverless.

Palavras Chave

Computação em Nuvem; Serverless; FaaS; Workflows Serverless; Serverless DAGs; Metadados; Previsão de Workflows

Contents

List of Figures

List of Tables

List of Algorithms

Listings

1

Introduction

Contents

Function-as-a-Service (FaaS) represents a serverless cloud computing paradigm that simplifies application deployment by abstracting away infrastructure management. It provides automatic, elastic scalability—potentially without limit—along with a fine-grained, pay-per-use pricing model. This has led to its widespread adoption for event-driven systems, microservices, and web services on platforms like AWS Lambda [1], Azure Functions [2], and Google Cloud Functions [3]. These applications typically benefit the most from FaaS because they are lightweight, stateless, and characterized by highly variable or unpredictable workloads, allowing them to leverage serverless platforms' on-demand scalability and cost-efficiency.

This paradigm is also increasingly used to execute complex scientific and data processing workflows, such as the Cybershake [4] seismic hazard analysis or Montage [5], an astronomy image mosaicking workflow. These applications are structured as workflows—formally represented as Directed Acyclic

Graphs (DAGs) of interdependent tasks. However, efficiently executing these complex workflows on serverless platforms remains a significant challenge.

1.1 Problem/Motivation

Despite their advantages, serverless platforms present several limitations that complicate the execution of complex workflows. Since these platforms allow scaling down to zero resources to save costs, they can also introduce unpredictable latency, known as *cold starts* [6], particularly for short-lived functions, affecting overall workflow performance. The lack of *direct inter-function communication* [7] means that tasks often have to rely on external services, such as message brokers or databases to exchange intermediate data, which can increase overhead and reduce efficiency. Interoperability between platforms is further limited by the use of platform-specific workflow definition languages, which restricts the portability of workflows across different serverless environments. Additionally, while statelessness simplifies scaling and management, it can introduce overhead and complexity for applications that require continuity or coordination across multiple function invocations. Finally, developers have limited control over the underlying infrastructure, restricting the ability to optimize resource usage or tune performance for specific workloads.

Several solutions have emerged to address the limitations of serverless platforms. Stateful functions (e.g., AWS Step Functions [8], Azure Durable Functions [9], and Google Cloud Workflows [10]) expand the range of applications that can run on serverless platforms by maintaining state across multiple function invocations, coordinating complex workflows, and providing built-in fault tolerance. Other approaches tackle limitations at the runtime level, proposing extensions to FaaS platforms (e.g., Faa$T [11], Palette [12], Lambdata [13]) or entirely new serverless architectures (e.g., Apache OpenWhisk [14]).

Other research projects focus on improved orchestration and coordination mechanisms that work on top of FaaS platforms, such as Moyer et al. [15]'s hole punching approach to allow direct inter-function communication, Pheromone [16], Triggerflow [17], FaDO [18], and FMI [19]. These solutions aim to overcome the inherent limitations of stateless functions through intelligent middleware layers that optimize function coordination, data placement, and workflow execution without requiring modifications to the underlying FaaS infrastructure.

Finally, some workflow-focused solutions (e.g., WUKONG [20], Unum [21], DEWEv3 [22]) employ scheduling strategies and workflow-level optimizations to enhance efficiency, primarily by improving data locality to bring computation closer to the data and minimize reliance on external services.

1.2 Gaps in prior work

These workflow-focused approaches, however, often use the *same resources for all tasks* in a workflow and rely on *"one-step scheduling"*, making decisions based solely on the immediate workflow stage without considering the broader context or the downstream effects of their decisions. This combination of homogeneous worker configurations and limited scheduling foresight can lead to inefficient use of resources when tasks have diverse requirements. Furthermore, the heuristic-based approaches used by other solutions can be inefficient in certain scenarios, as they lack mechanisms to adapt worker resource allocations to the specific needs of individual tasks. Moreover, we found no prior work that leverages metadata or historical metrics to inform scheduling decisions across an entire serverless workflow.

1.3 Proposed Solution

These research gaps motivated the central research question of this work: if we have knowledge of all DAG tasks, collect sufficient metrics on their behavior, and understand how they are composed to form the full workflow, can we leverage this information to make smarter scheduling decisions that minimize *makespan* (the total time taken to complete a workflow) and maximize resource efficiency in a FaaS environment?

To answer this research question, we propose a decentralized serverless workflow execution engine that leverages historical metadata from previous workflow runs to generate informed task allocation plans, which are then executed by FaaS workers in a choreographed manner, without needing a central scheduler. By relying on such planning, our approach aims to minimize the usage of external cloud storage services, which are often employed by similar solutions for intermediate data exchange and synchronization, while also avoiding the inefficiencies of homogeneous worker resource allocations.

1.4 Document Organization

The rest of this document is organized as follows: In Chapter 3 we do a background analysis on the serverless landscape, analyzing serverless platforms, offerings, open-source solutions and existing research work. In Chapter 4 we present our proposed solution, detailing its architecture and implementation of the core layers and components. In Chapter 5, we evaluate our proposed solution by comparing it with WUKONG's scheduling algorithm as well as with algorithms we have implemented. Finally, in Chapter 6 we conclude our work and discuss future directions for research.

2

Related Work

Contents

In this section, we explore the serverless computing landscape, starting by exposing the architecture of a typical serverless computing platform, referencing the use cases for this new cloud computing model, and presenting both commercial and open-source offerings. We also delve into workflows, showing how they can be represented, how they are run and managed, and contrasting traditional frameworks for workflow management with more recent solutions that explore cloud technologies, including serverless. Then, we write about three extension proposals to the current serverless platforms design, aiming to improve data locality. We finish this section by presenting relevant workflow orchestrators and schedulers (serverful, serverless, and hybrid) for executing tasks, highlighting their advantages but also some of their limitations and inefficiencies.

2.1 Serverless Computing

Traditionally, cloud applications have been deployed on virtual machines, such as Amazon EC2 [1], which provide full control over the operating system and runtime environment. This model allows predictable performance, flexible resource allocation, direct communication via local network interfaces between VMs, and the ability to run long-lived services, but it comes with significant operational overhead: developers must manage provisioning (which can take several minutes), scaling, patching, and fault tolerance.

Serverless computing addresses these challenges by abstracting away infrastructure management, enabling developers to focus solely on application logic. At the storage and database layer, serverless databases and object stores automatically scale with demand and charge based on actual usage. At the application level, **Backend-as-a-Service** (BaaS) platforms offer ready-to-use components like authentication and messaging. Finally, at the compute layer, **Function-as-a-Service** (FaaS) provides the most flexible and fine-grained model, allowing developers to deploy individual functions that execute on demand in response to events. In this document, we focus specifically on FaaS, as it is the model most relevant to our work.

The Function-as-a-Service (FaaS) model is now offered by major cloud providers, including Amazon (Lambda [1]), Google (Cloud Run Functions [3]), Microsoft (Azure Functions [2]), and Cloudflare (Workers [23]). In addition to these commercial offerings, several open-source runtimes such as OpenWhisk [14], OpenFaaS [24], and Knative [25] provide developers with alternatives for deploying FaaS in self-managed or hybrid environments.

[1] https://aws.amazon.com/pt/ec2/

2.1.1 Advantages

Recent industry reports [2] show that serverless computing has seen rapid adoption over the last few years. For example, in 2024 the global serverless computing market was estimated at USD 24.51 billion, and it is projected to more than double to USD 52.13 billion by 2030, with a compound annual growth rate (CAGR) of about 14.1%. Function-as-a-Service (FaaS) constitutes the majority service model, representing over 60% of serverless market share in 2024. This rapid growth highlights the increasing appeal of serverless architectures, which can be attributed to the following key benefits:

- **Operational Simplicity** means that developers are abstracted away from the underlying infrastructure management, without worrying about server maintenance, scaling, or provisioning. This enables faster development and deployment cycles;

- **Scalability** means the FaaS runtime handles increasing workloads by automatically provisioning additional computational capacity as demand grows, ensuring that applications remain responsive and performant. This makes the FaaS model ideal for applications with *highly variable* or *unpredictable* usage patterns, where we don't know *how many* or *when* requests will arrive;

- **Pay-per-use**: FaaS provides a pricing model where users are only charged for the resources used during the actual execution time over the memory used by their functions, rather than for pre-allocated resources (as in Infrastructure-as-a-Service).

Given these advantages, the serverless model is particularly attractive for applications with *highly variable* or *unpredictable* workloads, such as web services, event-driven pipelines, and real-time data processing. It also suits applications that benefit from rapid iteration and deployment, including microservices, and APIs, where minimizing operational overhead is crucial. Furthermore, serverless can be advantageous in cost-sensitive contexts, where pay-per-use pricing reduces expenses for workloads that do not require continuous execution.

These benefits make serverless computing attractive not only for simple, event-driven applications but also for more complex workflows. Serverless workflows are a composition of multiple computational tasks that are chained together to execute applications by orchestrating individual serverless functions into a coordinated sequence. Some workflows have been successfully experimented with on FaaS. Notable examples include ExCamera [26], a highly parallel video encoding system; Montage [5], an astronomical image mosaic generator; and CyberShake [4], a seismic hazard modeling framework.

[2] https://www.grandviewresearch.com/industry-analysis/serverless-computing-market-report

2.1.2 Limitations

While these advantages make serverless computing highly appealing for a wide range of applications, the model is not without its limitations. As adoption has grown, both practitioners and researchers have identified several technical and architectural challenges that hinder its broader applicability and performance. A number of studies have systematically analyzed these issues, among which Li et al. [27] provides a comprehensive overview of the benefits, challenges, and open research opportunities in the serverless landscape. The challenges mentioned include:

- **Startup Latencies**: It's the time it takes for a function to start executing user code. Cold starts (explained further) can be critical, especially for functions with short execution times;
- **Isolation**: In serverless, multiple users share the same computational resources (often the same Kernel). This makes it crucial to properly isolate execution environments of multiple users;
- **Scheduling Policies**: Traditional cloud computing policies were not designed to operate in dynamic and ephemeral environments, such as FaaS's;
- **Resource Management**: Particularly storage and networking, needs to be optimized (by service providers) to handle the low latency and scalability requirements of serverless computing. The lack of direct inter-function networking is an example of a limitation that narrows down the variety of applications that can currently run on FaaS, as some may not support the overhead of using intermediaries (external storage) for data exchange;
- **Fault-Tolerance**: Cloud platforms impose restrictions on developers by encouraging the development of *idempotent functions*. This makes it easier for providers to implement fault-tolerance by retrying failed function executions.

Hellerstein et al. [7] portrays FaaS as a *"Data-Shipping Architecture"*, where data is intensively moved to code, through external storage services like databases, bucket storage or queues, to circumvent the limitation of inter-function direct communication. This can greatly degrade performance, while also incurring extra costs.

These limitations notably impact workflows—complex applications composed of multiple functions orchestrated into a Directed Acyclic Graph (DAG), where each function's output serves as input for subsequent functions. Such workflows are prevalent in scientific computing, data processing, and machine learning pipelines.

2.1.3 Research Efforts

To overcome some of the inherent limitations of traditional Function-as-a-Service (FaaS) platforms, several research initiatives have proposed architectural innovations aimed at improving performance, scalability, and orchestration. Apache OpenWhisk [14] adopts a fully event-driven, trigger-based architecture, in which functions are invoked automatically in response to events, allowing for more responsive execution and efficient resource utilization. Its design supports complex workflows and fine-grained control over function composition, making it suitable for latency-sensitive and distributed applications.

Building on similar principles, TriggerFlow [17] extends the trigger-based approach by implementing an *event-condition-action* paradigm, enabling efficient orchestration of complex workflows such as state machines and DAGs. This allows high-volume event processing, dynamic scaling, and improved fault tolerance, making it well-suited for long-running scientific and data-intensive workflows.

Another notable platform, OpenFaaS [24], fights *vendor lock-in* by emphasizing simplicity and portability, allowing developers to deploy serverless functions on a wide range of infrastructures while maintaining an event-driven execution model. Collectively, these platforms show how architectural innovations—particularly in event handling and workflow orchestration—can mitigate many of the performance and scalability limitations found in conventional FaaS systems.

While solutions such as OpenWhisk and TriggerFlow propose completely novel serverless architectures, others such as Palette Load Balancing [12], Faa$T [11], Pocket [28], Pheromone [16], and Lambdata [13] propose extending either the FaaS runtime or the workflow definition language to address one of the most pressing limitation of the serverless paradigm: data management inefficiencies.

Palette [12] is a FaaS runtime extension that improves data locality by introducing the concept of **"colors"** as *locality hints*. These colors are parameters attached to function invocations, enabling the invoker to express the desired affinity between invocations without directly managing instances. Palette then uses these hints to route invocations with the same color to the same instance *if possible*, allowing for data produced by one invocation to be readily available to subsequent invocations, reducing the need for expensive data transfers, as it would be required in a typical FaaS runtime. This extra control that Palette provides can be used by workflow schedulers, which have insights on the data dependencies between tasks, to try co-locating tasks which share data dependencies, for example, leading to greater performance while also reducing resource utilization.

Faa$T (Function-as-a-Service Transparent Auto-scaling Cache) [11] tackles the same issue as Palette, locality, but does so on the data level, by adding a **transparent caching layer** into the FaaS runtime. Each application is assigned an in-memory *cachelet* that stores frequently accessed data, enabling subsequent invocations to reuse it without resorting to remote storage. Cachelets cooperate as a distributed cache using *consistent hashing* to share objects across instances, while pre-warming and auto-scaling mechanisms adapt the cache to workload demands. Unlike Palette, which requires user-provided hints, Faa$T operates automatically, preserving the simplicity of the serverless model, hence the "transparent" in its name.

Similarly, Lambdata [13] improves data locality by relying on explicit **data intents** provided by the developer. Functions declare which objects they will read and write, allowing the controller to co-locate invocations that share data dependencies on the same worker and reuse a local cache. This reduces remote storage accesses and data transfer overheads. Compared to Palette's flexible color hints, Lambdata's data intents are more precise but place stricter requirements on developers, while contrasting with Faa$T's fully automated and distributed approach. Contrasting with Palette, Lambdata requires less effort from the developer, but at the cost of reduced flexibility.

2.2 Workflows

As stated before, workflows represent systematic methodologies for organizing and executing computational processes, providing a structured approach to designing, managing, and reproducing generic computations. Workflows have proven to be useful for many different use cases, from application payments and order processing, to data analytics pipelines that move and transform large datasets, to scientific computing and simulations where complex experiments are broken into manageable steps.

2.2.1 Workflow Definition Languages

At their conceptual core, most workflows can be represented as directed acyclic graphs (DAGs), which model computational processes by depicting tasks as nodes and their dependencies as edges connecting them. As an example of a typical web application workflow, consider an online payment process. When a user makes a purchase, the workflow can be represented as a DAG, where each task corresponds to a step in the transaction process. The first task may involve verifying the user's credentials, followed by tasks such as checking product availability, processing payment, and confirming the order. Each of these tasks *depends on* the successful completion of the previous step, with dependencies that ensure the correct order of operations. For instance, payment processing cannot proceed without confirming the product availability, and order confirmation only occurs once payment has been processed. This simple DAG structure ensures that each task is executed in sequence, while also *enabling parallel execution* where possible, such as checking product availability and verifying payment simultaneously.

Despite the existence of other ways to express workflows, due to the simplicity of writing and interpreting DAGs, most systems and libraries use this representation. For instance, Apache Airflow [29] uses DAGs to define and schedule workflows defined in Python. Similarly, Dask [30], a Python parallel computing library, also utilizes DAGs to represent task dependencies, enabling the parallel execution of tasks across clusters. DAGMan (Directed Acyclic Graph Manager) [31] is a way HTCondor [32] (distributed computing job manager) users can organize independent jobs into workflows, also in the form of DAGs.

However, there are more flexible alternatives to define workflows. YAWL (Yet Another Workflow

Language) [33] is a *workflow language* that provides a highly expressive framework for workflow management, capable of supporting a wider range of workflow patterns. YAWL uses Petri networks [34] instead of DAGs to model workflows. This allows YAWL to handle more complex control-flow structures, such as loops, parallelism, and advanced synchronization patterns, offering greater flexibility and power in defining and managing intricate workflows.

While using more capable and flexible workflow languages, such as *YAWL* (Yet Another Workflow Language) allows the representation of more complex workflow patterns, most of the tools used for defining and running scientific workflows, like *Apache Airflow*, *Dask*, and HTCondor's DAGMan use the Directed Acyclic Graph format. This is because DAGs effectively model the majority of scientific workflows, which typically involve non-cyclic dependencies, making them simpler to compose, deploy, understand, debug, and visualize.

2.2.2 Traditional Workflow Scheduling

Going from a workflow definition to actual execution involves several key stages: provisioning resources to match computational demands, uploading code, dependencies, and data to ensure a consistent execution environment, scheduling tasks efficiently to optimize cost and performance, monitoring execution for performance and fault detection, and finally deprovisioning resources once the workflow completes. Traditional scheduling approaches from Grid and Cloud computing assume centralized control, which does not fully align with the ephemeral, stateless nature of serverless computing. Serverless platforms, however, can simplify many of these stages by automating resource scaling, data staging, fault handling, monitoring, and teardown, reducing operational overhead while adapting execution to dynamic workloads.

To alleviate some of the developers and researchers' pain points during these steps while scheduling workflows on more traditional *Infrastructure as a Service* (IaaS) platforms, several data processing and workflow scheduling frameworks have emerged. Among the platforms for **data processing pipelines** are Apache Spark [35], Apache Flink [36], and Apache Hadoop [37], which focus on processing and analyzing large datasets efficiently through parallel and distributed computing. On the **workflow scheduling and orchestration** side, traditional platforms include Apache Oozie [38] and HTCondor [32], which manage the execution and coordination of complex sequences of tasks, ensuring that dependencies are handled and resources are allocated effectively. These frameworks help streamline both the data processing and the management of workflows.

Apache Hadoop provided a foundation for large-scale data processing when it introduced the MapReduce [39] paradigm, supported by HDFS and YARN [40] for reliable storage and resource management. Apache Spark provides a flexible distributed model with rich libraries for analytics and efficient task dependency management, while Apache Flink specializes in real-time stream processing with low latency and robust state handling. In terms of workflow orchestration, Apache Oozie specializes in coordinating Hadoop-based tasks, whereas HTCondor targets high-throughput scientific workflows, efficiently managing complex dependencies.

Similarly, Dask [30] and its distributed scheduler (Dask Distributed [41]) extend the familiar Python ecosystem to large-scale data and compute workloads, enabling parallel execution of array, dataframe, and machine learning operations across clusters, with a lightweight task graph scheduler that supports both batch and interactive computation. Together, these frameworks illustrate the range of solutions available for executing and coordinating workflows on top of the IaaS model, spanning batch and streaming data as well as data-centric and scientific computing environments.

While these frameworks address many critical aspects of resource provisioning, code and dependency management, and workflow monitoring, they rely on the *Infrastructure as a Service* (IaaS) model. While offering significant flexibility and control over the computing environment, *IaaS* comes with notable drawbacks as mentioned previously. A major challenge of IaaS platforms is the complexity of *managing and provisioning* virtual machines, storage, and network resources, which requires expertise and incurs significant overhead. Users must also handle scaling, load balancing, and fault tolerance manually, often leading to inefficiencies. Predicting resource requirements is difficult, often resulting in *over-* or *under-provisioning*, and the typical hourly billing model can further increase costs, particularly for short workflows that run for only a few minutes.

As highlighted previously, the *serverless* paradigm excels in scenarios where automatic scaling and cost-efficiency are essential, while also providing a much easier set-up process for developers by abstracting away the underlying infrastructure and only requiring the user to follow a few coding rules and minor configuration. Despite its current inefficiencies, the serverless model shows great potential for efficiently running the same types of data processing pipelines and workflows as those handled by the frameworks previously mentioned. Next, we will explore some of the most relevant solutions for scheduling serverless workflows.

2.2.3 Modern Workflow Scheduling

The limitations of IaaS-based frameworks, such as those mentioned previously, have led to a new generation of workflow orchestrators designed for flexibility, usability, and ease of deployment. Unlike earlier systems bound to static clusters and rigid DSLs, modern platforms embrace Python APIs, containerization [3], and cloud-native [4] principles. Tools such as Prefect [42], Dagster [43], and Airflow [29] prioritize developer productivity and observability, while Argo Workflows [44] leverages Kubernetes [45] as its native execution environment. Another interesting project is Apache Beam, a cloud-native *unified*

[3] https://aws.amazon.com/what-is/containerization/
[4] https://aws.amazon.com/what-is/cloud-native/

programming model for batch and streaming data pipelines designed to be executed across multiple backends, with some of the most popular implementations being Google Cloud Dataflow [46]. These solutions mark a shift from infrastructure management toward higher-level abstractions that integrate seamlessly with cloud platforms and services.

2.2.3.A Stateful Serverless Functions

While modern orchestrators such as Argo, Prefect, Dagster, and Airflow provide powerful abstractions for coordinating workflows across diverse environments, they can be unnecessarily complex for developers who already rely on stateless serverless functions within a single cloud provider. In such cases, what is often required is not a general-purpose orchestration framework but a lightweight mechanism to compose stateless functions into more complex workflows, supporting coordination patterns such as *fan-out*, *fan-in*, and conditional branching. To address this gap, cloud providers have introduced *stateful serverless functions*, which augment the stateless Function-as-a-Service (FaaS) model with durable state management and execution control.

A **stateful serverless function** represents a workflow orchestration paradigm in which an external coordination layer manages the execution and state of multiple stateless function invocations. These orchestrators track workflow progress, preserve context across invocations, and handle retries, error propagation, and branching logic. A common mechanism across these platforms is the use of *snapshotting* or durable state persistence: the engine periodically records workflow state so that execution can be paused and later resumed without requiring all individual functions to remain active. By combining snapshotting with techniques such as event sourcing, persistent queues, and transactional state management, stateful orchestrators enable long-running workflows that can scale across distributed environments while remaining fault tolerant and cost efficient.

Prominent examples include AWS Step Functions [8], Google Cloud Workflows [10], Azure Durable Functions [9], and Cloudflare Workflows [47]. Despite differences in design and integration, they share the goal of simplifying complex coordination atop serverless platforms. AWS Step Functions offers JSON- or YAML-based state machines using the Amazon States Language (ASL)[5], enabling workflows that may last up to one year. Google Cloud Workflows provides YAML- or JSON-based definitions with a maximum duration of 60 minutes per execution, tightly integrated with Google Cloud services such as BigQuery and Cloud Run. Azure Durable Functions follows a code-centric approach in which developers write an *"orchestrator function"* in languages such as C#, JavaScript, or Python, with workflow durations of up to 30 days. Finally, Cloudflare Workflows emphasizes lightweight, edge-native [6] orchestration, optimized for event-driven scenarios at the edge.

[5] https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html
[6] https://www.cloudflare.com/learning/serverless/glossary/what-is-edge-computing/

Together, these services extend the applicability of serverless beyond short-lived, stateless tasks, enabling complex approval processes, data pipelines, machine learning training, and financial transaction workflows that require state persistence and long execution durations. At the same time, they shift the responsibility of orchestration away from developers, who would otherwise need to implement custom, serverful coordination layers.

The tradeoff, however, is strong vendor lock-in, since each provider enforces its own workflow representation and tight integration with its ecosystem, making portability and hybrid-cloud adoption more challenging. Furthermore, the user is billed for the orchestration service on top of the individual function executions, which can increase costs for long-running or highly parallel workflows compared to managing orchestration independently. In addition, the orchestrator can make suboptimal scheduling decisions, such as inefficient task placement or resource allocation, over which the user has little or no control, potentially impacting performance and cost efficiency.

2.2.4 Serverless Workflow Scheduling

Having discussed commercial stateful serverless functions and their advantages and limitations, we now turn to research-oriented approaches for orchestrating workflows in stateless serverless environments. Unlike managed offerings, some of these systems explore innovative scheduling strategies and trade-offs that are particularly relevant for our work.

2.2.4.A Workflow Scheduling Approaches

Workflow scheduling/orchestration in serverless environments can generally be categorized into three approaches:

- **Centralized scheduling:** In this approach, a single scheduler maintains a global view of the entire workflow, including task dependencies, resource availability, and execution progress. This allows the scheduler to make fine-grained decisions about task placement, load balancing, and prioritization, often optimizing for latency or resource utilization. However, the centralized model can become a bottleneck as workflow size and concurrency increase, introducing single points of failure. It also requires the scheduler to handle high-throughput metadata and state management, which can be challenging in highly dynamic serverless environments.

- **Queue-based or message-driven scheduling:** Here, tasks are decoupled from execution using queues or message-passing systems. Producers submit tasks to a queue, and workers pull tasks asynchronously when they become idle. This design improves elasticity, as workers can scale independently of the workflow controller, and naturally provides fault tolerance—failed tasks can be retried by re-queuing. While it removes the single bottleneck of a centralized scheduler, queue-based systems may have less optimal global scheduling decisions, and additional logic may be needed to enforce task ordering or dependency constraints.

- **Decentralized and Choreographed scheduling:** In this model, the responsibility for orchestration is distributed across the worker nodes rather than concentrated in a central entity. Each node independently manages task execution, coordinates with peers, and propagates state updates as necessary. This approach eliminates the need for dedicated scheduler infrastructure, mitigates bottlenecks, and enables faster scaling to thousands of concurrent functions. However, this model introduces greater complexity in ensuring fault tolerance and consistent state propagation across distributed, ephemeral environments. Ensuring that tasks execute *exactly once* becomes particularly challenging, requiring stronger coordination and consensus mechanisms. To address these issues, algorithms such as Paxos [48], Raft [49], or coordination services like ZooKeeper [50] can be employed, along with localized snapshots or lightweight distributed state stores, to maintain workflow coherence and reliability without relying on centralized scheduling. Most existing solutions, however, assume that tasks are *idempotent*, so that repeated execution does not produce unintended side effects, simplifying failure recovery and avoiding the need for strict exactly-once guarantees.

2.2.4.B Relevant Workflow Schedulers

To illustrate the spectrum of serverless workflow scheduling strategies, we now discuss representative systems that embody each approach and inspired our work. DEWE v3 [22] demonstrates a hybrid (IaaS and FaaS) queue-based model, where tasks are distributed through dedicated queues depending on their expected execution time. PyWren [51] exemplifies a more centralized scheduling paradigm, in which the user or submission machine maintains control over task execution while workers simply carry out assigned functions. Finally, Unum [21] and WUKONG [20] showcase decentralized choreographed orchestration, distributing workflow logic across workers to achieve high scalability and reduced reliance on central schedulers. Examining these systems provides concrete examples of the trade-offs and optimizations inherent to each scheduling strategy.

A – DEWE v3 DEWE v3 [22] introduces an innovative hybrid approach to serverless workflow orchestration that combines the best aspects of both serverless and serverful computing models. This hybrid workflow execution engine intelligently distributes tasks based on their characteristics: *short tasks* are run on FaaS workers while *longer tasks* run on virtual machines. The system employs a queue-based job distribution mechanism where jobs expected to complete within FaaS limits are published to a common job queue for serverless execution, while long-running jobs are directed to a separate queue for local, serverful execution on dedicated servers.

Jobs that fail to execute on FaaS workers, for being longer than expected and exceeding execution limits imposed by the platform, are redirected to the serverful queue. This dual-execution model enables DEWE to accommodate workflows with diverse resource consumption patterns. This system proves particularly effective for scientific workflows, such as Montage [5], where task durations and resource requirements vary significantly. However, this hybrid approach introduces specific trade-offs. Latency-sensitive workflows may be slowed by job queuing overhead. In addition, hybrid deployments often lead to resource underutilization, as serverful workers may sit idle when most tasks are executed on FaaS. Finally, the centralized workflow manager can become a scalability bottleneck when handling many short tasks.

B – PyWren PyWren [51], representing one of the pioneering pure serverless approaches, demonstrates the potential and limitations of leveraging unmodified serverless infrastructure for distributed computation. Built atop AWS Lambda, PyWren focuses on executing arbitrary Python functions as stateless serverless functions with minimal user management overhead, automatically handling function execution, dependencies, S3 bucket storage for serialized code and intermediate data. The system is ideal for *embarrassingly parallel* workloads, also known as *"bag-of-tasks"* scenarios, with many independent, parallel tasks such as simple data transformations, scientific simulations, parallel model training, and large-scale media processing. While PyWren's serverless orchestration model provides excellent scalability and removes the burden of infrastructure management, its simplicity limits its applicability. It is not well-suited for workflows with complex dependency structures or those that require sharing large intermediate results through object storage. Moreover, latency-sensitive applications are disadvantaged by function cold starts, since PyWren does not include mechanisms to mitigate their impact.

C – Unum Unum [21] takes a radically different approach from the two previous solutions by decentralizing orchestration logic entirely, eliminating the need for a standalone orchestrator service. This application-level serverless workflow orchestration system embeds orchestration logic directly into a library that wraps user-defined FaaS functions, leveraging an external scalable consistent data store for coordination during fan-ins and execution correctness. Unum introduces an Intermediate Representation (IR) to capture information about workflow progression (nearby tasks) and relies only on minimal, common serverless APIs (function invocation and basic data store operations) available across cloud platforms. This design choice provides exceptional portability and cost-effectiveness, as it can run on unmodified serverless infrastructure.

Unum also can and compile workflows defined in languages from providers like AWS Step Functions and Google Cloud Workflows into its IR format. However, Unum's generic approach comes with trade-offs: it currently supports only statically defined control structures and cannot express workflows where the next step is determined dynamically at runtime, and it lacks data locality optimizations since it cannot force related tasks to execute on the same worker, with each function instance executing only its specific task before triggering the next function.

2.2.4.C WUKONG

WUKONG [20] represents the most sophisticated approach among these solutions, designed as a *decentralized choreographed locality-enhanced* serverless workflow engine. Built on top of Dask's programming model and DAG generation capabilities [20], WUKONG reimagines the execution layer to address the limitations of traditional serverful models like Dask Distributed while maximizing the advantages of serverless computing. The system focuses on improving scale-out speed and enhancing data locality to minimize large object movement. WUKONG's architecture is divided into static components (operating before workflow execution) and dynamic components (operating during execution). The static scheduler includes a **DAG Generator** that leverages Dask's library to convert Python code into DAGs, a **Schedule Generator** that creates `n` static schedules for `n` root/leaf nodes (each containing every reachable task in a depth-first search starting at that node), **Initial Task Executor Invokers** that launches the first Lambda instances for each root task, and a **Subscriber Process** on the client that waits for and downloads final results.

After receiving the initial schedules, FaaS workers (referenced to as *AWS Lambda Executors*) drive workflow execution. Workers execute tasks until they encounter a *fan-out*, at which point they transfer data to intermediate storage, execute 1 of the `N` fan-out tasks and invoke *N - 1 new* executors for the other tasks. Then, when they find a *fan-in*, the group of executors that reach the common fan-in node cooperate using a *dynamic scheduling* model to select only one executor to proceed. This coordination is managed through a shared **dependency counter** in a Key-Value Store (KVS). Each involved executor *atomically* updates this counter; the one whose update satisfies the final input dependency for the fan-in task will execute that task and continue along its static schedule. The other executors transfer their intermediate data to storage, and then stop their execution, decreasing the workflow parallelism.

Besides dynamic scheduling, WUKONG employs data locality optimization techniques designed to avoid moving large data objects; **Task Clustering for Fan-Out Operations** allows executors to continue executing downstream tasks when a fan-out task produces large outputs, becoming the executor of multiple fan-out targets rather than just executing 1 and invoking `N - 1` new executors; **Task Clustering for Fan-In Operations** enables executors to recheck dependencies after uploading large objects to storage, potentially executing fan-in tasks themselves if dependencies are satisfied during the upload process, potentially avoiding large downloads; **Delayed I/O** allows executors to hold off on writing large intermediate results to external storage until it is absolutely necessary. Instead of immediately storing data when some downstream tasks are not yet ready, the executor first runs any tasks that can proceed and then checks again if the remaining ones have become ready. If they have, the executor can execute them directly using the data already in memory, avoiding both the write and a later read from storage. Only when no further progress is possible are the results finally written out. This can reduce unnecessary data transfers.

These optimizations, combined with WUKONG's decentralized scheduling approach, significantly enhance performance compared to both **Dask Distributed** and **PyWren** by minimizing data transfer overhead and eliminating central scheduler bottlenecks. However, WUKONG shares the limitation of supporting only statically defined control structures, requiring workflow DAGs to be known ahead-of-time, similarly to our proposed solution. Additionally, its optimization heuristics can lead to inefficiencies in certain scenarios: *Delayed I/O* may increase makespan and storage usage if dependencies aren't met after retries; fan-in conflicts where multiple tasks produce large objects can result in resource waste depending on upload timing; and fan-out scenarios with small inputs may not justify the overhead of invoking multiple executors as it can make subsequent fan-in's more expensive. Furthermore, WUKONG assumes a homogeneous execution environment, where all workers provide identical resources (e.g., each task is allocated 2 CPU cores and 512 MB of memory), which prevents tailoring resources to tasks with different computational or memory demands.

While WUKONG represents a significant advance in serverless workflow orchestration through its decentralization, its scheduling and optimizations remain limited. The system bases decisions only on the next stage of the workflow (i.e., one-to-one, fan-in, or fan-out transitions). We refer to this as *one-step scheduling*, since it relies solely on information about the next step. Crucially, WUKONG does not exploit the global knowledge of the workflow structure, even though the entire workflow structure is known before execution begins. Also, its optimizations rely on heuristic-based strategies that can lead to suboptimal performance when workflow behavior deviates from expected patterns.

2.3 Discussion/Analysis

The existing body of related work highlights a clear progression in workflow orchestration strategies, moving from traditional cluster-based frameworks to cloud-native platforms and finally to serverless execution engines. Cluster-based systems such as Hadoop, Spark, Flink, and Dask emphasize fine-grained control, strong data locality, and predictable performance, but they incur significant operational overhead, limited elasticity, and often require technical expertise to deploy and manage. Commercial cloud-native platforms like AWS Step Functions and Azure Durable Functions simplify deployment and management by abstracting state handling and orchestration, but they typically incur additional costs and are bound to vendor-specific ecosystems. Runtime extensions such as Palette, Faa$T, and Lambdata tackle the core inefficiencies of serverless platforms by enhancing data locality and reducing communication overhead.

Serverless workflow engines like PyWren, Unum, and WUKONG represent the most direct attempts to build high-performance workflow execution on unmodified FaaS infrastructure. While PyWren demonstrates the feasibility of large-scale embarrassingly parallel workloads, it struggles with complex workflows. Unum advances decentralization by embedding orchestration logic directly into functions, achieving portability and cost efficiency, but it remains limited in its support for dynamic control structures and lacks optimizations for data locality. WUKONG achieves major improvements through decentralization and data-aware heuristics such as *Task Clustering* and *Delayed I/O*, delivering great scalability and cost efficiency. Taken together, these systems provided the most direct inspiration for our work, while also highlighting key open challenges that our approach seeks to address.

3

Architecture

Contents

3.1 Workflow Definition Language

We will now present our DAG-based workflow engine that transforms ordinary Python functions into parallelizable tasks, automatically managing dependencies and execution through an intuitive decorator-based API. It is inspired by WUKONG, Dask and Airflow's way of expressing workflows: the user can create workflows by composing individual Python functions, as shown in Listing 3.1. In this example, we define two tasks, `task_a` and `task_b`, and then compose them into a DAG by passing their results as arguments to the next task. The resulting workflow structure is illustrated in Figure 3.2.

```python
# 1) Task definition
@DAGTask
def task_a(a: int) -> int:
    # ... user code logic ...
    return a + 1

@DAGTask(forced_optimizations=[PreLoadOptimization()])
def task_b(*args: int) -> int:
    # ... user code logic ...
    return sum(args)

# 2) Task composition (DAG/Workflow)
a1 = task_a(10)
a2 = task_a(a1)
a3 = task_a(a1)
b1 = task_b(a2, a3)
a4 = task_a(b1)
```

Figure 3.1: DAG definition example

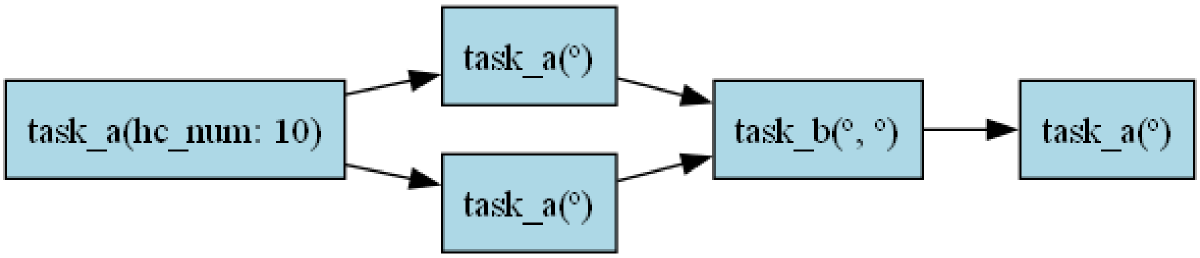

Figure 3.2: Simple DAG example

When `task_a(10)` is invoked, it doesn't actually run the user code. It instead creates a representation of the task, which can be passed as argument to other tasks. The workflow planning and execution only happens once `.compute()` is called on the last/sink task (`a4`), as shown in Listing 3.3. When `compute()` is called, we can create a representation of the entire workflow structure by backtracking the task dependencies.

One limitation of this DAG definition language is that it doesn't support "dynamic fan-outs" (e.g., creating a variable number of tasks depending on the result of another task) on a single workflow. This is a powerful and expressive feature, but that is seldom supported in other DAG definition languages (e.g., Dask, WUKONG, Unum, Oozie [38] do not support it). These languages require the user to split the workflow into multiple workflows, one for each *dynamic fan-out*: one workflow runs up to the task that generates a list of results, while a second workflow starts with a number of tasks that depends on the size or contents of that list.

```python
result = a4.compute(
    dag_name="simpledag",
    config=Worker.Config(
        faas_gateway_address=...,
        intermediate_storage_config=(ip, port, password),
        metrics_storage_config=(ip, port, password),
        planner_config=UniformPlanner.Config(
            sla=sla,
            worker_resource_configuration=TaskWorkerResourceConfiguration(cpus=3, memory_mb=512),
            optimizations=[PreLoadOptimization, TaskDupOptimization]
        )
    )
)
```

Figure 3.3: Setting up and launching workflow execution

Apache AirFlow [1] supports this feature through an extension to their DAG language, allowing a variable number of tasks to be created at run-time depending on the number of results produced by a previous task. Implementing similar functionality is possible, but it would reduce the accuracy of predictions. This is because we would also need to predict the expected fan-out size, and any errors in that prediction could amplify inaccuracies in the predictions for the rest of the workflow.

We will now present the architecture, highlighting the core layers of our solution.

3.2 Overview

The overall architecture and logical flow of our decentralized serverless workflow execution engine is organized into 3 high-level layers. Figure 3.4 provides an overview of this architecture. The upper part represents the components that run on the user's machine, while the lower part represents the components that run outside the user's machine.

The user writes its workflows in Python (demonstrated in Section 3.1). First, a planning algorithm, chosen by the user, will run locally to generate a static workflow plan. This plan defines a task-to-worker mapping and other task-level optimization hints for FaaS workers. Once the plan is done, the client launches the initial workers for the root tasks, kicking off workflow execution. The user program then waits for a storage notification indicating workflow completion and then retrieves the final result from storage.

The following sections should provide a deeper understanding of each layer as well as how the user interacts with the system.

1. **Metadata Management**: Responsible for collecting and storing task metadata from previous executions. It also uses this metadata to provide predictions regarding task execution times, data transfer times, task output sizes, and worker startup times;

[1] https://airflow.apache.org/docs/apache-airflow/stable/authoring-and-scheduling/dynamic-task-mapping.html

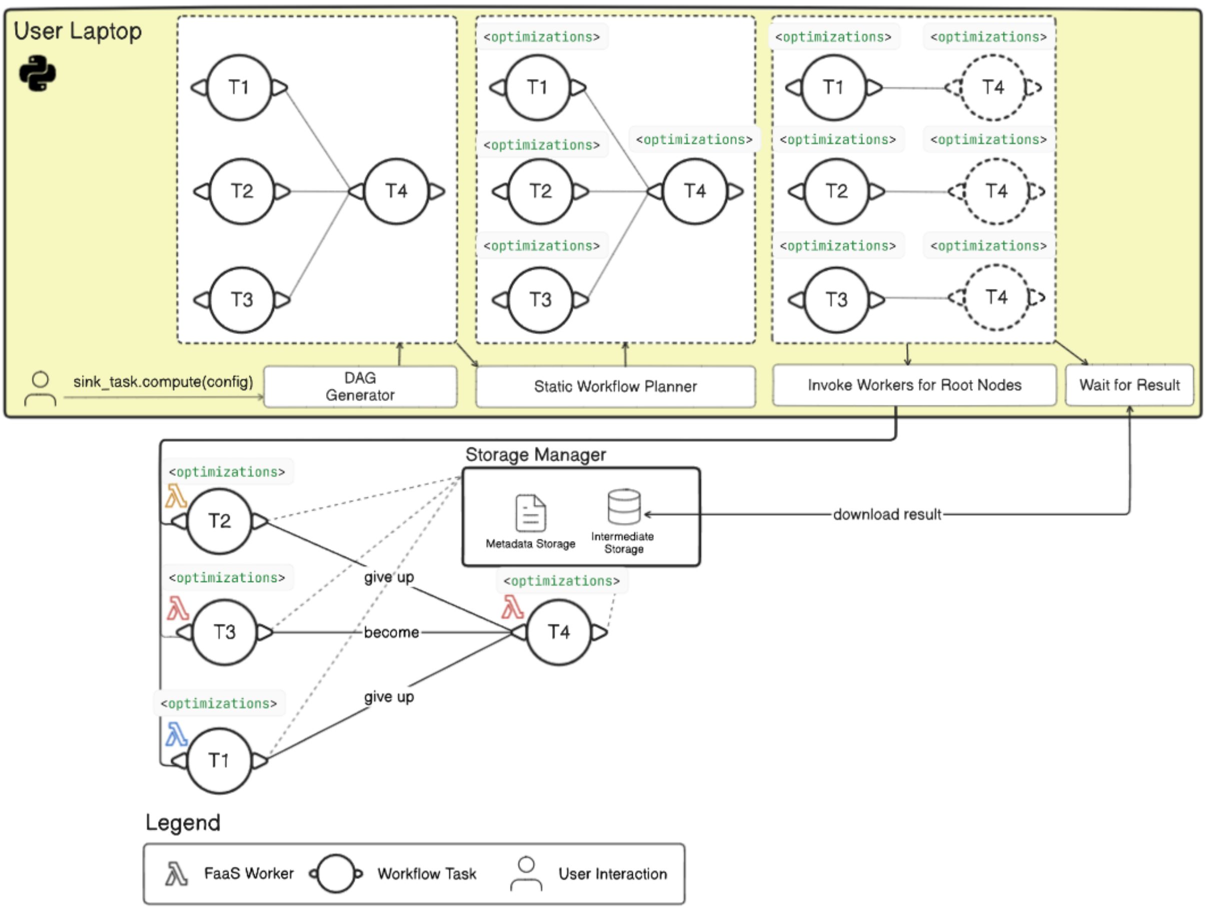

Figure 3.4: Solution Architecture

2. **Static Workflow Planning**: Receives the entire workflow, represented as a Directed Acyclic Graph (DAG), and a "planner" (an algorithm chosen by the user). This planner will use the predictions provided by Metadata Management to create a static plan/schedule to be followed by the workers;

3. **Scheduling**: This component is integrated into the workers, and it is responsible for executing the plan generated by the Static Workflow Planning layer, applying optimizations and delegating tasks as needed.

There are 3 distinct computational entities involved in this system:

- **User Computer**: Responsible for creating workflow plans, submitting them (triggering workflow execution), and receiving its results. The planning phase also happens on this computer, right before a workflow is submitted for execution;

- **Workers**: These are the FaaS workers (often running in containerized environments), that execute one or more tasks. The decentralization of our solution is due to the fact that these workers are responsible for scheduling of subsequent tasks, delegating tasks and launching new workers when needed without requiring a central scheduler. Lastly, they are also responsible for collecting and uploading metadata;

- **Storage**: Consists of an *Intermediate Storage* for intermediate outputs which may be needed for subsequent tasks and a *Metadata Storage* for information crucial to workflow execution (e.g., notifications about task readiness and completion).

Next, we will explain how the user defines and submits workflows for execution.

3.3 Metadata Management

The goal of the **Metadata Management** layer is to provide the most accurate task-wise predictions to help the planner algorithm chosen by the user to make better decisions. To achieve this, while the workflow is running we collect metrics about each task's execution. These metrics are stored in *Metadata Storage*: task execution time, data transfer size and time, task input and output sizes, and worker startup time.

Storing these metrics enables us to provide a prediction API, shown in Listing 3.5. To improve accuracy, metrics are kept separate for each workflow. As a result, even if two workflows use the same function or task code, their metrics are stored independently. This design choice reflects our assumption that different workflows may follow different execution patterns. To avoid introducing runtime overhead, metrics are batched and uploaded when the worker shuts down.

The prediction methods take an additional parameter, `SLA` (Service-level Agreement), which is specified by the user and influences the selection of prediction samples. For example, `SLA="median"` will use the median of the historical samples, whereas `SLA=Percentile(80)` will return a more conservative estimate. By allowing the user to control this parameter, the API can provide predictions that are tailored to different performance requirements.

In addition, metrics such as worker startup time, data transfer time, and task execution time are tied to the specific worker resource configuration. To account for this, our prediction method follows two paths. If we have enough historical samples for the same resource configuration, we use only those. Otherwise, when there are not enough samples with the same resource configuration, we fall back to a normalization strategy: we adjust samples from other memory configurations to a baseline, use those to estimate execution time, and then rescale the result back to the target configuration.

After filtering samples we use an algorithm that selects a limited number of the most relevant samples for each prediction. This algorithm works by gradually widening a tolerance window around the reference value until it finds enough nearby samples. Within each window, it balances samples that are smaller, larger, or exactly equal to the reference, giving preference to the closest ones. If there still aren't enough

```
class PredictionsProvider:
  def predict_output_size(function_name, input_size, sla) -> int

  def predict_worker_startup_time(resource_config, state: 'cold' | 'warm', sla) -> float

  def predict_data_transfer_time(
        type: 'upload' | 'download',
        data_size_bytes,
        resource_config,
        sla
    ) -> float

  def predict_execution_time(
        task_name,
        input_size,
        resource_config,
        sla: SLA
    ) -> float
```

Figure 3.5: Task Predictions API

candidates, it falls back to simply picking the nearest available samples overall. This way, the algorithm adapts to the data while keeping the selection both relevant and limited in size.

3.4 Static Workflow Planning

This layer executes on the user side, and it receives the workflow representation and a workflow planning algorithm chosen by the user (as shown in Listing 3.3). Its job is to execute the planning algorithm, providing it access to the predictions exposed by the Metadata Management layer (Section 3.3).

3.4.1 Workflow Simulation

The availability of predicting task execution time, data transfer latency, function I/O, and worker startup cost, exposed through the Metadata Management layer (Section 3.3), makes it possible to simulate the execution of an entire workflow instance.

To this end, we implemented a dedicated simulation module. The module accepts a workflow representation in which tasks are annotated with worker IDs, resource configurations, and optimizations. For each task, the simulator estimates lots of metrics, such as whether the worker should be cold or warm, both the earliest feasible start time and the corresponding completion time, allowing it to derive the expected overall *makespan* of the workflow as well as the critical path.

This simulation capability significantly facilitates the work of planning components. Planners can experiment with alternative resource configurations, task placement policies, and co-location strategies without executing the workflow in a live environment. The fidelity of the simulation results, however, is directly dependent on the accuracy of the predictive models exposed through the *Predictions API*.

3.4.2 Planning Algorithms

For each task, the planner assigns both a `worker_id` and a resource configuration (vCPUs and memory). The `worker_id` specifies the worker instance that must execute the task—analogous to the "colors" in Palette Load Balancing [12], but in our case this assignment is mandatory rather than advisory, giving strict control over execution locality. Two tasks assigned the same `worker_id` should be executed on the same worker instance. If `worker_id` is not specified, workers will, at run-time, have to decide whether to execute or delegate those tasks, similar to WUKONG's [20] scheduling. We refer to these workers as *"flexible workers"*.

Users can select from three provided planners or implement their own planner by implementing an interface. All planners have access to the predictions API as well as the workflow simulation. The planners the user can choose from are the following:

1. **WUKONG**: All tasks will use the same worker configuration (specified by the user) and won't be assigned a `worker_id`, meaning they will be executed by *"flexible workers"*. This is a more dynamic scheduling approach where tasks aren't tied to specific workers that try to reproduce WUKONG's scheduling behavior;

2. **Uniform**: Tasks share a common worker configuration specified by the user, with each task assigned a `worker_id` to allow for co-location of tasks.

3. **Non-Uniform**: Tasks can use different worker configurations (list of available resources is specified by the user). Each task is assigned a `worker_id`. This algorithm starts by assigning the best available resources to all tasks. Then it runs a resource downgrading algorithm that attempts to downgrade resources of workers *outside the critical path* as much as possible without introducing a new critical path.

Both the **Uniform** and **Non-Uniform** planners follow a two-phase approach for task allocation: resource configuration assignment followed by worker ID assignment. The planners differ in their resource allocation strategies. The **Uniform planner** applies a single, user-specified CPU and memory configuration to all tasks, while the **Non-Uniform planner** selects the most powerful configuration from the user-specified options for each task. After resource configuration, both planners employ the logic detailed in Algorithm 3 for worker ID assignment. This algorithm implements an intelligent clustering strategy with two primary objectives: launch as few new workers as possible while trying not to overload workers; and minimizing network data transfers by co-locating tasks whose outputs are expected to be larger. This clustering approach provides an additional benefit of reducing fan-in operation costs.

After this, the **Non-Uniform** planner runs an additional algorithm, shown in Algorithm 4, that attempts to downgrade resources of workers *outside the critical path* as much as possible without introducing a

Algorithm 1 Worker Assignment Algorithm

```
Require: nodes, predictions, base_rc, SLA, MAX_CLUSTERING
assigned ← ∅
                                                        ▷ nodes are topologically sorted
for all n ∈ nodes do
    if n ∈ assigned then
        continue
    end if
    if n.upstream = ∅ then                                              ▷ root nodes
        roots ← {r ∈ nodes | r.upstream = ∅ ∧ r ∉ assigned}
        AssignGroup(null, roots)
    else if |n.upstream| = 1 then                                       ▷ 1→1 or 1→N
        u ← n.upstream[0]
        if |u.downstream| = 1 then
            AssignWorker([n], u.worker)                                 ▷ reuse worker
        else                                                            ▷ 1→N
            fanout ← {d ∈ u.downstream | d ∉ assigned}
            AssignGroup(u.worker, fanout)
        end if
    else                                                                ▷ N→1
        outputs ← {u.worker : predictions.output_size(u) | u ∈ n.upstream}
        best ← arg max_{w∈outputs} outputs[w]
        AssignWorker([n], best)
    end if
end for

function AssignGroup(up_worker, tasks)
    if tasks = ∅ then return
    end if
    exec_t ← {t : predictions.exec_time(t) | t ∈ tasks}
    out_sz ← {t : predictions.output_size(t) | t ∈ tasks}
    median ← Median(exec_t.values())
    longs ← {t ∈ tasks | exec_t[t] > median}
    shorts ← SortLargerOutputFirst({t ∈ tasks | exec_t[t] ≤ median})

                        ▷ 1) cluster short tasks with bigger outputs on upstream worker
    if up_worker ≠ null ∧ shorts ≠ ∅ then
        cluster ← shorts[0 : MAX_CLUSTERING]
        AssignWorker(cluster, up_worker)
        shorts ← shorts[MAX_CLUSTERING :]
    end if

                ▷ 2) pair long tasks with remaining short tasks (1 long per group)
    while longs ≠ ∅ ∧ shorts ≠ ∅ do
        cluster ← [longs[0]] + shorts[0 : MAX_CLUSTERING − 1]
        worker_id ← NewWorkerId
        AssignWorker(cluster, worker_id)
        longs ← longs[1 :]
        shorts ← shorts[MAX_CLUSTERING − 1 :]
    end while

                                                    ▷ 3) group remaining short tasks
    while shorts ≠ ∅ do
        worker_id ← NewWorkerId
        AssignWorker(shorts[0 : MAX_CLUSTERING], worker_id)
        shorts ← shorts[MAX_CLUSTERING :]
    end while

                                            ▷ 4) group remaining longs (half-size)
    half ← max(1, ⌊MAX_CLUSTERING/2⌋)
    while longs ≠ ∅ do
        worker_id ← NewWorkerId
        AssignWorker(longs[0 : half], worker_id)
        longs ← longs[half :]
    end while
end function
```

new critical path, by iteratively simulating the effect of downgrading resources of workers *outside the critical path* with different configurations.

With the information they have access to, planners can estimate whether it is worthwhile to offload a task to a more powerful worker. This involves weighing the overhead of uploading the input data, waiting for the worker to be provisioned, and then executing the task, against the alternative of simply executing the task on the current, less powerful worker.

3.4.3 Optimizations

Aside from their `worker_id` and resource assignments, planners can also apply different **optimizations** to further improve the workflow execution. The optimizations to be used are selected by the user, as shown in Listing 3.3. Similarly to planners, we provide three optimizations: **pre-warm**, **pre-load** and **task-dup**, but it is also possible to create new optimizations and define how workers should react to them. Now, we will describe the three base optimizations and how they are assigned to tasks:

1. **pre-warm**(worker_config) [Pre-warming Workers]:

    - *Interpretation*: Tasks/Nodes with this optimization should perform a special invocation to the FaaS gateway that forces it to launch a new worker with the specified resource configuration `worker_config`. This can be used to warm up workers ahead of time and mask cold start latencies;

    - *Assignment Logic*: For the nodes whose workers are expected to have cold starts, find the "optimal" node to perform the pre-warming by searching for nodes/tasks whose activity timing falls within a window (goal: avoid the pre-warmed worker from going cold before needed, while also not being warm too late). The optimization will be added to the "optimal" node, which will be responsible for doing the special "empty invocation" to the FaaS gateway.

2. **pre-load** [Pre-Loading Dependencies]:

    - *Interpretation*: Workers assigned to tasks or nodes with this optimization should begin downloading the task's dependencies as early as possible. This prevents scenarios where a worker must fetch all dependencies at once. The optimization is effective only if the worker is active before executing the task, allowing it to download dependencies in parallel with other ongoing tasks. This is implemented by having the worker receive completion notifications from the *Metadata Storage* for all tasks upstream of the optimized task;

    - *Assignment Logic*: First, all tasks with more than a configurable threshold of dependencies/upstream tasks are assigned the optimization. Then, an iterative process is used to

Algorithm 2 Resource Downgrading Algorithm

Require: $dag, nodes, critical\_path\_ids, original\_cp\_time, configs, predictions$

$workers\_outside \leftarrow \emptyset$

▷ 1) Identify workers outside the critical path

for all $n \in nodes$ **do** ▷ nodes are topologically sorted
    $wid \leftarrow n.worker\_id$
    if $n.id \notin critical\_path\_ids \land \forall cp \in dag.critical\_path\_nodes : wid \neq cp.worker\_id$ **then**
        $workers\_outside \leftarrow workers\_outside \cup \{wid\}$
    end if
end for
$nodes\_outside\_cp \leftarrow \{n \in nodes \mid n.id \notin critical\_path\_ids\}$

▷ 2) Attempt downgrade for each worker outside critical path

for all $wid \in workers\_outside$ **do**
    $last\_good\_rc \leftarrow \{n.id : n.config \mid n \in nodes\_outside\_cp \land n.worker\_id = wid\}$

▷ Iterate through weaker configurations (skip strongest at index 0)

    for $i \leftarrow 1$ **to** $|configs| - 1$ **do**
        $trial \leftarrow configs[i].$CLONE$(wid)$

▷ Apply trial configuration to all nodes of this worker

        for all $n \in nodes\_outside\_cp$ **do**
            if $n.worker\_id = wid$ **then**
                $n.config \leftarrow trial$
            end if
        end for

▷ Recompute workflow timing with predictions

        $cp\_time \leftarrow$ SIMULATECRITICALPATHTIME(dag)

        if $cp\_time = original\_cp\_time$ **then**
            ▷ Downgrade acceptable, record as last good state
            for all $n \in nodes\_outside\_cp$ **do**
                if $n.worker\_id = wid$ **then**
                    $last\_good\_rc[n.id] \leftarrow n.config$
                end if
            end for
        else
            ▷ Downgrade increases critical path, revert and move on to the next worker
            for all $n \in nodes\_outside\_cp$ **do**
                if $n.worker\_id = wid$ **then**
                    $n.config \leftarrow last\_good\_rc[n.id]$
                end if
            end for
            break ▷ move to next worker
        end if
    end for
end for

optimize the workflow along its critical path using this optimization. First, the algorithm identifies the critical path and assigns the optimization to eligible nodes on it. The critical path is then recalculated: if the total execution time increases, the optimization is removed; if the execution time decreases but the critical path changes, the algorithm restarts with the new path. This process repeats until no further improvements are possible, or the algorithm hits a fixed iteration limit.

3. **task-dup** [Task Duplication]:

    - *Interpretation*: Tasks or nodes with this optimization can be executed by other workers if doing so helps unlock dependent tasks more quickly. The task could be "duplicated" by workers that depend on its output. It is a trade-off between performance and resource utilization, allowing potentially faster execution at the cost of using additional compute resources;
    - *Assignment Logic*: Assigned to all nodes whose execution time and input size do not exceed predefined thresholds. Whether duplication actually occurs is decided at run-time. The optimization targets fast tasks with small inputs, as these are inexpensive to duplicate in terms of both downloading dependencies and execution. This way, even if duplication turns out to be unnecessary, the impact on performance and resource usage remains minimal.

Because planners may sometimes lack sufficient information to make optimal decisions about optimization assignments, it is important to not only allow the user to select optimizations at the workflow-level, but also allow them the flexibility to specify optimizations at the *task-level*. An example of this feature is shown in Listing 3.1, where the user requests that `task_b` attempt to use the *pre-load* optimization.

Once these optimizations are assigned, workflow planning is complete, and workers can begin execution. Because planning occurs on the user's machine (i.e., the machine launching the workflow), it is responsible for initiating the workflow by starting the initial workers. From that point onward, workers dynamically invoke additional workers as needed, following a choreographed, decentralized execution model.

To illustrate this execution model, Figure 3.6 provides a visual trace of how a planned workflow would be executed. The diagram depicts the workflow with the optimizations and `worker_id` assignments for each task. The non-dashed arrows represent task dependencies, while the dashed arrows represent interactions with the *Intermediate Storage* to either upload or download task data. We can see that task outputs are only uploaded to storage when there is at least one downstream task that depends on it and is assigned to another worker.

It is also worth noting that the planner assigned `Task 6` to `Worker 2`. This decision might be due to `Worker 2` being more powerful than `Worker 1`, and because the output of `Task 5` is larger than that of `Tasks 4` and `5`. Therefore, even if the task were executed on a more powerful worker (such as `Worker 3`,

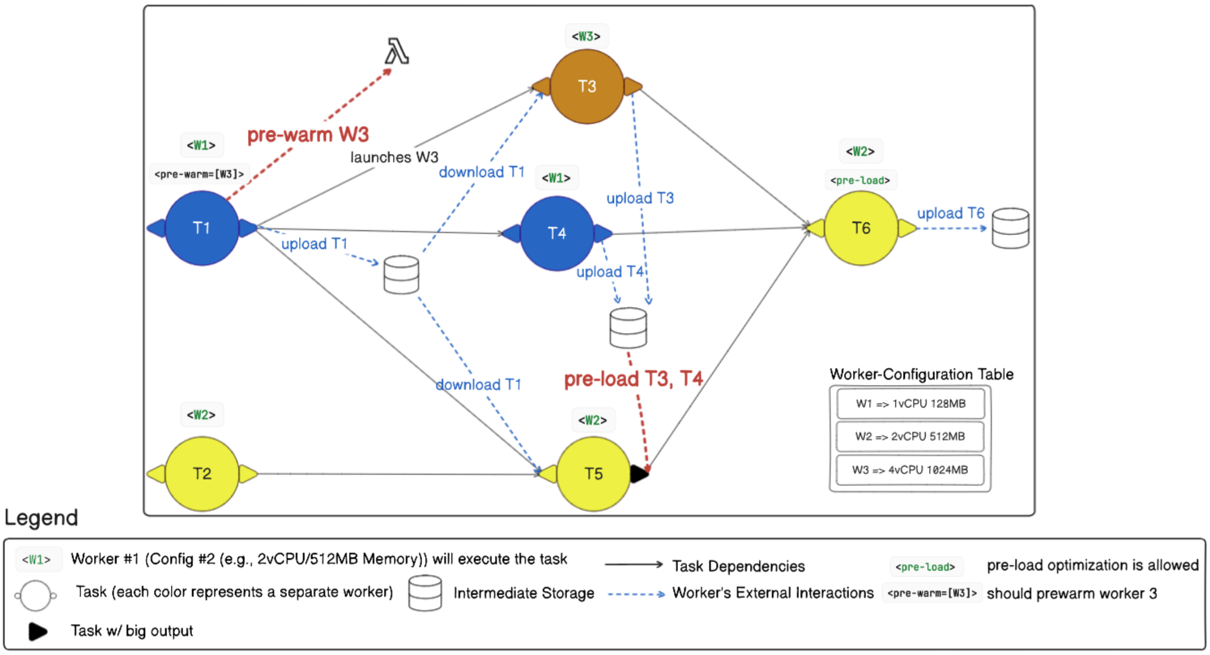

Figure 3.6: Planned Workflow Execution Example

which handled `Task 3`), the potential performance gain would not offset the additional time or resources required. This is an example of a planner deciding to co-locate `Tasks 5 and 6` on the same worker to reduce data movement.

Regarding **optimizations**, we can see `Task 1 pre-warming Worker 3`, by making a dummy invocation to the FaaS gateway, in an attempt to make it available before `Task 3` needs it. The `pre-load` optimization is used in `Task 5`, where the planner decided that `Worker 2` should start downloading the external dependencies for `Task 6` (`Task 3` and `Task 4`) as soon as they are available. This `pre-loading` can begin as soon as `Task 6`'s dependencies are ready in storage, potentially overlapping with `Task 2`'s execution instead of `Task 5`, as shown in the figure.

3.5 Decentralized Scheduling

Since our target execution platform is FaaS, the worker logic is implemented as a FaaS handler. Due to the decentralized nature of our solution, workers will be responsible for performing both task execution and scheduling in a choreographed manner.

When invoked, a worker receives the `workflow_id` and the `task_ids` of the tasks it should execute first. Using this information, it retrieves the DAG structure and execution plan from *Metadata Storage*. Rather than immediately executing the initial tasks, the worker first subscribes to `TASK_READY` and `TASK_COMPLETED` events for specific tasks. These events are essential both for enabling certain optimiza-

Algorithm 3 Worker Assignment Algorithm

Require: $nodes, predictions, base\_rc, SLA, MAX\_CLUSTERING$

$assigned \leftarrow \emptyset$
▷ nodes are topologically sorted
for all $n \in nodes$ **do**
  if $n \in assigned$ **then**
    continue
  end if
  if $n.upstream = \emptyset$ **then** ▷ root nodes
    $roots \leftarrow \{r \in nodes \mid r.upstream = \emptyset \wedge r \notin assigned\}$
    AssignGroup(*null*, $roots$)
  else if $|n.upstream| = 1$ **then** ▷ 1→1 or 1→N
    $u \leftarrow n.upstream[0]$
    if $|u.downstream| = 1$ **then**
      AssignWorker($[n]$, $u.worker$) ▷ reuse worker
    else ▷ 1→N
      $fanout \leftarrow \{d \in u.downstream \mid d \notin assigned\}$
      AssignGroup($u.worker$, $fanout$)
    end if
  else ▷ N→1
    $outputs \leftarrow \{u.worker : predictions.output\_size(u) \mid u \in n.upstream\}$
    $best \leftarrow \arg\max_{w \in outputs} outputs[w]$
    AssignWorker($[n]$, $best$)
  end if
end for

function AssignGroup(up_worker, $tasks$)
  if $tasks = \emptyset$ **then return**
  end if
  $exec\_t \leftarrow \{t : predictions.exec\_time(t) \mid t \in tasks\}$
  $out\_sz \leftarrow \{t : predictions.output\_size(t) \mid t \in tasks\}$
  $median \leftarrow$ Median($exec\_t.values()$)
  $longs \leftarrow \{t \in tasks \mid exec\_t[t] > median\}$
  $shorts \leftarrow$ SortLargerOutputFirst($\{t \in tasks \mid exec\_t[t] \leq median\}$)

  ▷ 1) cluster short tasks with bigger outputs on upstream worker
  if $up\_worker \neq$ *null* $\wedge\ shorts \neq \emptyset$ **then**
    $cluster \leftarrow shorts[0 : MAX\_CLUSTERING]$
    AssignWorker($cluster$, up_worker)
    $shorts \leftarrow shorts[MAX\_CLUSTERING :]$
  end if

  ▷ 2) pair long tasks with remaining short tasks (1 long per group)
  while $longs \neq \emptyset \wedge shorts \neq \emptyset$ **do**
    $cluster \leftarrow [longs[0]] + shorts[0 : MAX\_CLUSTERING - 1]$
    $worker\_id \leftarrow$ NewWorkerId
    AssignWorker($cluster$, $worker\_id$)
    $longs \leftarrow longs[1 :]$
    $shorts \leftarrow shorts[MAX\_CLUSTERING - 1 :]$
  end while

  ▷ 3) group remaining short tasks
  while $shorts \neq \emptyset$ **do**
    $worker\_id \leftarrow$ NewWorkerId
    AssignWorker($shorts[0 : MAX\_CLUSTERING]$, $worker\_id$)
    $shorts \leftarrow shorts[MAX\_CLUSTERING :]$
  end while

  ▷ 4) group remaining longs (half-size)
  $half \leftarrow \max(1, \lfloor MAX\_CLUSTERING/2 \rfloor)$
  while $longs \neq \emptyset$ **do**
    $worker\_id \leftarrow$ NewWorkerId
    AssignWorker($longs[0 : half]$, $worker\_id$)
    $longs \leftarrow longs[half :]$
  end while
end function

Algorithm 4 Resource Downgrading Algorithm

Require: $dag, nodes, critical\_path\_ids, original\_cp\_time, configs, predictions$

$workers\_outside \leftarrow \emptyset$

▷ 1) Identify workers outside the critical path

for all $n \in nodes$ **do** ▷ nodes are topologically sorted

  $wid \leftarrow n.worker\_id$

  if $n.id \notin critical\_path\_ids \wedge \forall cp \in dag.critical\_path\_nodes : wid \neq cp.worker\_id$ **then**

    $workers\_outside \leftarrow workers\_outside \cup \{wid\}$

  end if

end for

$nodes\_outside\_cp \leftarrow \{n \in nodes \mid n.id \notin critical\_path\_ids\}$

▷ 2) Attempt downgrade for each worker outside critical path

for all $wid \in workers\_outside$ **do**

  $last\_good\_rc \leftarrow \{n.id : n.config \mid n \in nodes\_outside\_cp \wedge n.worker\_id = wid\}$

  ▷ Iterate through weaker configurations (skip strongest at index 0)

  for $i \leftarrow 1$ **to** $|configs| - 1$ **do**

    $trial \leftarrow configs[i].$CLONE$(wid)$

    ▷ Apply trial configuration to all nodes of this worker

    for all $n \in nodes\_outside\_cp$ **do**

      if $n.worker\_id = wid$ **then**

        $n.config \leftarrow trial$

      end if

    end for

    ▷ Recompute workflow timing with predictions

    $cp\_time \leftarrow$ SIMULATECRITICALPATHTIME(dag)

    if $cp\_time = original\_cp\_time$ **then**

      ▷ Downgrade acceptable, record as last good state

      for all $n \in nodes\_outside\_cp$ **do**

        if $n.worker\_id = wid$ **then**

          $last\_good\_rc[n.id] \leftarrow n.config$

        end if

      end for

    else

      ▷ Downgrade increases critical path, revert and move on to the next worker

      for all $n \in nodes\_outside\_cp$ **do**

        if $n.worker\_id = wid$ **then**

          $n.config \leftarrow last\_good\_rc[n.id]$

        end if

      end for

      break ▷ move to next worker

    end if

  end for

end for

tions and for ensuring the worker follows the workflow plan correctly.

After that, the worker starts executing the initial tasks concurrently. The logic for executing tasks is the following:

1. **Gathering Dependencies**: Check which dependencies are missing (not downloaded yet) and download them from storage;

2. **Executing Task**: Execute the task. Tasks' code is stored in a serialized/pickled format (using cloudpickle[2]) and deserialized and executed by the workers. This enables the worker to remain generic, capable of receiving and executing arbitrary task code;

3. **Handling Output**: This phase is responsible for evaluating whether it's necessary to upload the task's output to storage and emitting a `TASK_COMPLETED` event;

4. **Delegating Downstream Tasks**: For each downstream task, the worker performs an `atomic_increment_and_get(` operation on a "dependency counter" (inspired by WUKONG [20]) stored in *Metadata Storage*, which tracks how many dependencies of a task have been satisfied. When the counter indicates that all dependencies for a downstream task have been satisfied, the worker emits a `TASK_READY` event for that task. The worker then consults the execution plan to determine how to proceed for each downstream task unlocked: if the unlocked task is assigned to another worker, a `TASK_READY` event is emitted; if the unlocked task is assigned to the same worker and has no remaining dependencies, the worker immediately continues the cycle by executing it.

To illustrate how workers handle downstream tasks, Figure 3.7 presents an example of choreographed decentralized scheduling with three workers (A, B, and C) and seven tasks (T1-T7). In this example, once Task T1 completes on Worker B, the worker inspects the dependency counters for tasks T3, T4, and T5. It determines that T3 and T4 are ready to run, while T5 is still pending because Task T2 has not yet completed. Worker B then launches a new worker (C) to execute T3 and proceeds to execute T4 itself. Later, when T2 finishes on Worker A, all dependencies of T5 are satisfied, prompting Worker A to execute it directly.

A workflow is considered complete once the output of the final (sink) task is available in storage. The worker that uploads this final result is also responsible for cleaning up all intermediate results before shutting down. Meanwhile, after submitting the workflow, the user's machine subscribes to the `TASK_COMPLETED` event for the sink task; upon receiving this notification, it retrieves the final result from *Intermediate Storage*.

By delegating downstream tasks to workers, our approach eliminates the need for a central scheduler, a common component in many existing FaaS-based workflow engines. This decentralization in-

[2] https://github.com/cloudpipe/cloudpickle

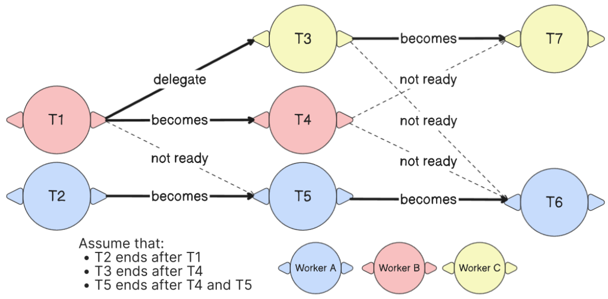

Figure 3.7: Choreographed Scheduling Example

creases flexibility and scalability, as workers can dynamically invoke additional workers as needed, following a choreographed decentralized execution model.

Having described the design and implementation of the system, we now turn to its evaluation. The next section presents the experimental setup, results, and analysis used to assess the strengths and weaknesses of our approach.

4

Evaluation

Contents

4.1 Maecenas vitae nulla consequat

5

Conclusion

Contents

5.1 Conclusions

5.2 System Limitations and Future Work

Bibliography

[1] Aws lambda. [Online]. Available: https://aws.amazon.com/pt/lambda/

[2] Azure functions. [Online]. Available: https://azure.microsoft.com/en-us/products/functions

[3] Google cloud run functions. [Online]. Available: https://cloud.google.com/functions

[4] R. Graves, T. H. Jordan, S. Callaghan, E. Deelman, E. Field, G. Juve, C. Kesselman, P. Maechling, G. Mehta, K. Milner, D. Okaya, P. Small, and K. Vahi, "Cybershake: A physics-based seismic hazard model for southern california," *Pure and Applied Geophysics*, vol. 168, no. 3, pp. 367–381, 2011. [Online]. Available: https://doi.org/10.1007/s00024-010-0161-6

[5] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. Prince, and R. Williams, "Montage: A grid portal and software toolkit for science-grade astronomical image mosaicking," *International Journal of Computational Science and Engineering*, vol. 4, no. 2, pp. 73–87, 2009. [Online]. Available: https://doi.org/10.1504/IJCSE.2009.026999

[6] M. Golec, G. K. Walia, M. Kumar, F. Cuadrado, S. S. Gill, and S. Uhlig, "Cold start latency in serverless computing: A systematic review, taxonomy, and future directions," *ACM Computing Surveys*, vol. 57, no. 3, pp. 1–36, 2024.

[7] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018.

[8] Aws step functions. [Online]. Available: https://aws.amazon.com/en/step-functions/

[9] Azure durable functions. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview

[10] Google cloud workflows. [Online]. Available: https://cloud.google.com/workflows

[11] F. Romero, G. I. Chaudhry, I. n. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "Faa$t: A transparent auto-scaling cache for serverless applications," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New

York, NY, USA: Association for Computing Machinery, 2021, p. 122–137. [Online]. Available: https://doi.org/10.1145/3472883.3486974

[12] M. Abdi, S. Ginzburg, C. Lin, J. M. Faleiro, I. Goiri, G. I. Chaudhry, R. Bianchini, D. S. Berger, and R. Fonseca, "Palette load balancing: Locality hints for serverless functions," in *EuroSys*. ACM, May 2023. [Online]. Available: https://www.microsoft.com/en-us/research/publication/palette-load-balancing-locality-hints-for-serverless-functions/

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

[14] Apache openwhisk. [Online]. Available: https://openwhisk.apache.org/

[15] D. Moyer and D. S. Nikolopoulos, "Punching holes in the cloud: Direct communication between serverless functions," in *Serverless Computing: Principles and Paradigms*. Springer, 2023, pp. 15–41.

[16] M. Yu, T. Cao, W. Wang, and R. Chen, "Pheromone: Restructuring serverless computing with data-centric function orchestration," *IEEE Transactions on Networking*, vol. 33, no. 1, pp. 226–240, 2025.

[17] P. G. López, A. Arjona, J. Sampé, A. Slominski, and L. Villard, "Triggerflow: trigger-based orchestration of serverless workflows," in *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 3–14. [Online]. Available: https://doi.org/10.1145/3401025.3401731

[18] C. P. Smith, A. Jindal, M. Chadha, M. Gerndt, and S. Benedict, "Fado: Faas functions and data orchestrator for multiple serverless edge-cloud clusters," in *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*, 2022, pp. 17–25.

[19] M. Copik, R. Böhringer, A. Calotoiu, and T. Hoefler, "Fmi: Fast and cheap message passing for serverless functions," in *Proceedings of the 37th ACM International Conference on Supercomputing*, ser. ICS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 373–385. [Online]. Available: https://doi.org/10.1145/3577193.3593718

[20] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proceedings of the 11th ACM symposium on cloud computing*, 2020, pp. 1–15.

[21] D. H. Liu, A. Levy, S. Noghabi, and S. Burckhardt, “Doing more with less: Orchestrating serverless applications without an orchestrator,” in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 1505–1519.

[22] Q. Jiang, Y. C. Lee, and A. Y. Zomaya, “Serverless execution of scientific workflows,” in *International Conference on Service-Oriented Computing*. Springer, 2017, pp. 706–721.

[23] Cloudflare workers. [Online]. Available: https://workers.cloudflare.com/

[24] Openfaas. [Online]. Available: https://www.openfaas.com/

[25] Knative. [Online]. Available: https://knative.dev/docs/

[26] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, “Encoding, fast and slow:{Low-Latency} video processing using thousands of tiny threads,” in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 363–376.

[27] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, “Serverless computing: State-of-the-art, challenges and opportunities,” *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1522–1539, 2023.

[28] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, “Pocket: Elastic ephemeral storage for serverless analytics,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 427–444. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/klimovic

[29] Apache airflow. [Online]. Available: https://airflow.apache.org/

[30] Dask - python parallel computing framework. [Online]. Available: https://www.dask.org/

[31] Dagman. [Online]. Available: https://htcondor.readthedocs.io/en/latest/automated-workflows/dagman-introduction.html

[32] Htcondor. [Online]. Available: https://htcondor.org/

[33] W. M. Van Der Aalst and A. H. Ter Hofstede, “Yawl: yet another workflow language,” *Information systems*, vol. 30, no. 4, pp. 245–275, 2005.

[34] Petri nets. [Online]. Available: https://en.wikipedia.org/wiki/Petri_net

[35] Apache spark. [Online]. Available: https://spark.apache.org/

[36] Apache flink. [Online]. Available: https://flink.apache.org/

[37] Apache hadoop. [Online]. Available: https://hadoop.apache.org/

[38] Apache oozie. [Online]. Available: https://oozie.apache.org/

[39] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," vol. 51, no. 1. New York, NY, USA: Association for Computing Machinery, Jan. 2008, pp. 107–113. [Online]. Available: https://doi.org/10.1145/1327452.1327492

[40] Hadoop yarn. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[41] Dask distributed. [Online]. Available: https://distributed.dask.org/en/stable/

[42] Prefect. [Online]. Available: https://www.prefect.io/

[43] Dagster. [Online]. Available: https://dagster.io/

[44] Argo workflows. [Online]. Available: https://argoproj.github.io/workflows/

[45] Kubernetes. [Online]. Available: https://kubernetes.io/

[46] Google dataflow. [Online]. Available: https://cloud.google.com/products/dataflow?hl=en

[47] Cloudflare workflows. [Online]. Available: https://www.cloudflare.com/developer-platform/products/workflows/

[48] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

[49] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USA: USENIX Association, 2014, p. 305–320.

[50] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "{ZooKeeper}: Wait-free coordination for internet-scale systems," in *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

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

A

Code of Project

B

A Large Table