Serverless Dataflows - V2

PIC2 - Master in Computer Science and Engineering
Instituto Superior Técnico, Universidade de Lisboa

Diogo Alexandre Ferreira de Jesus — 111266*
diogofjesus@tecnico.ulisboa.pt

Advisor: Luís Veiga

Abstract Serverless computing has gained traction as a promising alternative to traditional cloud computing. Its appeal lies in its ease of operation, automatic scalability, and cost-efficient pricing model, which have fueled its adoption across diverse use cases. However, the efficient execution of workflows, particularly those involving substantial data exchange between tasks, remains a challenge on serverless platforms. This difficulty arises from the inherently stateless nature of serverless functions, which forces functions to perform indirect communication through external services, often leading to performance inefficiencies and additional costs.

Several approaches have been introduced to mitigate these issues, such as stateful functions and serverless runtime extensions. Furthermore, innovative orchestrators and schedulers have been developed to take advantage of serverless architectures. This document examines the challenges and opportunities in serverless workflow execution and presents the architecture of a solution for improving serverless workflow scheduling by leveraging historical data from past executions.

Unlike traditional single-step scheduling methods typically employed by other schedulers, our approach takes advantage of having a global-view of the workflow and its execution context, including performance and resource metrics. The solution of the proposed architecture incorporates a *Metadata Management* component to collect and store task metadata, a *Static Workflow Planner* to annotate the workflow with scheduling decisions and optimizations, and a *Scheduler* integrated into workers to execute that plan, applying optimizations when needed. This solution aims to enhance both performance and resource utilization, while also being flexible, allowing custom policies to be implemented through different algorithms that plan the workflow.

Evaluation will be performed by comparing two different planning algorithms against the three different WUKONG data locality optimizations. This will help us draw conclusions about the impact and potential of using historical metadata, such as task performance statistics, to enhance serverless workflow scheduling.

Keywords — Serverless, FaaS, Workflows, Scheduling, Cloud Computing

*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 (https://nape.tecnico.ulisboa.pt/en/apoio-ao-estudante/documentos-importantes/regulamentos-da-universidade-de-lisboa/).

Contents

1 Introduction

Serverless computing, with its Function-as-a-Service (FaaS) model, offers a compelling alternative to traditional cloud computing paradigms. By abstracting infrastructure management and allowing developers to concentrate on business logic, serverless architectures offer operational simplicity, automatic scalability, and a pay-per-use pricing model. These advantages have led to widespread adoption of serverless platforms for various applications, including event-driven systems, microservices, IoT processing, and web services, with the most relevant commercial offerings being AWS Lambda [1], Google Cloud Run Functions [2], and Azure Functions [3].

In this setting, however, the efficient execution of complex workflows on serverless platforms remains a significant challenge. While serverless platforms excel at handling embarrassingly parallel jobs with uniform, short-duration tasks, they struggle with workflows involving substantial data exchange between tasks [4]. This limitation stems from the stateless nature of serverless functions, which hinders direct communication and necessitates reliance on external storage for data transfer. Consequently, workflows with intricate data dependencies often suffer from performance degradation and increased latency.

Various approaches have been proposed to address these challenges. Commercial cloud platforms offer *stateful functions* [5] [6] [7], which make it easier to manage complex workflows made up of *stateless functions*, but use their own workflow languages, and are tightly coupled with their cloud ecosystem. Solutions such as Palette [8], Faa$T [9], and Lambdata [10] propose extensions to serverless runtimes in general, to improve data locality and provide it with some level of statefulness, with some being more transparent to the user than others.

Building on the current state of serverless platforms, new orchestrators and schedulers have been implemented to take advantage of the benefits brought forth by this new cloud computing model, while also assessing its effectiveness and applicability. PyWren [11] is an orchestrator that leverages serverless stateless functions for parallel computations, but lacks data locality optimizations, focusing on *embarrassingly parallel* workflows. On the serverless scheduling side, WUKONG 2.0 [12] showcases promising performance improvements by using decentralized scheduling and locality optimizations to enhance scalability and minimize data transfers.

Despite these advancements, existing solutions often employ a *one-step scheduling* approach, making decisions based solely on the immediate workflow stage without considering the global implications. We believe that, by leveraging metadata from previous workflow runs, we can learn about individual tasks behavior and use that information to improve scheduling decisions and perform optimizations that reduce workflow *makespan* and *increase resource efficiency*.

1.1 Work Objectives

Our expected contributions with this work are the following:

- Discuss the current landscape of serverless workflows, highlighting its limitations and opportunities, and presenting some of the most relevant works in this area;
- Propose the architecture of a solution aiming to enhance serverless workflow scheduling by leveraging historical data from previous workflow runs. The architecture of the proposed solution incorporates the following components:
  - **Metadata Management:** Collects and stores metadata of tasks from previous executions, and exposes an API that uses this metadata to provide predictions about execution times, data transfer times, and task output sizes;
  - **Static Workflow Planner:** It runs a user-selected algorithm to *plan* the workflow. This algorithm will leverage the API exposed by *Metadata Management* to *annotate* the DAG tasks. Annotations specify which worker instance should run each task and suggest optimizations, such as *pre-load*, *pre-warm*, and *task-duplication*, to be performed at run-time;
  - **Scheduler:** This component, integrated into the workers, schedules tasks and performs optimizations based on the *annotated DAG* produced by the **Static Workflow Planner**.
- Integrate our solution in an existing decentralized serverless scheduler, WUKONG 2.0, since we believe it to be the solution that exploits serverless computing benefits the best. The high-level modifications and extensions required are presented in this document;
- Evaluate our solution to assess its performance, scalability, data movement, resource efficiency, cost-effectiveness, and identify potential bottlenecks. This will allow us to draw conclusions on how it compares against one-step scheduling approaches, such as WUKONG's.

1.2 Document Roadmap

In this document, Section 2 provides insights on the current serverless computing paradigm, workflows in general, and modern solutions that exploit the advantages of this emergent cloud computing model. The architecture of our proposed solution is shown and described in detail in Section 3. In Section 4, we explain how we will evaluate our solution. This involves enumerating the workloads to be used, the metrics to be collected, and what schedulers we plan to compare against. We conclude in Section 5.

2 Related Work

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

Serverless computing represents a paradigm shift in how applications are developed and deployed. At its core, it allows developers to focus on business logic without caring about the underlying infrastructure. This is achieved through the function-as-a-service (FaaS) model, implemented by all major cloud providers, such as Amazon(Lambda [1]), Google(Cloud Run Functions [2]), and Azure(Functions [3]). Some open-source FaaS runtimes include OpenWhisk [13], OpenFaaS [14], Knative [15], and Kubeless [16] (deprecated).

The biggest upsides of the serverless model are the following:

- **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 unpredictable usage patterns, where we don't know *how many* or *when* requests will arrive;
- **Pay-per-use** means 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).

Li et al. [17] surveys the serverless computing landscape, highlighting its benefits, challenges and research opportunities. 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. [4] 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.

To overcome these limitations and explore the opportunities of serverless computing, *"Serverless Computing: State-of-the-Art, Challenges and Opportunities"* (Li et al.) gathered research being carried out on several fronts. Applying *machine learning* to resource management aims to optimize resource allocation and predict application needs, reducing costs and latencies. Although promising, the latency requirements and complexity of the settings of the machine learning approaches are not yet satisfactory. Integration with edge computing, driven by 5G, offers the opportunity to leverage the computing power of thousands of edge devices to provide low-latency services. This survey also hints at the importance of developing comprehensive and representative benchmarks to evaluate and compare different platforms and guide the design of serverless applications.

2.1.1 Serverless Architecture

The core of a typical serverless/FaaS runtime is composed of 4 layers (as depicted in Figure 1): Controller, Workers, Storage, and Monitoring. The **Controller** is responsible for forwarding requests to workers (often run in containers) and scaling (up and down) the number of available workers at any given point in time. **Workers** receive invocation requests from the controller and have to prepare the runtime, download the function code from storage, run it and return a response to the controller. The **Storage** component stores user-specific data, functions code, metadata, etc. Also, Serverless runtimes usually contain **Monitoring** components that send alerts to the controller to help it make load balancing and scaling decisions. Besides that, this module is particularly important for commercial platforms because it is what allows them to use metrics such as function execution time and memory usage to charge users.

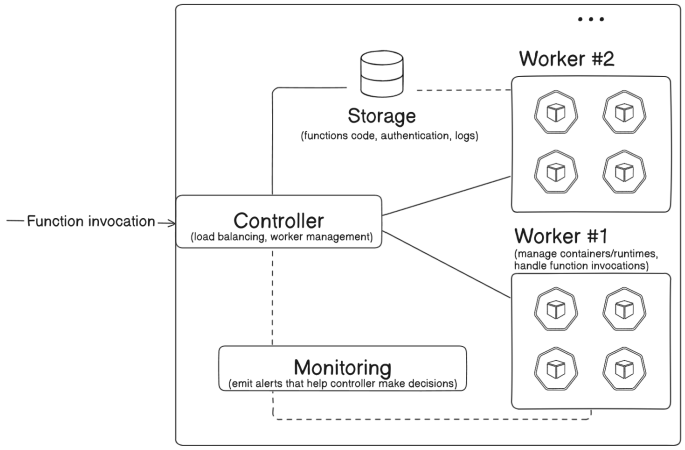

Figure 1: Typical Serverless Platform Architecture

When a function is invoked in a serverless environment, the runtime must *allocate the necessary compute resources*, which can include provisioning a container. It then needs to *retrieve the function code* from the storage system, as well as any required dependencies or environment configurations. After the code is loaded, the worker *initializes the execution environment*, which may involve setting up memory, network access, or other runtime settings. Only once these steps are complete can the worker *execute the function* and return the result. In a **cold start**, all of these steps are necessary and introduce additional latency (some function's execution time may even be shorter than the function bootstrapping phase). A **warm start** is when many of these steps are skipped. For instance, the function code may already be loaded in a worker's environment, and the execution context may be reused, allowing the function to execute sooner. Serverless runtimes aim to achieve warm starts by dynamically directing requests to workers who have already completed most of the necessary initialization steps.

Some research efforts focus on reducing cold start times, such as Nightcore (Jia et al. [18]), which is a FaaS platform designed for latency-sensitive, interactive microservices, achieving microsecond-scale overheads through *low-level* optimizations such as fast internal function calls and low-latency message channels. Other research papers try to avoid cold starts by intelligently caching execution environments. One example is Flame (Yang et al. [19]), a recent research project that seeks to avoid cold starts by improving function caching through a centralized cache controller. This controller employs *hotspot-aware scheduling*, which dynamically identifies and prioritizes

"hotspot" functions-those that are frequently invoked. By distributing these popular functions across servers to avoid resource contention, Flame enhances cache efficiency and reduces the likelihood of cold starts, leading to more effective resource utilization.

2.1.2 Use Cases

Due to its advantages, the serverless computing model has been gaining traction among companies and developers. The ability to build and deploy applications quickly, combined with cost-efficiency and effortless scalability, makes it an attractive choice for many types of applications. In this section, we will mention the most common types of applications deployed on serverless, explaining how this model fulfills their needs.

Generically, the serverless model naturally fits **stateless**, **event-driven**, **embarrassingly parallel**, and/or applications with **unpredictable frequency of access**. Types of applications with these properties include *real-time data processing*, such as log aggregation or stream processing. Serverless is also well-suited for *microservices* architectures, where each function operates as a small, independent service, and services scale based on demand. FaaS enables splitting up *microservices* into even smaller services (*nanoservices*).

Additionally, IoT applications benefit from serverless computing due to their need for processing large volumes of sensor data in near real-time, where the compute requirements can vary widely depending on the number of devices or events triggered at any given moment. Serverless models are also ideal for *short-lived jobs* that are executed on an on-demand basis, like image processing or video transcoding (ExCamera [20]) tasks, where computation can be divided into multiple parallel jobs.

Traditional web applications are particularly well-suited for serverless architectures due to their sporadic access, scalability requirements, computational simplicity and statelessness. Since serverless platforms rely on the principle of *statelessness*, where each function is independent and does not maintain state between invocations, they are ideal for applications that do not require complex session management or persistent connections. For example, functions that handle short-lived tasks like processing unrelated HTTP requests, generating dynamic content, or handling user inputs (e.g., file uploads) can execute quickly and efficiently without the need to maintain a running server. Furthermore, serverless platforms automatically scale to accommodate varying loads, allowing applications to handle spikes in demand without over-provisioning resources.

As a result, developers can focus on writing code rather than managing infrastructure, while also benefiting from cost savings, as they only pay for the compute time used during function execution. This combination of automatic scaling, cost efficiency, and reduced operational overhead makes serverless an attractive choice for stateless web applications as well as the use cases described previously.

2.1.3 Commercial Serverless Platforms

Here, we will give examples of the main serverless offerings from commercial cloud platforms. Among the most notable are AWS Lambda [1], Google Cloud Run Functions [2] (replacement of Google Cloud Functions), and Azure Functions [3]. Other platforms with different approaches to serverless are Cloudflare Workers [21] and Akamai EdgeWorkers [22]. All of them follow the serverless model mentioned previously, where the user submits function code (in a variety of programming languages), defines triggers that activate functions, the resources required, and desired concurrency limits. Their differences lie in maximum execution time, resource allocation limits, supported programming languages, pricing, and event sources (integration with services from the same provider is usually easier and more performant).

AWS Lambda, introduced by Amazon Web Services in 2014, is a pioneer in the serverless computing space and remains one of the most mature and widely adopted platforms. AWS Lambda's extensive integration with other AWS services provides seamless event-driven architecture capabilities. Its robust ecosystem, comprehensive documentation, and strong community support make it an excellent reference point for comparing other serverless offerings like Google Cloud Run Functions and Azure Functions. We will also briefly write about an *edge* serveless computing platform (Cloudflare Workers [21]) to highlight the main differences and limitations in comparison with the previous solutions mentioned.

AWS Lambda Properties:

- **Event Sources**: Supports a wide range of event sources including S3, DynamoDB, Kinesis, SNS, and API Gateway;
- **Execution Time**: Maximum execution time of 15 minutes;

- **Languages Supported**: Javascript, Python, Ruby, Java, Go, .NET Core, and custom runtimes via AWS Lambda Layers;
- **Resource Allocation**: Configurable memory allocation up to 10,240 MB (and 6vCPU);
- **Pricing**: Pay-per-use model based on the number of requests and the compute time over memory.

Google Cloud Run Function Properties:

- **Event Sources**: Supports Google Cloud Storage, Firestore, Pub/Sub, HTTP triggers and many other events compatible with Eventarc [23];
- **Execution Time**: Maximum execution time of 60 minutes for HTTP-triggered functions (more than AWS Lambda's 15 minutes), and 9 minutes for event-triggered functions;
- **Languages Supported**: Javascript, Python, Go, Java, Ruby, PHP, .NET Core;
- **Resource Allocation**: Configurable memory allocation up to 17,179 MB (and 4vCPU);
- **Pricing**: Similar pay-per-use model, with slight variations in pricing structure and free tier limits;

Azure Functions operates differently from other serverless platforms by offering multiple hosting plans, each with its own set of limitations, allowing users to choose the most suitable option for their specific needs. These plans are designed to cater to varying levels of performance, scaling, and resource allocation, and you can find more details about the different plans and their respective limitations in the official documentation.

Azure Functions Properties:

- **Event Sources**: Supports a wide range of event sources including Azure Event Hubs, Azure Cosmos DB, Azure Storage Blobs, Azure Service Bus, and HTTP triggers;
- **Execution Time**: Maximum execution time of 5 minutes (can be extended to 60 minutes with the Premium plan);
- **Languages Supported**: C#, JavaScript, Python, Java, PowerShell, and TypeScript;
- **Resource Allocation**: Memory allocation limit is 1,610 MB (0.5 vCPU);
- **Pricing**: The *Consumption Plan* is based on the number of executions and execution time over memory used, with additional options (pay for *pre-warmed* instances) for Premium and Dedicated (App Service) plans.

In addition to the *Consumption Plan*, Azure Functions offers the *Premium Plan*, *Flex Consumption Plan*, and *Dedicated (App Service) Plan*. The Premium Plan extends the maximum execution time to 60 minutes, and provides more consistent performance with *pre-warmed* instances to reduce startup latencies. The Flex Consumption Plan also offers pre-warmed instances reducing cold starts, while also allowing for longer execution durations and more memory (up to 4096 MB (1 vCPU)) options. The Dedicated (App Service) Plan offers even greater flexibility by allowing functions to run on dedicated VMs, which can be scaled manually or automatically based on demand, while also including other features. Here we focused on the *Consumption Plan*.

Cloudflare Workers Properties:

- **Event Sources**: Primarily triggered by HTTP requests, but with support for Cron triggers as well;
- **Execution Time**: Maximum execution time of 30 seconds per HTTP request, 15 minutes for Cron triggers;
- **Languages Supported**: JavaScript, TypeScript, Python, Rust, and WebAssembly (C, C++, Kotlin, and Go can be compiled to WASM);
- **Resource Allocation**: Fixed memory allocation up to 128 MB;

To sum it up, AWS Lambda is deeply integrated with the AWS ecosystem, providing extensive support for various AWS services and third-party tools. It also offers robust monitoring and debugging features through AWS CloudWatch. Google Cloud Run Functions and Azure Functions, while offering similar core functionality, are optimized for their respective cloud ecosystems. On the other end of the spectrum, Cloudflare Workers are built on top of edge computing, allowing developers to run code close to the end user, achieving lower latency for *Web Applications*. They are particularly useful for tasks like handling HTTP requests, caching, and performing real-time data transformations for IoT applications (which already use these technologies). As edge serverless platforms (e.g., Cloudflare Workers, Akamai EdgeWorkers [22], Lambda@Edge [24]) run on edge devices, they usually have more strict limits on network bandwidth, execution time, memory usage, and computational power, making them mostly suitable for stateless and lightweight Web or IoT applications with low-latency requirements.

2.1.4 Open-Source Serverless Platforms

While commercial serverless platforms offer powerful solutions for deploying and managing serverless applications, they often come with constraints such as vendor lock-in, limited customization, and potentially higher costs at scale. Open-source serverless platforms offer greater flexibility, transparency, and control over the infrastructure. They enable organizations to tailor the serverless environment to their specific needs and integrate seamlessly with existing systems. In this section, we will briefly explore the architectures of two of the most relevant open-source serverless platforms: Apache OpenWhisk [13] and OpenFaaS [14], highlighting their features, advantages, and unique characteristics.

Apache OpenWhisk Apache OpenWhisk is a widely adopted open-source serverless platform designed to enable developers to build and deploy function-based applications with high efficiency. One of the key architectural elements of OpenWhisk is its *trigger-based mechanism*, where functions, known as *actions*, are executed in response to *events*, or *triggers*, from various sources. These sources can include HTTP requests, database changes, message queues, or custom events.

The architecture of OpenWhisk is *modular and highly extensible*, consisting of several core components:

- **Controller**: The controller is the central component that handles incoming API requests. It routes these requests to the appropriate components, manages user authentication, and maintains metadata about actions, triggers, and rules;
- **Invoker**: Invokers are responsible for executing actions. Each invoker runs within a Docker container, ensuring that functions are isolated and can be scaled independently. This containerized approach not only enhances security and isolation but also leverages Docker's capabilities for resource management and rapid deployment;
- **Message Broker**: OpenWhisk uses a message broker, typically Apache Kafka, to facilitate communication between the controller and the invokers. This decouples the components, allowing for more scalable and resilient operation;
- **Database**: Metadata about actions, triggers, rules, and activations (the records of function executions) are stored in a CouchDB database. This database enables efficient querying and retrieval of information necessary for the orchestration of serverless workflows;
- **Load Balancer**: OpenWhisk includes a load balancer that distributes incoming requests among available invokers, ensuring optimal resource utilization and maintaining performance under varying loads.

OpenWhisk allows a wide variety of programming languages by supporting language-specific runtime environments packaged as Docker images. Its flexible *event-driven model* allows complex workflows to be defined using rules that link triggers and actions. For example, a single trigger can invoke multiple actions, or an action can be configured to invoke other actions in sequence, enabling the construction of sophisticated serverless applications. OpenWhisk also integrates with tools like Logstash [25] for collecting logs and metrics that can be visualized using Grafana [26] custom dashboards. This comprehensive monitoring suite makes it easier to manage, debug, and optimize serverless applications.

OpenWhisk's modular architecture makes it attractive for research and academic advancements in serverless computing. Lambdata [10], for instance, tested its data locality optimizations by extending OpenWhisk's Controller and Invoker to allow function invokers to provide *hints* about what input and output objects will be accessed.

OpenFaaS OpenFaaS (Functions as a Service) is a serverless platform designed to simplify the deployment and management of functions at scale. It is built on top of Kubernetes and Docker, taking advantage of their powerful orchestration and containerization capabilities. OpenFaaS provides an easy-to-use interface through both a web UI and command-line interface (CLI), making it accessible for developers looking to deploy serverless functions in any language that can be compiled into a Docker image, similarly to Apache OpenWhisk.

The architecture of OpenFaaS is centered around Kubernetes, with several key components contributing to its functionality:

- **Gateway**: The gateway is the main entry point for OpenFaaS, receiving incoming HTTP requests and routing them to the appropriate function. It handles API calls, scaling, and manages the execution of functions, ensuring they are triggered as required;
- **Function Executor**: Functions in OpenFaaS run as containers managed by Kubernetes. Each function is deployed as a Docker container, isolated and able to scale independently based on demand. This enables quick provisioning and execution of functions, while also providing high availability and fault tolerance;
- **Store**: The function store is a repository that holds pre-built function images, allowing users to easily find and deploy functions without needing to build them from scratch. This feature promotes reusability and speeds up function deployment;
- **Scaling and Load Balancing**: OpenFaaS uses Prometheus [27] metrics alerts to make scaling decisions and manage the number of running function instances based on the current demand. The platform automatically scales the functions up or down to ensure optimal performance and resource utilization;
- **Metrics and Monitoring**: OpenFaaS integrates with Prometheus and Grafana [26] to collect detailed metrics on function execution, such as invocation count, execution time, and error rates.

OpenFaaS and OpenWhisk are both open-source platforms designed to facilitate and improve serverless computing's potential, but they have differences in their architecture, deployment, and ecosystem. OpenFaaS is known for its simplicity and ease of use, leveraging Docker and Kubernetes to deploy and manage functions. It focuses on providing a better developer experience with an intuitive UI, CLI, and integration with a wide range of programming languages. On the other hand, OpenWhisk offers a more robust and flexible environment. It also supports a variety of programming languages and allows complex workflows through its rule-based event-driven architecture. While OpenFaaS prioritizes ease of deployment and developer productivity, OpenWhisk provides a comprehensive platform for building large-scale and flexible event-driven applications with a higher degree of customization.

Table 1 summarizes the generic serverless component names mapped to the specific names in each of the 2 solutions mentioned above.

Table 1: Component name mapping of OpenWhisk and OpenFaaS

	Controller	Workers	Storage	Monitoring
OpenWhisk	Controller	(w/ Invoker)	CouchDB	CouchDB
OpenFaaS	API Gateway	(w/ Watchdog)	Store	AlertManager and Prometheus

Summary

In this section we talked about how serverless computing represents a transformative paradigm in cloud computing, enabling developers to focus more on business logic while abstracting away infrastructure management. Through the Function-as-a-Service (FaaS) model, pioneered by platforms like AWS Lambda and subsequently adopted by major cloud providers, serverless computing offers significant advantages including operational simplicity, automatic scalability, and a pay-per-use pricing model. Despite challenges such as cold start latencies, resource management complexities, and potential performance limitations, serverless architectures have found widespread applicability in event-driven applications, microservices, IoT processing, and web services. We covered commercial cloud platforms like AWS Lambda, Google Cloud Run Functions, and Azure Functions, as well as open-source alternatives such as Apache OpenWhisk and OpenFaaS. As the technology continues to mature, ongoing research focuses on optimizing serverless efficiency and performance, expanding the applicability of serverless computing.

2.2 Workflows and Data Processing Pipelines

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 especially useful for scientific experiments, in which case they are called *scientific workflows*. As defined by

Deelman et al. [28], scientific workflows are structured compositions of computational tasks that transform data through a sequence of interconnected processing steps. These workflows can be created and used by scientists to analyze, manipulate, and derive insights from complex datasets across various research domains. At their conceptual core, most workflows can be represented as directed acyclic graphs (DAGs), which visually model computational processes by depicting tasks as nodes and their dependencies as edges.

This section explores how workflows can be represented and the computational infrastructure required to run them, which has been transitioning from *traditional high-performance computing (HPC) clusters* to more dynamic and scalable environments. We will reflect on how *cloud computing* platform solutions can be leveraged to help set up and run workflows more easily, and with greater performance and cost-effectiveness.

2.2.1 Workflow Representation

Workflows are typically represented as directed acyclic graphs (DAGs), where nodes denote computational tasks and edges depict data and compute dependencies between these tasks. Each task processes input data from preceding tasks and generates output data, which can trigger dependent/downstream tasks to execute. This structure enables the representation of diverse workflows across domains, such as web applications with complex interactions, large-scale simulations for earthquake studies (Graves et al., [29], climate simulation models, genome analysis workflows, and astronomical data processing pipelines (Deelman et al., [28]). This demonstrates the applicability and flexibility of DAG-based workflow representations in capturing complex computational processes.

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.

Several well-known scientific workflows utilize specific patterns that benefit from the DAG model. For instance, *Monte Carlo simulations*, which are often used in fields such as physics, finance, and risk analysis, rely on repetitive random sampling to estimate complex quantities. These simulations can be effectively modeled as DAGs where each task involves generating a random sample and processing it to compute an approximation, with dependencies on previous computations. Another example is *Montage* [30], a tool used in astronomy to create large-scale mosaics of the sky. The workflow involves combining image data from multiple telescopes, where tasks such as image transformation, alignment, and stitching follow a well-defined sequence of dependencies. In *machine learning* workflows, tasks such as data preprocessing, model training, and evaluation often follow a clear sequence that can be represented as a DAG, enabling efficient parallelization and scaling on large datasets. Similarly, complex workflows like *CyberShake* [29], used to model earthquake ground shaking, involve multiple stages of computation and data processing that are naturally represented using DAGs. These examples highlight the variety of scientific fields where DAG-based workflow representations are essential for representing and managing the complexity and interdependencies of computational tasks.

Several systems and libraries also leverage directed acyclic graphs (DAGs) for modelling and executing workflows. For instance, Apache Airflow [31] uses DAGs to define and schedule workflows defined in Python. Similarly, Dask [32], 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) [33] is a way HTCondor [34] (distributed computing job manager) users can organize independent jobs into workflows in the form of DAGs.

Despite the DAG format being the most used way to represent workflows, there are more flexible alternatives. YAWL (Yet Another Workflow Language) [35] 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 nets [36] 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 Workflow Resource Management, Scheduling and Deployment

Resource Provisioning The execution of workflows is a multi-stage process that requires planning, infrastructure management, and continuous monitoring. After creating the individual computational steps/tasks and composing them into a workflow, the next step is infrastructure provisioning, which means launching and configuring computational resources. These may include virtual machines, distributed computing clusters or even cloud resources. The infrastructure must be tailored to the specific computational demands of the scientific workflow, considering factors such as computational complexity, data volume, processing requirements, and potential scalability needs. However, accurately predicting the exact resources needed can be challenging; *over-provisioning* leads to wasted resources and increased costs, while *under-provisioning* can result in slow performance or failures. Serverless computing offers an attractive solution to this phase by automatically scaling resources to meet the demands of the workflow.

Uploading Code and Dependencies Code and dependency management represents another crucial phase in workflow execution. Users upload their workflow code, along with all necessary dependencies, to the chosen execution environment. This process requires attention to version compatibility, library management, and ensuring that the computational environment matches the requirements of the workflow. **Data placement** is equally critical, with users ensuring that workflow inputs are correctly *placed* and *accessible*. Serverless platforms can simplify this phase by automatically handling dependencies and ensuring that the execution environment is consistent with the workflow requirements, reducing the overhead of manually managing libraries and their versions, security patches, and more.

Workflow Scheduling In order to improve *resource efficiency* and *makespan* of workflows running on distributed computing, lots of algorithms have emerged for both Grid computing and Cloud computing (IaaS model). From the Cloud workflow scheduling category, some solutions use the concept of budget to make resource allocation decisions (Yao et al. [37]), others focus on meeting user-defined deadlines (Arabnejad et al. [38]) by calculating critical path, while some aim for *Multi-objective Scheduling*, trying to deliver lower costs while also meeting time-constraints (Hosseinzadeh et al. [39]).

These algorithms were designed for a computing model where the scheduler is *centralized* and has great control over workers, being able to assign tasks to specific workers, monitor their resource usage, and directly communicate with them. In the serverless model, however, these assumptions don't hold due to the ephemeral and stateless nature of this computing model, as well as it's current limitations.

Workflow Monitoring During workflow execution, continuous monitoring can be crucial for some workflows. This involves comprehensive tracking of workflow progress, performance, errors, and system health. This can help in identifying computational bottlenecks, diagnose and resolve issues *on the fly*, and even optimize resource utilization. Serverless architectures can enhance monitoring by providing built-in metrics and logging capabilities, offering granular insights into performance, as well as automating alerts for anomalies.

Resource Deprovisioning After completing the workflow, resources must be deprovisioned. This stage involves shutting down computational instances, archiving results, and deleting intermediate data. Serverless computing can also aid in scaling-down compute, as it automatically deprovisions resources when they are no longer needed.

2.2.3 Traditional Workflow/Pipeline Frameworks

To alleviate some of the researchers' pain points during these steps, several data processing and workflow scheduling frameworks have emerged. Among the traditional platforms for **data processing pipelines** are Apache Spark [40], Apache Flink [41], and Apache Hadoop [42], 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 [43] and HTCondor [34], 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. We will first write about **data processing pipelines**.

Apache Hadoop Apache Hadoop is a platform for large-scale data processing, enabling researchers to execute *MapReduce* [44] jobs. It processes vast amounts of data by distributing workloads across clusters of computers in cloud environments or high-performance computing centers. Its core components are Hadoop Distributed File System (HDFS) [45] and YARN [46], providing reliable data storage and dynamic resource allocation. Hadoop's scalable architecture and fault-tolerant design make it a powerful tool for processing large datasets efficiently.

Apache Spark Apache Spark's distributed computing model supports multiple programming languages (Scala, Python, Java, R) and offers rich libraries for machine learning, graph processing, and streaming analytics. Spark simplifies workflow execution by providing efficient data processing, dynamic resource allocation, and sophisticated task dependency management.

Apache Flink Apache Flink is a powerful stream processing framework particularly well-suited for workflows that require real-time data processing. Flink can handle both streaming and batch data with low latency, making it ideal for researchers working with continuous data streams from scientific instruments or complex sensor networks. Its distributed computing architecture allows seamless scalability across cloud and high-performance computing environments, while providing robust fault-tolerance mechanisms. It also supports multiple programming languages and offers advanced windowing and state management capabilities, supporting sophisticated computational pipelines.

Next, we will reference two systems designed to **schedule and coordinate** the execution of complex sequences of tasks: *Apache Oozie* and *HTCondor*.

Apache Oozie Apache Oozie is a workflow scheduler designed to manage and coordinate the execution of data processing tasks in Hadoop-based environments, therefore benefiting from its underlying distributed file system (HDFS). It allows researchers to define, schedule, and monitor workflows that involve a series of jobs, such as MapReduce, Hive, Pig, or other Hadoop-related tasks. Oozie also supports the integration of both batch and real-time processing, making it suitable for a wide variety of workflows.

HTCondor HTCondor provides a powerful framework for executing scientific workflows across distributed computing environments, specializing in high-throughput computing and computational task scheduling. Designed specifically for academic and research settings, HTCondor excels at managing complex workflows by distributing computational tasks across heterogeneous pools of computational resources. Its scheduling supports intricate workflow dependencies, task prioritization, and efficient resource utilization, from desktop computers to large-scale computing clusters. HTCondor's unique ability to opportunistically use idle computing resources makes it particularly attractive for research institutions, enabling complex scientific computations to be executed cost-effectively while providing robust mechanisms for job management, checkpointing, and fault tolerance.

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. One major challenge is the complexity of *managing and provisioning* virtual machines, storage, and network resources, which requires a deep understanding of infrastructure management and incurs substantial overhead in terms of time and expertise. Additionally, IaaS users must *manually handle scaling* (which is usually slow), load balancing, and fault tolerance, which can lead to inefficiencies and increased costs. As we've stated before, accurately predicting resource requirements for a given workflow is challenging, often resulting in either over-provisioning or under-provisioning. Furthermore, *IaaS* cloud providers typically charge users on an *hourly basis* for the resources allocated, regardless of whether those resources are fully utilized or not. This billing model can lead to higher costs, particularly penalizing shorter workflows. For example, a workflow that only runs for a few minutes, will be billed for an entire hour.

As highlighted previously, the *serverless* paradigm excels in scenarios where automatic scaling and cost-efficiency are essential, while also providing an easy set-up process for developers by abstracting away the underlying infrastructure. 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.

2.2.4 Cloud-Native Workflow/Pipeline Frameworks

Cloud platforms have developed various offerings to help users run their data processing pipelines and create workflows composed of different programs that interact with each other (e.g., microservices). These tools are designed to facilitate the orchestration of tasks, manage state, and handle large-scale data processing, enabling organizations to take advantage of Cloud technologies. Broadly, these offerings can also (similar to the previous section) be separated into two categories: **data processing pipelines** and **workflow schedulers** (in the form of *stateful functions*). Some cloud providers offer services that correspond to *managed* instances of the traditional frameworks mentioned in 2.2.3, such as Apache Spark and Apache Flink. These managed services simplify the deployment and management of large-scale data processing tasks, allowing users to focus on developing and running their applications without worrying about the underlying infrastructure.

Apache Beam (Data Processing Model) To further enhance the portability and flexibility of data processing pipelines, Apache Beam [47] was created. Apache Beam provides a unified programming model designed to define and execute both batch and stream processing workflows. Unlike traditional frameworks that may be optimized for either batch or stream processing, Beam allows developers to write their pipelines once and run them on multiple execution engines, such as Google Cloud Dataflow [48], Apache Flink, and Apache Spark. This portability makes Beam a powerful tool for modern data engineering, enabling seamless transitions between different environments and simplifying the management of complex data processing tasks.

Apache Airflow (Workflow Scheduler) On the **workflow scheduling** side, Apache Airflow [31] is an open-source versatile workflow orchestration tool that allows users to define, schedule, and monitor complex workflows defined as DAGs in Python scripts. Unlike Apache Oozie, which is tightly integrated with the Hadoop ecosystem and primarily uses XML for workflow definitions, Airflow provides a more flexible and developer-friendly approach by allowing workflows to be written in Python, making it more adaptable to a variety of environments and use cases.

Airflow integrates with multiple services, including cloud-based services from various providers such as AWS, Google Cloud Platform, and Azure. This capability enables users to create workflows that can span across different cloud environments, orchestrating tasks that interact with diverse external systems. For example, a workflow could involve extracting data from an AWS S3 bucket, processing it with a machine learning model hosted on GCP, and then storing the results in an Azure SQL database.

Airflow was also developed with portability in mind, as it can be deployed anywhere—from on-premises data centers to public, private, and hybrid cloud environments. This is facilitated by Airflow's modular architecture and support for various databases and executors. For example, it can use MySQL, PostgreSQL, or SQLite as a metadata database, and its components can run on top of Kubernetes [49]. This flexibility allows Airflow to scale from single-node setups to distributed environments seamlessly.

Google Cloud Composer [50] is a managed service for Apache Airflow, providing a fully managed, scalable, and enterprise-ready orchestration service. It abstracts away the complexities of managing Airflow infrastructure, allowing users to focus on developing their workflows.

While Apache Beam focuses on the portability and execution of data processing pipelines across different execution engines, Airflow (and by extension, Google Cloud Composer) focuses on the orchestration and scheduling of workflows. Both tools serve complementary purposes in the data engineering ecosystem. Beam ensures that the data processing logic can run efficiently on various platforms, while Airflow coordinates the execution of these processing tasks, integrating them with other necessary operations to form a cohesive workflow.

Stateful Functions While tools like Apache Airflow are powerful for orchestrating complex workflows whose tasks span across multiple systems and cloud providers, they can be overkill for developers which already have stateless functions deployed in a single cloud provider and need more complex coordination (e.g., *fan-out*, *fan-in*) among them. Building on the strengths of serverless, the emergence of *stateful functions* further extends the serverless applicability to a wider range of scientific workflows. Such products include AWS Step Functions [5], Azure Durable Functions [6] and Google Cloud Workflows [7] and they allow the composition of stateless functions to create complex workflows that run on FaaS and automatically manage task state and coordination.

A stateful function represents a serverless **workflow orchestration** paradigm that manages the execution and state of multiple stateless functions. It is a coordination mechanism that tracks the progress, preserves context, and manages the interactions between discrete, stateless function calls. By employing techniques like event sourcing, persistent queues, and transactional state management, these orchestrators enable long-running, complex workflows that can pause, resume, and dynamically branch based on intermediate computational results. The underlying architecture typically implements lightweight state machines that track function transitions and reconstruct the entire workflow's state from a sequence of deterministic events, providing robust fault tolerance and execution traceability. Essentially, "stateful function" is more about the orchestration layer that provides state management for stateless functions. We will now briefly highlight the core features of 3 of the most relevant *stateful functions* commercial offerings.

AWS Step Functions is a powerful orchestration service that enables stateful workflows by managing the state and execution history of tasks across distributed systems. It orchestrates tasks interacting with AWS Lambda, ECS, and other AWS services. Unlike stateless functions like AWS Lambda, which have a maximum execution time of 15 minutes, AWS Step Functions can support workflows that last for up to one year, making it ideal for long-running processes. Each task within a workflow has its state saved, enabling advanced error handling, retries, and conditional branching. Workflows in Step Functions are designed using Amazon States Language (ASL), a JSON-based language that defines the sequence of steps, conditions, retries, and error handling within

the workflow. Step Functions offers a visual workflow designer, allowing users to create and visualize workflows in a drag-and-drop interface. This visual representation simplifies workflow management and debugging, providing a clear view of task progress and transitions in real-time.

Google Cloud Workflows is a fully managed orchestration service designed for creating and managing workflows that span across various Google Cloud and third-party services. Unlike stateless functions (e.g., Google Cloud Run Functions [2]), Cloud Workflows is tailored for more complex use cases, allowing workflows to run for longer durations (up to 60 minutes per execution). It enables users to define multi-step workflows using YAML or JSON syntax, facilitating integration with other Google Cloud services like BigQuery and Cloud Run. Cloud Workflows provides built-in error handling, retries, and state persistence. Cloud Workflows does not have a native visual designer; instead, it provides a detailed execution history and logs via the Google Cloud Console, which allows users to track and visualize workflow progress, errors, and retries.

Azure Durable Functions extends the serverless model to support stateful workflows with built-in state management and persistence. In terms of workflow durations, Azure Durable Functions can run for up to 30 days. Users define their "orchestrator function" as code (C#, JavaScript, Python, Powershell or Java) that controls the execution flow, maintaining the state of the workflow without the need for explicit state management (e.g., queues, databases) by the developer. Azure Durable Functions supports patterns such as function chaining, and fan-out/fan-in, enabling complex workflows like approval processes or batch data processing. Similarly to the other platforms mentioned, it integrates seamlessly with other Azure services, making it ideal for orchestrating workflows within the Azure ecosystem.

Besides opening the serverless landscape to more types of workflows, stateful functions also eliminate the burden of workflow orchestration from developers. Before stateful functions were introduced, workflows made up of multiple stateless functions would require the developer to create its own orchestration solution in a serverful fashion (a Virtual Machine that decides when to invoke the next stateless function and which data to provide it). Stateful functions are meant to be used in scenarios where workflows need to retain context across multiple invocations, require complex and multi-step processes, or require more complex coordination patterns (fan-in's and fan-out's). Stateless functions like AWS Lambda or Google Cloud Functions are ideal for short-lived, independent tasks but are limited by their execution time (e.g., 15 minutes for AWS Lambda) and lack of state persistence. This makes them less suitable for workflows that require more complex coordination (fan-in's and fan-out's), long-running executions, or dynamic branching based on intermediate results. Stateful orchestrators like AWS Step Functions, Azure Durable Functions, and Google Cloud Workflows are more appropriate for workflows that require extended durations (up to one year in AWS or 30 days in Azure), where tasks need to be paused and resumed. Workflows like approval processes, data pipelines, machine learning model training, and financial transaction processing can greatly benefit from stateful functions. These types of workflows would be challenging or inefficient to implement with stateless functions due to their need to handle intermediate states, long-running processes, and complex error handling.

A recent paper proposes SeBS-Flow (Schmid et al. [51]), a benchmarking suite designed to evaluate the performance of serverless workflows across different platforms (AWS Step Functions, Azure Durable Functions, and Google Cloud Workflows). It tries to address the challenge of inconsistent performance evaluations due to the variations in programming models and infrastructure of existing serverless workflow platforms by providing a platform-agnostic workflow model.

While commercial stateful functions provide immense value in terms of flexibility, complexity, and scalability, they can introduce a level of vendor lock-in. AWS Step Functions, Azure Durable Functions, and Google Cloud Workflows each have their own workflow representation format and are tightly integrated with their respective cloud platforms, which can make migration or Hybrid Cloud [52] strategies more challenging. An alternative to these commercial stateful orchestrators is using open-source workflow schedulers like Apache Airflow. Airflow stands out because it uses a Directed Acyclic Graph (DAG) representation for workflows written in Python, a format and programming language that is both well-known and widely adopted across many industries. The DAG model provides a universal and flexible way of defining workflows, making it easier to move workflows between different platforms without having to rewrite the entire orchestration logic.

2.2.5 Open-source FaaS runtime extensions

Although stateful functions enable users to orchestrate interactions between stateless functions and cloud services, they were not originally designed with performance or data efficiency in mind. As previously discussed, commercial serverless platforms have several limitations, making certain applications extremely inefficient or even unfeasible. The following solutions extend the FaaS runtime to improve **data locality**, with the key insight that most applications involve *related calls*. This often means functions access the *same data*, or two executions are part of a pipeline where *one function's output is only relevant as the input to the next*. These solutions argue that FaaS runtimes could be significantly enhanced, even for stateless applications that, while not explicitly sharing data

between invocations, frequently access the same data within short time frames.

Palette Load Balancing Palette Load Balancing (Abdi et al. [8]) 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 users 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*. This allows 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.

Palette's approach maintains the simplicity and flexibility of serverless computing by treating colors as *hints* rather than hard constraints. The platform is free to ignore the hints if resource management considerations require it. This approach offers several advantages:

- **Improved Cache Hit Ratios:** Palette enables effective local caching by routing related requests to the same instance, leading to *higher hit ratio* of local caches. For example, in a social networking application, requests related to the same post or user can be assigned the same color, leading to a higher cache hit rate for the post data;
- **Reduced Data Transfer Costs:** For data-intensive applications, Palette can minimize expensive data transfers between function instances;
- **Flexibility in Coloring Policies:** Palette allows users to implement different coloring policies based on the application's needs. This opens the possibility for *serverless workflow schedulers to achieve greater results* by influencing co-location of function invocations. For instance, schedulers can choose to color tasks within a workflow based on their data dependencies, or they can use a more dynamic approach that takes into account factors such as data size and task execution time to *implement custom policies*.

Palette's proposal offers a practical and efficient way to improve data locality in FaaS runtimes. By leveraging the concept of colors as locality hints, Palette bridges the gap between the stateless nature of serverless functions and the need for data locality in many applications.

Faa$T Caching Faa$T (Function-as-a-Service Transparent Auto-scaling Cache) (Romero et al. [9]) is a serverless **caching layer** where each application has its own dedicated in-memory cache, called a *"cachelet"*. This *cachelet* stores data accessed during function executions, making it readily available for subsequent related calls, and thus, minimizing the need for expensive data transfers from remote storage.

Faa$T operates *transparently*, requiring *no modifications to the application code*. It integrates seamlessly with the FaaS runtime, *intercepting data requests* and checking the cache before accessing remote storage. This transparency makes Faa$T *easy to use and deploy*, preserving the simplicity of the serverless paradigm.

The *cachelets* collaborate to form a *cooperative distributed cache*, enabling efficient data sharing and reuse across multiple application instances. Using *consistent hashing*, they identify cached object "owners" and communicate *directly* to exchange cached data. These *cachelets* scale dynamically with the application's infrastructure, meaning they also *scale down to zero* when the application is not accessed.

Faa$T core mechanisms are the following:

- **Pre-warming:** When an application is reloaded into memory, Faa$T predicts and pre-fetches *frequently accessed objects based on historical metadata*. With this statistical approach, the *"pre-warming"* process can greatly reduce latency;
- **Auto-Scaling:** Faa$T scales its cache size and bandwidth dynamically based on the application's *data access patterns* and *object sizes*. This auto-scaling of *compute*, *cache size*, and *bandwidth* ensures that the cache effectively adapts to changing workloads and data requirements;
- **Consistent Hashing:** Faa$T uses *consistent hashing* to distribute cached objects across multiple *cachelets*, ensuring efficient data finding and minimizing the overhead of metadata management.

While both Palette and Faa$T focus on improving data locality in serverless computing, they differ in their approaches. Palette utilizes **locality hints provided by the user**, allowing for a flexible and customizable way to express data affinity between function invocations. Faa$T takes a more **automated and transparent approach**, providing a dedicated, auto-scaling cache for each application. It does not require user intervention to specify locality. Instead, it automatically manages data placement and retrieval based on observed access patterns.

Lambdata Similarly to Palette, Lambdata (Tang et al. [10]) focuses on optimizing data locality by leveraging user-provided information. In Lambdata this information is called "data intents", where users specify *exactly* what data objects a function will *read and/or write*.

Here's a closer look at how Lambdata enhances data locality:

- **Data Intent Declarations:** Function invokers annotate their functions with *"get_ data"* and *"put_ data"* parameters, explicitly listing the data objects required for input and output, respectively. These annotations provide the *Lambdata Controller* with insights into the data access patterns of each function. For instance, a thumbnail-generating function would declare its intent to read a specific image file ("pic/1.jpg") and write the resulting thumbnail to another location ("thumb/1.jpg");
- **Local Caching:** Each *Lambdata Invoker*, responsible for executing functions, maintains a local object cache. When a function is invoked, Lambdata checks the local cache for the required data objects specified in the "get_data" intent. If the data is present locally, the function directly reads it from the cache, avoiding the latency of retrieving it from remote storage;
- **Data-Aware Scheduling:** The *Lambdata Controller* utilizes the declared data intents to make *informed scheduling decisions*. When multiple functions require access to the same data, Lambdata tries to schedule them on the same *Invoker*. This colocation allows the functions to *reuse the cached data*, minimizing data transfer overheads.

In essence, Lambdata optimizes data locality by intelligently scheduling functions and leveraging local caching within a single instance. It relies on user-provided data intent declarations to guide its decisions, promoting data reuse and reducing reliance on remote data transfers. Compared to Faa$T, which manages a distributed cache and proactively fetches data from other instances, Lambdata adopts an approach that simplifies deployment/integration and usage. While it might not offer the same level of fault tolerance or data availability as a distributed cache, it simplifies implementation and reduces the overhead associated with inter-instance communication. Similar to Palette, Lambdata seeks to reduce data transfer overhead by scheduling related functions together. However, rather than using *"color" hints*, Lambdata depends on *explicit data intent declarations*, which can restrict the flexibility of applications built on it.

Table 2 sums-up the core differences between these FaaS extensions regarding their data locality, level of user interaction/application modification required, complexity, and caching approach. We consider Faa$T to be the most transparent solution, meaning that the developer doesn't need to perform modifications to its application to benefit from Faa$T's advantages. As Lambdata requires the user to specify the exact objects a function manipulates, we believe it is easier for the developer when compared to choosing *"colors"* (Palette). We also classify Faa$T as being the most complex solution, since it's a distributed caching layer that requires changes to the scheduling made by serverless platforms.

Table 2: Comparison between different FaaS runtime extensions (Palette, Faa$T, and Lambdata)

	Palette	Faa$T	Lambdata
Locality Mechanism	colors	none	input/output objects
Transparency	+	+++	++
Solution Complexity	+	+++	++
Caching Mechanism	None	Distributed Cachelets Layer	Independent Caches on Invokers

Summary In this section, we focused on describing *how workflows and data processing pipelines are represented, managed, and executed*. We explored *traditional frameworks, schedulers, and orchestrators* that support these types of computational jobs (such as Apache Hadoop, Apache Spark, Apache Flink, HTCondor, and Oozie). Then, we wrote about *modern solutions* that were designed with ease of use and deployment in mind, and seamlessly run and integrate with cloud solutions, providing a more comfortable experience for developers and researchers (like Apache Airflow and Apache Beam). We then discussed how *stateful functions* (AWS Step Functions, Azure Durable Functions, Google Cloud Workflows) build on top of stateless functions to provide an orchestration layer and how that can be inefficient for certain types of use cases. Lastly, we explored three different *extensions to the FaaS runtime* (Palette, Faa$T, and Lambdata) which aim at solving some of the current serverless inefficiencies by enhancing data locality.

2.3 Relevant Related Systems

With the goal of efficiently running diverse workflows on the serverless, stateless model, several workflow orchestrators and schedulers have emerged. These systems are notable for their innovative approaches, design decisions,

and trade-offs, making them highly relevant in the field of serverless computing. In this section, we will discuss some of the most relevant and interesting solutions: **PyWren** (Jonas et al. [11]) leverages stateless cloud functions for executing large-scale parallel computations across distributed infrastructure; **DEWE v3** (Jiang et al. [53]), a serverless workflow management and execution system designed for complex scientific workflows, uses a hybrid model that combines serverless functions with dedicated/local clusters to overcome certain limitations inherent in serverless environments; **Dask Distributed** ([54]), allows for running Dask computations, providing dynamic task scheduling and robust support for complex, large-scale computations across both local and cloud-based IaaS infrastructures; **Unum** (Liu et al. [55]) proposes a decentralized orchestration strategy that runs on unmodified serverless infrastructure without adding services or components. It wraps user functions' code to perform local orchestration logic and coordinate with other nodes when needed; lastly, **WUKONG** (Carver et al. [12]) is another decentralized solution that implements more sophisticated scheduling strategies to optimize resource utilization and workflow performance in serverless environments.

2.3.1 PyWren: Distributed Python code on AWS Lambda

PyWren leverages the serverless computing paradigm to simplify distributed computation for a wide range of users. Built atop AWS Lambda [1], PyWren is designed to handle embarrassingly parallel workloads effectively. The core principle of PyWren is to execute arbitrary Python functions as stateless serverless functions, requiring minimal user/developer management. PyWren automatically creates and manages function execution and dependencies, an S3 bucket with serialized function code and intermediate data, and a Key-Value store instance for smaller data that is also accessed more frequently.

PyWren is particularly well-suited for workloads with many **independent**, parallel tasks, often referred to as embarrassingly parallel workloads or *bag-of-tasks*. In these types of workloads, there is a large number of independent tasks. Some examples include simple data transformations, scientific simulations, parallel model training, or large-scale media processing. Although simple, this model is very popular and lends itself to many practical scenarios, for which research has been carried out. For example, Silva et al. [56] uses heuristics to optimize IaaS resource allocation for these workloads. A more recent survey by Khan et al. [57] reviews numerous task scheduling techniques for cloud and fog computing, with the majority being suitable for handling workloads of this nature.

An interesting project build on top of PyWren is Numpywren (Shankar et al. [58]), which builds upon the principles of PyWren by extending the serverless computing paradigm to linear algebra operations. While PyWren focuses on executing arbitrary Python functions in a stateless, distributed manner on AWS Lambda, numpywren specifically targets large-scale linear algebra computations.

Limitations:

- **Large Intermediate Ojects:** PyWren's stateless nature limits direct interaction between tasks, making it unsuitable for workflows whose tasks require frequent communication or sharing of large intermediate results;
- **Latency-sensitive Requirements:** Workflows that require real-time processing, for example, may not perform well due to inherent cold start latency and the overhead associated with using external storage for intermediate data;
- **Long-running Tasks:** PyWren may not be ideal for long-running tasks that exceed the maximum execution time limit of the underlying execution environment.

2.3.2 DEWE v3: Hybrid Workflow Execution Engine

DEWE v3 is a hybrid workflow execution engine that uses serverless computing for short tasks and serverful computing for longer tasks. DEWE v3's hybrid approach can handle various resource consumption patterns throughout workflow execution, mitigating issues related to execution duration limits, memory limits, and storage space limits by retrying failed FaaS jobs on a serverful machine ("local job handler").

DEWE v3 employs a queue to manage the distribution of jobs between FaaS and the "local job handler". Jobs expected to complete within FaaS's limits are published to a *common job queue* for FaaS execution, while long-running jobs are directed to a separate queue designated for local, serverful execution. The FaaS job handler, a function deployed to the FaaS platform, picks up jobs from the common job queue and executes them within the FaaS environment. The local job handler, running on dedicated servers, retrieves jobs from the *long-running job queue* and executes them.

This hybrid approach enables DEWE v3 to provide the serverless model advantages to a wider variety of workflows, namely those that have both short and long-running jobs. DEWE v3 is particularly well suited for scientific workflows that involve many precedence-constrained tasks, such as Montage [30] and CyberShake [29].

Limitations:

- **Latency-sensitive Requirements:** The overhead associated with placing task invocation requests in queues and then invoking the serverless function, which is subject to cold starts, makes DEWE v3 unsuitable for latency-sensitive workflows;
- **Large Intermediate Objects:** Workflow performance can degrade if tasks that produce large data objects are run on FaaS, because direct function communication is not allowed;
- **Resource Underutilization:** Due to its hybrid nature, DEWE v3 can suffer from resource underutilization since the serverful environments may not be near full capacity for a good amount of time;
- **Scheduling Overhead:** Its centralized *workflow management system* and queues may introduce a performance bottleneck, as they can become points of congestion, limiting the system's scalability and efficiency when dealing with a high volume of jobs.

Compared to PyWren, DEWE v3 shares similar limitations but allows workflows with longer tasks to benefit from FaaS.

2.3.3 Dask Distributed: Scalable Parallel Computing Framework

Dask is a flexible, parallel computing library designed to scale Python programs from a single machine to large distributed clusters. Built on top of familiar Python libraries such as NumPy, pandas, and scikit-learn, Dask enables efficient parallel execution of large-scale computations without requiring major changes to existing codebases. By offering high-level APIs for parallel computing and distributed data processing, Dask allows users to scale their applications seamlessly. Beyond its utility for data scientists, Dask can also parallelize generic Python functions, making it a versatile tool for a wide range of applications.

Dask Distributed extends the core functionality of Dask by providing an advanced scheduler for parallelizing and distributing computations across a cluster of machines. It introduces robust management of worker resources, fault tolerance, and high-level control over task execution, offering flexible ways to execute parallel computations both locally and in large distributed environments. The Dask Distributed scheduler prioritizes data locality, by executing tasks where the data it depends on resides.

Dask Distributed is particularly useful for workflows involving large datasets, complex computations, or tasks that require large data transfers (due to its data locality optimizations). Dask is ideal for any computation that can benefit from parallelism and the ability to handle distributed data processing.

Limitations: Despite its scalability and flexibility, Dask Distributed may not be suitable for certain types of workloads:

- **Slower Scaling:** Because Dask Distributed relies on a serverful architecture, it can take longer to provision and scale resources compared to serverless solutions. This delay can be significant for embarrassingly parallel workloads;
- **Resource Underutilization:** In a serverful environment, resources are often pre-allocated and may remain underutilized during periods of low demand;
- **Infrastructure Setup Difficulty:** Managing a distributed cluster involves significant overhead in terms of setup, configuration, and maintenance of the infrastructure. This can include ensuring network reliability, managing security, and dealing with hardware failures, all of which require dedicated effort and expertise;
- **Complexity of Deployment:** Deploying Dask Distributed can be more complex compared to the previous alternatives, which abstract away much of the infrastructure management. This complexity can be a barrier for teams without dedicated DevOps resources or expertise in distributed computing.

By using only the IaaS model, Dask Distributed inherits its limitations. It scales slower than DEWE or PyWren, it has less resource efficiency, and it requires more complex infrastructure management. However, its scheduler has greater control over the workers (more flexibility), achieving better data locality. These characteristics make it suitable for workflows where tasks exchange large data objects, but possibly less suitable for embarrassingly parallel workflows.

2.3.4 Unum: Decentralized Application-Level Serverless Orchestrator

Unum is an application-level serverless workflow orchestration system without a standalone orchestrator, offering a decentralized approach that contrasts with traditional serverless orchestrators like AWS Step Functions [5]. It achieves this by decentralizing orchestration logic into a library that wraps around user-defined FaaS functions' code, leveraging a scalable consistent data store for coordination on fan-ins and execution correctness. Unum introduces an Intermediate Representation (IR) to represent node-local information about the next stage of the workflow.

This solution relies on a minimal set of existing serverless APIs (function invocation and basic data store operations) that are common across cloud platforms. This design choice allows Unum to avoid the need for a separate orchestration service, reducing costs and increasing flexibility and portability. Furthermore, workflow definitions from languages like AWS Step Functions and Google Cloud Workflows [7] can be compiled into Unum's IR, enabling Unum to run arbitrary workflows defined using those languages.

Limitations: Despite its portability and cost-effectiveness, Unum has a few limitations:

- **Statically Defined Control Structures**: Unum currently only supports statically defined control structures, and cannot express workflows dynamically as code where the next step is determined at runtime. For example, workflows that use arbitrary logic to determine the next workflow step during runtime may not be compatible with Unum;
- **Limited Data Locality**: Due to its generic approach, Unum does not optimize for *data locality* as it can't force two tasks to execute on the same worker. In Unum, each FaaS function instance executes only its specific task and then triggers the next function. This contrasts with WUKONGs' approach, which we will describe next, where a single executor can dynamically take on multiple tasks along a subgraph, caching intermediate results and reducing data transfer latency.

By eschewing the traditional standalone orchestrator model, Unum offers a different approach to serverless orchestration. It achieves greater flexibility and reduces costs by embedding the orchestration logic within the application functions themselves, rather than relying on an external service. This decentralized approach allows it to run on unmodified serverless infrastructure, automatically benefiting from improvements in FaaS schedulers, workers and data stores.

2.3.5 WUKONG: Decentralized Locality-Enhanced Framework for Serverless Parallel Computing

WUKONG [12] is a serverless parallel computing framework designed to address the limitations of traditional serverful computing models like Dask Distributed while also exploring the advantages of the serverless model. WUKONG's goal is to improve Lambda scale-out speed, reduce user deployment effort, and enhance data locality, minimizing large objects movement. A key part of WUKONG's scalability is its distributed scheduling approach which avoids the bottleneck introduced by traditional scheduling approaches. This framework is currently meant to run Dask computations, which can be expressed as DAGs even before running those computations. While the most recent version of WUKONG is *WUKONG 3.0*[1], here we will be writing about *WUKONG 2.0*.

WUKONG's architecture, shown in Figure 2, can be divided into two parts: **static** (before workflow executes) and **dynamic** (while workflow is executing). The **static** part comprises the following components:

- **DAG Generator (Static Scheduler):** Converts user Python code into DAGs using the Dask library;
- **Schedule Generator (Static Scheduler):** For a DAG with n leaf nodes, it generates n static schedules. A static schedule for leaf node L includes every task node that can be reached in a depth-first search starting at L. This component also helps achieve faster scaling on large fan-outs;
- **Initial Task Executor Invokers (Static Scheduler):** For each *leaf node/task*), these invokers launch a new AWS Lambda instance;
- **Subscriber Process (Static Scheduler):** At the end of the workflow, it downloads the final results from the key-value store and returns them to the user.

The **dynamic** part of WUKONG's architecture is composed of the following components:

[1]WUKONG 3.0 is fully serverless, improving usability by replacing the serverful components of WUKONG 2.0. Currently, it's slower than WUKONG 2.0. Here we focus on WUKONG 2.0 because its codebase is stable and publicly available.

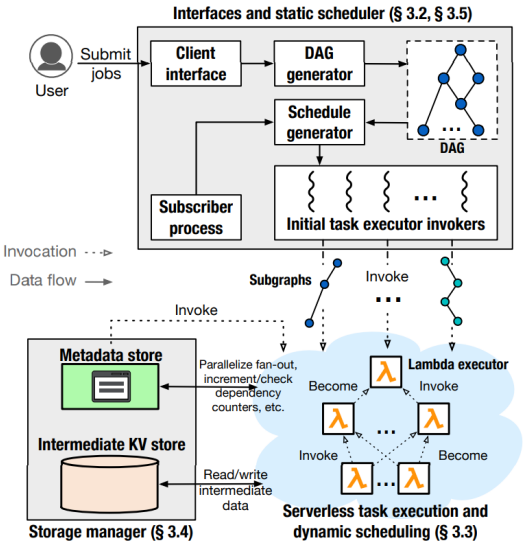

Figure 2: WUKONG 2.0 Architecture Diagram (taken from the paper [12])

- **Lambda Executors (Dynamic Scheduling):** Lambda Executors receive a static schedule, either from the **Initial Task Executor Invokers** or other Lambda executor. They execute all the tasks in their schedule (achieving great data locality) until they find a fan-out (a task that has N tasks depending on it). When they find a *fan-out* (*1* to N), they transfer their data to the *Intermediate KV Store* and invoke N new Lambda executors. In the invocation parameters, these executors receive their *schedule of tasks* (a sub-graph of the original DAG) and the *KVS Store keys* that contain their input. When an executor reaches a *fan-in* (N to *1*), meaning multiple tasks have input that a single task needs, it needs to coordinate with the other N executors to decide who will execute the fan-in task. All other executors (N - *1*) stop. The resolution of *fan-ins* happens at runtime and is called dynamic conflict resolution;
- **Metadata Store (Storage Manager):** This component is a key-value store for workflow-related metadata, such as *static schedules* and *counters* used for fan-in synchronization;
- **Intermediate KV Store (Storage Manager):** A key-value storage (KVS) for storing intermediate workflow data. When a Lambda executor needs to delegate (fan-out) or stop it uploads the ouput of the tasks it executed to this component.

Worker Scale-out Speed If at some point of the workflow, a certain task fans-out to a large number of tasks, some of those new executor invocations are delegated to the **Static Scheduler**, making these fan-outs faster. Additionally, WUKONG's decentralized scheduling approach, where each executor has its own static schedule, helps to avoid bottlenecks that can occur with a centralized scheduler. For example, in *numpywren* [58], as the number of workers increases, contention at the framework's central queue or scheduler also increases, resulting in performance degradation.

WUKONG Deployment Regarding deployment, WUKONG 2.0 is designed to be executed on AWS. It uses AWS Lambda for serverless executors, AWS EC2 virtual machines to run the Static Scheduler, Metadata Store, and Key-Value-Store Proxy components, and an AWS Fargate cluster for elastic storage of intermediate data. To simplify the deployment process, WUKONG provides automated scripts for deploying the necessary AWS infrastructure.

Data Movement Optimizations WUKONG applies 3 data-locality optimization techniques/optimizations to avoid moving large data objects around. The goal of these optimizations is that, if a task produces a large output object, the tasks that depend on that object will run on the same executor to avoid large data movements. These optimizations are the following:

- **Task Clustering for Fan-Out Operations:** If the output object of a fan-out task T is larger than a threshold (user-defined), *T's* executor E will try to also execute any of *T's* downstream tasks that are *ready* (all dependencies are complete) to execute. This means that E will *"become"* the executor for multiple fan-out task targets, instead of invoking new executors for the targets. For example, if task *C's* output is large, its executor may also execute the downstream tasks F and G if they are ready to run;

- **Task Clustering for Fan-In Operations:** Normally, if an executor E is about to execute a fan-in task T whose input dependencies have not been satisfied, E sends the intermediate object needed by T to the *Intermediate KV Store*. However, if the object to be transferred is large, after uploading it to external storage, E will recheck *T's* input dependencies to see if they have been satisfied while the object was being stored/uploaded. If *T's* dependencies have been satisfied, E will *"become" T's* executor, avoiding the time loss of a new executor of T having to download this large object from storage;
- **Delayed I/O:** If executor E executes a task with large output that fans out and finds some tasks are *not ready*, E will execute any *ready* fan-out tasks and *delay* deciding how to handle the unready tasks (doesn't immediately upload the large object to storage). E will then recheck if the unready tasks' input dependencies have been satisfied. If so, E will execute the newly ready tasks itself and recheck the unready tasks again. If, after executing all the *ready* tasks there are still unready tasks, E will upload the object to external storage. After uploading, E can re-check if the *unready* tasks became *ready* and if so execute them, saving download time.

Strengths WUKONG's evaluation shows that its optimizations combined significantly enhance performance and efficiency when compared to Dask Distributed and PyWren. By **clustering** tasks for both fan-out and fan-in operations, WUKONG can minimize the need to transfer large intermediate objects between executors, *reducing data movement* and *improving data locality*. **Delayed I/O** further optimizes the execution by allowing executors to handle ready tasks immediately while postponing the storage of large objects until absolutely necessary, thus reducing the overhead associated with frequent external data storage access.

Due to its decentralized scheduling, where executors resolve conflicts through dynamic scheduling without needing to communicate with a centralized scheduler, WUKONG is more performant and scalable while also avoiding a single point of failure.

Limitations: Similarly to Unum, WUKONG only supports statically defined control structures (Workflow DAG must be known *ahead-of-time*), therefore being unable to run workflows where the next step is determined at runtime. Moreover, WUKONG's decision-making and optimizations (Delayed I/O and Task Clustering) can lead to inefficiencies in certain scenarios. Delayed I/O may increase execution time if task dependencies aren't met after retries. In fan-ins, if two tasks produce large objects, the executor that takes longer to upload its object determines the downstream task's executor, potentially wasting time and resources. For fan-outs, invoking multiple executors for tasks with small inputs may not be efficient. We will go into more depth into these limitations in the solution in Section 3.

Compared to PyWren and DEWE v3, WUKONG's decentralized scheduling and scheduling optimizations provide better scalability and data locality. WUKONG is better than DEWE, but especially PyWren, in embarrassingly parallel workflows, or when tasks require frequent communication or large intermediate results. DEWE v3 mitigates some of these issues by adopting a hybrid approach, yet it still faces challenges with latency-sensitive requirements, resource underutilization, and scheduling overhead. As shown in WUKONG's evaluation, its decentralized scheduling and optimizations also outperform Dask Distributed for larger-scale problems where the scheduler becomes a bottleneck and workers can suffer from resource contention.

Summary Here we have highlighted the most relevant works regarding *serverless workflow orchestration and scheduling*. These include *PyWren*, *DEWE v3*, *Dask Distributed* and *WUKONG*. First, we looked at **PyWren**, which focuses on *embarrassingly parallel workloads*, enabling simple distributed computing through AWS Lambda with *minimal management overhead*. **DEWE v3** takes a *hybrid* approach, distributing jobs between *serverless and serverful* environments to handle complex scientific workflows with varying task requirements. **Dask Distributed** offers a flexible parallel computing framework that *scales Python computations across distributed systems*, prioritizing *data locality* and familiar library integrations. Lastly, **WUKONG** stands out with its *decentralized scheduling* and *data locality optimizations*, implementing techniques like task clustering and delayed I/O to minimize data transfer overhead.

Table 3 compares the five systems mentioned in this section across various aspects regarding workflow execution. PyWren, with its serverless and centralized approach, is the simplest to use but lacks data locality and scalability. DEWE v3's hybrid model and queues for job submission achieves better data locality and scalability for workflows. Dask Distributed, primarily serverful, offers excellent data locality but is harder to deploy and manage. Unum's decentralized design provides performance and efficiency benefits over state-of-the-art commercial serverless workflow orchestrators, while also providing portability across multiple providers. Lastly, WUKONG 2.0, with its decentralized scheduling, achieves the highest scalability and good data locality, while maintaining a balance in ease of use.

Table 3: Comparison between different Workflow Schedulers (PyWren, DEWE v3, Dask Distributed and WUKONG 2.0)

Aspects	PyWren	DEWE v3	Dask Distributed	Unum	WUKONG 2.0
Execution Model	Serverless	Hybrid	Serverful	Serverless	Serverless
Scheduling	Centralized	Centralized w/ Job Queues	Centralized	Decentralized	Decentralized
Data Locality	None	+	+++	+	++
Scalability	+	++	++	+++	+++
Ease of use	+++	+	+	+++	++

While each system presents unique strengths, from PyWren's simplicity to WUKONG's more sophisticated scheduling, they also share *common limitations*, such as challenges with large intermediate data, inter-task data exchanges, potential resource underutilization, and the complexities of managing distributed computing environments. These challenges are often introduced or amplified by the inherent limitations of current serverless runtimes. Nonetheless, these solutions collectively represent significant progress in demonstrating the potential of FaaS as a versatile execution environment for various types of workflows.

2.4 Analysis

In this section, we have discussed how serverless computing, represented by the Function-as-a-Service (FaaS) model, offers a transformative approach to cloud computing by abstracting infrastructure management and enabling developers to focus on business logic. Serverless architectures provide operational simplicity, automatic scalability, and a pay-per-use pricing model. Despite challenges such as cold start latencies and resource management complexities, these platforms are improving and have been widely adopted for event-driven applications, microservices, IoT processing, and web services.

Then, we explored traditional data processing pipelines and workflow scheduling solutions, which have been used for managing and running computational jobs for the past decades but often require significant setup and management effort. In contrast to these traditional solutions, we also presented modern cloud-native solutions that seamlessly integrate with cloud environments, enhancing ease of use, portability, and deployment for developers and researchers. We also highlighted innovative extensions to the FaaS runtime that aim to improve data locality and mitigate some of serverless computing limitations. These solutions enhance the performance and scalability of serverless workflows by optimizing data access patterns and reducing data transfer overheads.

Finally, we compared serverless workflow execution orchestrators and schedulers, from which we found WUKONG 2.0 to be the most interesting, innovative and promising approach for exploiting the most out of the serverless computing paradigm. WUKONG 2.0 achieves **fast scale-out times** by delegating part of the worker instancing to an external component, **scalability** with its distributed scheduling approach, and **data locality** with its optimizations that try to run related tasks on the same worker while also minimizing large data transfers.

Despite its advantages, we also pointed out some situations where we believe WUKONG's optimizations could end up degrading performance. We believe that WUKONG 2.0 scheduling decisions are optimized for workflows with *short and uniform* tasks. **Information from previous workflow runs** could help WUKONG achieve greater performance and resource usage, which would naturally allow it to run a wider number of workflows on top of FaaS.

3 Proposed Solution

As we have stated before, a new computing paradigm (serverless) demands adaptation of the scheduling techniques that targeted traditional computing models (e.g., IaaS). Throughout Section 2, we highlighted some projects with interesting and diverse architectural decisions that try to understand the best way to exploit serverless platforms advantages to run workflows. Regarding performance, current serverless computing platforms can only best the IaaS model on *embarrassingly parallel jobs with uniform and short-duration tasks*. Currently, FaaS's characteristics are not ideal for workflows where tasks require a lot of data exchange. Despite this, we believe that serverless platforms will continue to improve, eventually overcoming their current limitations, and prove to be a viable alternative to IaaS for workflows, which perfectly fit the event-driven model used in serverless platforms while also providing a more user-friendly experience. Regardless of any improvements (on compute power, data exchange, time limits, cold starts, and others) on serverless platforms in the future, we believe that our solution will still be relevant to improve the scheduling of workflows on serverless platforms.

Most of the schedulers used by the solutions explored previously use a *one-step scheduling* approach, meaning they make decisions based only on the next workflow stage. Using WUKONG as an example, on fan-outs, the upstream task executor will launch one worker for each downstream task without taking into consideration the

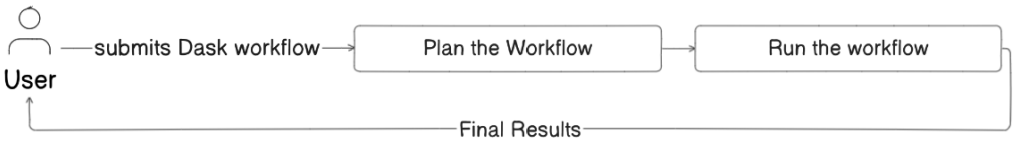

Figure 3: Use Case Diagram

implications of that: later, when the workflow fans-in, it should take longer because there are more workers active. Despite having a global view of the workflow before it even starts, WUKONG doesn't take advantage of that to improve its scheduling. We believe that workflow schedulers could greatly benefit from leveraging metadata from previous runs of the same workflow. Such metadata could include **average execution time** and **input/output ratio** *for each task*. Lambdata [10] is an example of a research project that improves the accuracy of its scheduling decisions by predicting data transfer, cold start and warm start times. By collecting this metadata and associating it to its runtime configuration (e.g., worker CPU/RAM), we could extrapolate to make predictions about execution times, data transfer times, and output sizes of each task. This metadata could then be leveraged to implement "smarter" algorithms that are aware of the entire workflow and know what to expect from each task.

3.1 Architecture Overview

Our solution proposal aims to collect and use historical data from previous runs of the same task to help executors make better scheduling decisions. Its architecture features **3 high-level components** that can be integrated into, but not limited to, serverless schedulers.

The first component is the **Metadata Management**, which will store task metadata. Such metadata will include *execution time*, *worker configuration*, *input size* and *output size* for tasks, while also recording all *data transfer times*. The next component is the **Static Workflow Planner**, which receives the entire *workflow description*, as a DAG, and *an algorithm*. This algorithm will receive the DAG and access to the metadata collected by the *Metadata Management*. It should use this information to output the same DAG, but with *annotations on each task*, where each annotation indicates in *which worker instance* a task should run. Equal worker IDs mean the same worker, similar to *"color"* in Palette [8]. The last component proposed is the **Scheduler**, to be implemented at the worker level (decentralized), and will use the *annotations* to drive scheduling and perform certain *optimizations* (e.g., pre-warm workers, download/upload data). By knowing where each task will execute, these optimizations can improve the resource efficiency and reduce workflow *makespan*.

In Figure 3, we show how a user would interact with the system. It starts by using a Python library to upload its Dask workflow, which is then transformed by the **Static Workflow Planner**, using an algorithm selected by the user to annotate the workflow. The workflow is then executed, following the static plan and, once over, the results are returned to the user.

3.2 Distributed Architecture

As the diagram in Figure 4 illustrates, there are 4 distinct computational entities involved in our solution, being:

- **User Computer:** Workflows are submitted from here and their results are sent here too;
- **Virtual Machine:** This will contain the *Static Workflow Planner* component, which is responsible for adding annotations to the workflow tasks, indicating where they will run and optimizations;
- **Workflow Runners:** These are the workers, which will execute one or more tasks and drive the workflow scheduling. Their runtime is FaaS containers, and they upload task output and metadata to external storage;
- **Elastic Storage:** Composed of Intermediate Storage (store intermediate task data) and *Metadata Store* (store metadata relevant for workflow planning).

Our solution will be built on top of WUKONG 2.0, by modifying and extending it with the components mentioned previously. These modifications and extensions are illustrated in Figure 5. Our **Static Workflow Planner** component should receive the DAG produced by the *DAG Generator* component of WUKONG, outputting an *annotated DAG*. WUKONG's Static Schedule Generator will then partition the DAG into sub-graphs to be delivered to the initial workers. We will have to modify this component to use the information produced by our *Static Workflow Planner*.

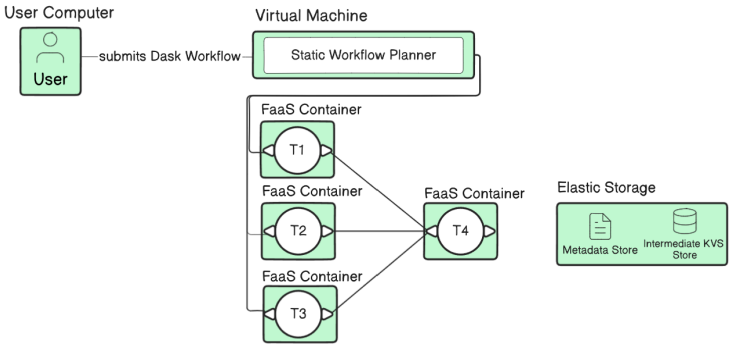

Figure 4: Solution's Distributed Architecture

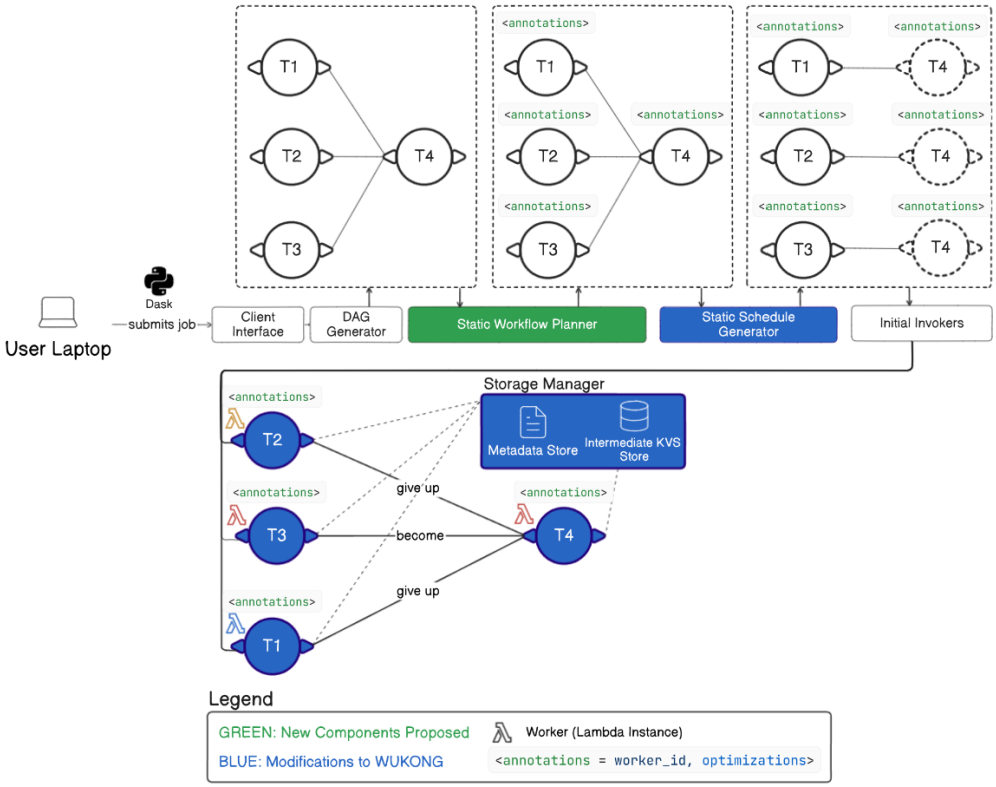

Figure 5: Solution's Distributed Architecture Integrated in WUKONG 2.0

The WUKONG workers will also require modification to discard WUKONG's optimizations and incorporate our optimizations, which are *aware of task placement*. Lastly, the structure of the data stored at WUKONG's *Metadata Store*, inside the *Storage Manager*, should also be modified.

We will now go deeper on the 3 solution components: **Metadata Management**, **Static Workflow Planner**, and the **Scheduler**.

3.2.1 Metadata Management

The *Metadata Management* will be responsible for providing task-wise predictions about *execution time* and *input/output ratio*, as well as data transfer times. These predictions will be made using data collected from previous workflow runs, which is also a responsibility of this component. The Metadata Management stores *historical information* in WUKONG's *Metadata Store*. The proposed structure for task-related metadata is shown in Listing 6, and for data transfers in Listing 7. Before a worker stops, it uploads this information for each task it executed, and for each data transfer it performed.

```
{
    task_id: string,
    worker_configuration: { v_cpu: number, ram: number },
    execution_time: number,
    input_size: number,
    output_size: number
}
```

Figure 6: Task Metadata Structure, to be stored in *Metadata Store*

```
{
    data_size: number,
    transfer_time: number,
}
```

Figure 7: Data Transfer Metadata Structure, to be stored in *Metadata Store*

The metrics stored in the *Metadata Store* will then be exposed through an API (pseudocode in Listing 8), to be used by the **Static Workflow Planner**'s algorithm to produce an *annotated DAG* (Listing 9). This API will allow the algorithm to: predict the **time to download data** of a given size from a remote worker; predict the **output of a given task**, provided an input size; and predict the **execution time of a task** on a range of worker configurations (provided by the algorithm), given an input size.

Each of these functions will also receive an `sla` (service-level agreement) argument, allowing the algorithm to specify the desired statistical metric to guide the prediction. This can give algorithms greater control over how well the predictions align with specified requirements.

```
interface MetadataAccess {
    predict_remote_download_time(data_size: number, sla?: median | percentile_value | avg): number;
    predict_output_size(task_id: string, input_size: number, sla?: median | percentile_value | avg):
        number;
    predict_execution_time(task_id: string, input_size: number, worker_configs: List<{ worker_config: {
         v_cpu: number, ram: number } }>, sla?: median | percentile_value | avg): List<{ worker_config:
         { v_cpu: number, ram: number }, time: number }>;
}
```

Figure 8: MetadataAccess API interface

There is research that tries to make these predictions as accurate as possible. One example of a recent FaaS workflow orchestrator that uses a performance model to capture the execution characteristics and variability of serverless environments is Jolteon (Zhang et al. [59]). Jolteon's performance model learns and updates over time by capturing execution characteristics, data transmission times, computational requirements, and the impact of resource allocation. It then uses this model to achieve application-level requirements like latency or cost by deciding on which *resource configuration* each invocation should run. Jotleon seems more suitable for service providers who want to give the best performance possible to a user with high confidence that the budget won't be exceeded.

Predictions in our solution are more of an insight into previous executions, rather than providing strong guarantees about how each function invocation will behave given a certain resource configuration or input size.

3.2.2 Static Workflow Planner

This component is a software module that will receive the **workflow DAG**, and a user-provided **algorithm**. It will then run *the algorithm*, providing it access to the *workflow DAG* as well as the *MetadataAccess API*. As a result, the algorithm will output an *Annotated DAG*, where the structure of each task will be extended with the worker instance ID and optimizations, as shown and explained in Listing 9.

```
{
    task_id: string,
    # worker that will execute this task
    worker_id: number,
    optimizations: {
        # indicates if the worker of this task can start download data dependencies for future tasks
            while executing this task
        pre-load: bool,
        # the workers should the worker of this task pre-warm if appropriate (at runtime) (only makes
            sense for algorithms that use non-uniform workers)
        pre-warm: List<worker_ids>,
        # indicates if this task be executed more than once if needed
        task-dup: bool,
    }
    // ... other task properties
}
```

Figure 9: AnnotatedTask structure

The idea is that the user can choose from a range of algorithms, or create its own. Different algorithms can implement different policies by planning in which worker each task will execute and what optimizations to perform at each step of the workflow. The proposed *algorithm interface* is presented in Listing 10.

```
interface PlanningAlgorithm {
    plan(dag: OriginalDAG, metadataAccess: MetadataAccess, sla: median | percentile_value | avg): Tuple
        <AnnotatedDAG, WorkerConfigurationMapping>
}
```

Figure 10: Planning Algorithm interface

The algorithm should also output a `WorkerConfigurationMapping` (Listing 11), which maps *Worker IDs* to *Worker Configurations* (vCPU and RAM).

```
Map<
    worker_id: number,
    worker_configuration: { v_cpu: number, ram: number }
>
```

Figure 11: WorkerConfigurationMapping structure

Our solution implementation should include *two algorithms*. The **first algorithm** would target *uniform Lambda workers*. It would use the *MetadataAccess API* to predict the longest workflow path (critical path). Then it would simulate using the *pre-load* optimization on this path. If optimizing the critical path made it shorter than the second longest path, the algorithm would repeat the process for this new critical path. This would be repeated until we can't optimize the critical path any further.

The **second algorithm** should use non-uniform workers. It would first *find the critical* path by predicting times of all workflow tasks running on the same/best worker configuration. Then, it would downgrade resources on the other paths as much as possible without introducing a new critical path. After attributing tasks to workers (at plan-time, not run-time), this algorithm would then simulate using optimizations to further improve resource efficiency and reduce *makespan*.

Since the first algorithm is similar to WUKONG 2.0, it should allow us to assess the effectiveness of using historical workflow data. The second algorithm, by using different workflow configurations, should perform better than the first, and the challenge here is to achieve better resource efficiency, avoiding/masking data transfer latency and minimizing large data transfers. However, as mentioned before, new algorithms can be added by implementing the interface in Listing 10.

To improve the scheduling accuracy, the algorithms could also define **re-planning** points. Essentially, this would mean that the initial planning could be adjusted while the workflow runs. The benefit of this is that these plans would be based on the most updated metadata, resulting in greater planning precision. However, this optimization could result in overhead if not done properly or at the right time.

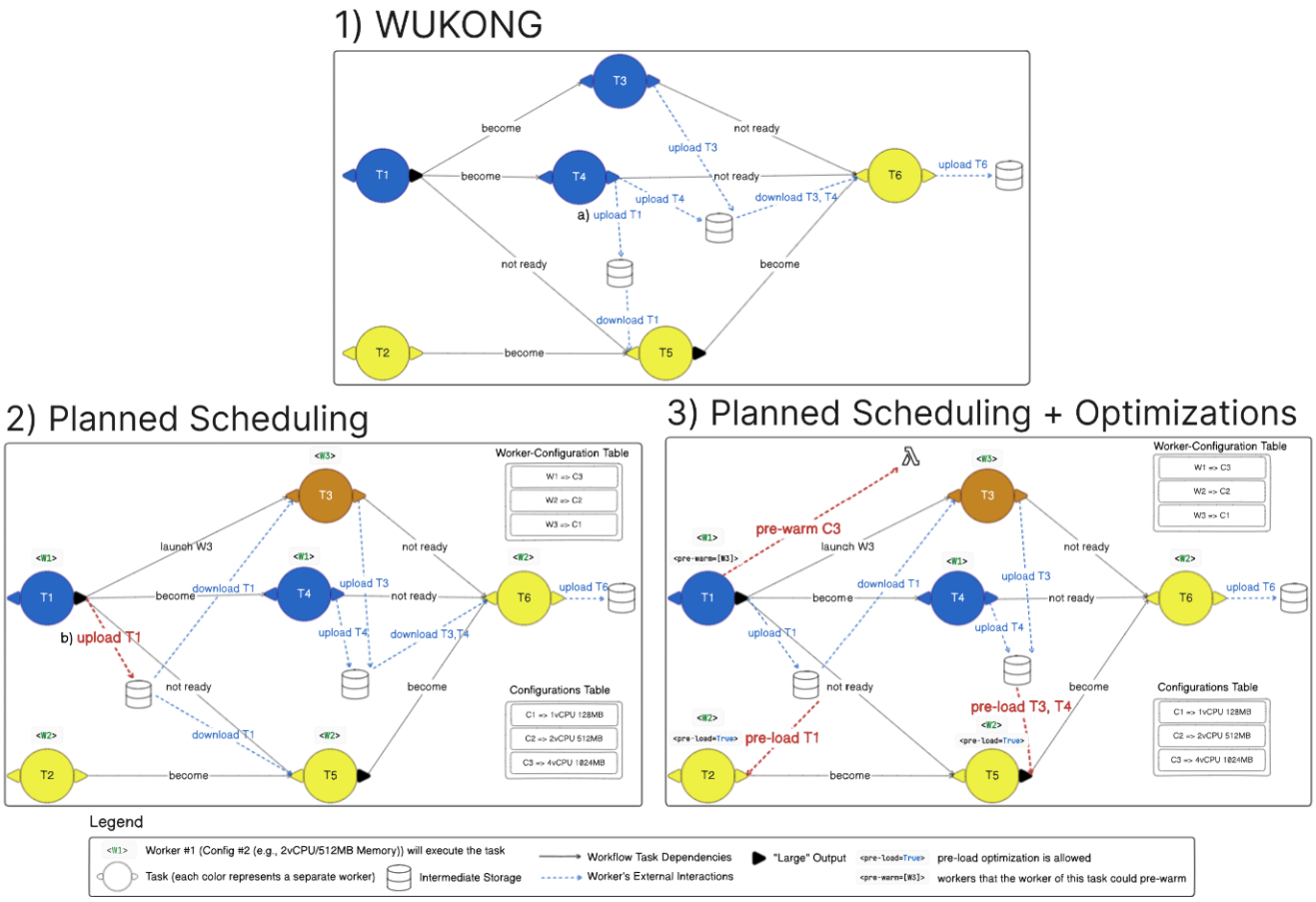

Figure 12: Workflow instance examples on different approaches

3.2.3 Scheduler

This software component, to be integrated in the workers, will be responsible for following the plan made by the **Static Workflow Planner** algorithm. In addition to deciding what tasks the local worker should execute and when to launch new workers, the *Scheduler* can take advantage of knowing where tasks will execute to apply optimizations. Such optimizations could be controlled by the algorithm, and some examples include:

- **pre-warm (Empty invocations to workers):** By pre-warming FaaS workers, we can *reduce cold starts*, achieving shorter workflow *makespan*;
- **pre-load (Downloading data dependencies ahead of time):** As the output of upstream tasks is uploaded to remote storage, the worker designated to execute a given downstream task can start downloading this data at the same time it is executing other tasks;
- **task-dup (Task duplication):** If a *Worker A* is waiting for the data of an upstream *Task 1* (executing or to be executed on *Worker 2*) to be available, it can execute that task itself. By executing *Task 1* locally, *Worker 2* won't need to wait for the data to be available and then download it from external storage. The results produced by *Worker 2* will be ignored by *Worker 1*. This could reduce *makespan* and save data download time.

These optimizations can be visualized in Figure 12, which contains 3 diagrams. All diagrams show the same workflow with different possible scheduling decisions and external storage interactions, depending on the solution.

Diagram 1 depicts WUKONG, with its three data locality optimizations. Since *Task 1*'s output is considered "large", its worker will try to execute all downstream tasks (T3, T4, and T5) locally to avoid large data transfers. Assuming *Task 2* finishes after *Task 1*, the worker of *Task 1* will execute *Tasks 3 and 4* and then check if *Task 5* became ready in the meantime. Assuming *Task 2* didn't finish at this point, the worker of *Task 1* will start uploading its output to external storage. Only after this upload, and potentially a download by another worker, will *Task 5* run. This could be inefficient if *Task 2* then finishes, but the worker of *Task 5* still has to wait for the *Task 1's* output, which was delayed unnecessarily.

Diagram 2 illustrates how the same workflow would be executed by simply *following a plan*, made by an arbitrary algorithm that adds task annotations, representing where each task will run. Here we can see that, as soon as the worker of *Task 1* finishes, since it knows it won't execute *Task 5*, it uploads *Task 1's* output as soon as it's ready. This means that once *Task 5* is ready to execute it only needs to download *Task 1's* output, and not wait for its upload, since it could have already happened. Note that this only makes sense because we know where each task we execute.

Diagram 3 is similar to Diagram 2, except that it adds *two optimizations*. The first optimization consists on *pre-warming* the worker of *Task 3* before it executes. This is done by *Task 1's* worker and can reduce latency by *eliminating cold starts*. The second optimization consists on workers *pre-loading* data dependencies they will need for future tasks. For example, *Worker 2* can start downloading *Task 1's* output at the same time it executes *Task 2*, because it will need it to execute *Task 5*. Pre-loading means parallelizing the download of data dependencies, which can help reduce workflow *makespan*.

3.3 Software Architecture

In the previous subsection, we presented our three solution components. Figure 13 illustrates our software architecture. It is based on WUKONG 2.0. We highlight WUKONG 2.0 modules that need to be extended, along with the new modules that need to be added to integrate our solution.

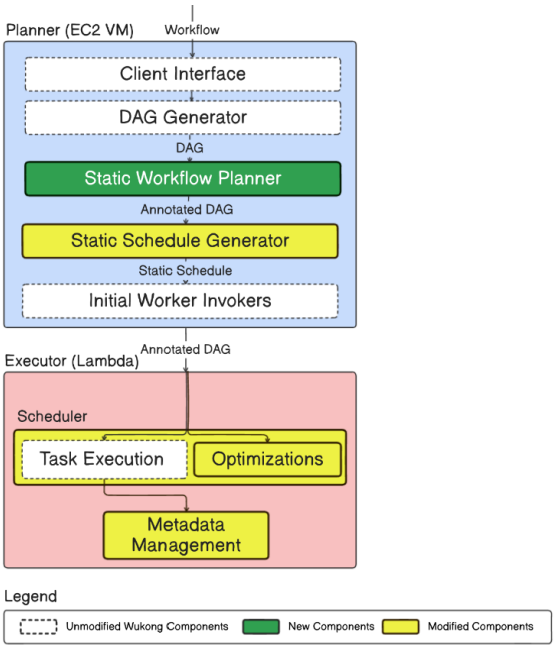

Figure 13: Software Component Modifications to WUKONG 2.0

3.3.1 Modifications to Workflow Preparation

As mentioned before, our *Static Workflow Planner* component will be integrated on the code responsible for receiving, transforming, and initiating execution of the user workflow. WUKONGs' *Static Schedule Generator* assumes we don't know where tasks will run yet. It also doesn't provide a global-view of the workflow to the workers, which we may need for some optimizations (e.g., *pre-loading data*). For this reason, we will have to modify it to partition the DAG differently or not partition it at all.

3.3.2 Modifications to Workflow Execution

Regarding WUKONG executors' software, implementing our **Metadata Management** component will require expanding the module that interacts with the Metadata Store, since we will need to upload our metrics described previously. WUKONGs' scheduling will also be modified, replacing WUKONGs' three data locality optimizations, which no longer make sense when we know where each task will execute, by our optimizations. The scheduler will be modified so that it can use the annotated DAG to understand which optimizations it should do at each moment (e.g., before executing tasks, while executing tasks, and after executing tasks).

4 Evaluation Methodology

In this section, we explain how we plan on evaluating our solution. After implementing our proposed solution on top of WUKONG 2.0 [12], we will evaluate it in order to draw conclusions about the following aspects:

- **Performance and Resource Efficiency:** Understand if our approach can reduce the *makespan* of specific workflows, when compared to other solutions using equivalent computational resources, while reducing monetary costs;
- **Scalability:** Assess the impact of our solution in the scalability of WUKONG. Scalability could be hurt if some optimizations require lots of external communication to coordinate workers. On the other hand, scalability could improve because, by knowing where each task should run, data movement inefficiencies should be reduced;
- **Bottlenecks:** The results obtained should allow us to identify potential bottlenecks that may be limiting the scalability or performance of the solution.

Naturally, we should compare our solution against WUKONG 2.0. This will be the fairest comparison that will help us understand the true impact of using historical metadata (e.g., statistical distributions of past executions' resource usage and performance, provided as input for algorithms, e.g., average, median, percentile-90, etc.) to drive workflow scheduling in serverless. Despite comparing our solution using the two planning algorithms mentioned in Section 3, we will also compare against different combinations of WUKONG's data locality optimizations (Task Clustering for Fan-out, Task Clustering for fan-in, and Delayed I/O).

The workloads to test should be the same as WUKONG 2.0: *Tree Reduction (TR)*, *Singular Value Decomposition (SVD)*, *Support Vector Classification (SVC)*, *General Matrix Multiplication (GEMM)*, *Tall-and-Skinny QR Factorization (TSQR)*. This will make it easier to compare with the original WUKONG evaluation results. Despite SeBS-Flow [51] being a benchmarking suite for comparing different aspects of existing commercial workflow platforms as they evolve, we could test our solution with some of the workloads used by SeBS-Flow, such as Word Counting, Video Analysis, or Machine Learning pipelines. To be able to properly evaluate the aspects mentioned above, we believe that the metrics to be collected should be the following:

- **CPU Time:** Measures the computational efficiency of the system, helping identify any unnecessary overhead introduced by our solution;
- **Memory Usage:** Highlights the memory efficiency of our solution, particularly in resource-constrained environments;
- **Intermediate Data Storage Usage:** Tracks the volume of external storage used for intermediate workflow results, which impacts overall cost and performance of the workflows;
- **Data Transfer Latency:** Capturing time spent waiting for or transferring data *outside the critical path*, could reveal scalability and/or resource utilization inefficiencies that could not be captured by the other metrics. In the IaaS model, the most important thing is to optimize the critical path because resources are typically *pre-allocated* and paid for regardless of utilization. However, in the serverless model, where resources are provisioned dynamically and billed based on execution time, inefficiencies outside the critical path are not negligible.
- **Monetary Cost:** Indicates the cost-effectiveness of our approach, critical in serverless environments where resources are billed based on usage;

In summary, the evaluation of our proposed solution will focus on understanding its impact on performance, scalability, and resource efficiency, while also identifying potential bottlenecks that could hinder its effectiveness. By benchmarking against WUKONG 2.0 with varying optimizations and utilizing a diverse set of workloads, we aim to demonstrate the practical advantages of leveraging historical metadata for workflow scheduling in serverless environments. The selected metrics should provide a comprehensive view of the solution's strengths and weaknesses.

5 Conclusion

In this document, we have explored serverless computing, particularly its application in executing workflows. While serverless platforms offer significant advantages like operational simplicity, automatic scalability, and cost-efficiency, their performance and resource-effectiveness for workflows is hindered by several limitations. We provided

an overview on some alternatives and extensions proposed to improve the applicability of serverless to a wider range of workloads.

Our proposed solution's goal is to enhance serverless workflow scheduling by leveraging historical data from past executions. By learning from individual task behaviour, this solution aims to make better scheduling decisions, leading to reduced workflow *makespan* and increased resource efficiency. The proposed architecture incorporates components for metadata management, workflow planning, and decentralized scheduling. The integration of this solution with WUKONG 2.0, an existing decentralized serverless scheduler, is detailed, outlining the modifications and extensions required.

The next steps involve implementing the proposed solution and conducting a comprehensive evaluation to assess its performance, scalability, resource efficiency, and identify potential bottlenecks. This evaluation will provide insights into the effectiveness of the solution and its ability to enhance serverless workflow scheduling.

Bibliography

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

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

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

[4] 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.

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

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

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

[8] M. Abdi, S. Ginzburg, C. Lin, J. M. Faleiro, 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/

[9] 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

[10] 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.

[11] 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.

[12] 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.

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

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

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

[16] Kubeless. [Online]. Available: https://github.com/vmware-archive/kubeless

[17] 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.

[18] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 152–166.

[19] Y. Yang, L. Zhao, Y. Li, S. Wu, Y. Hao, Y. Ma, and K. Li, "Flame: A centralized cache controller for serverless computing," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, 2023, pp. 153–168.

[20] 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.

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

[22] Akamai edgeworkers. [Online]. Available: https://www.akamai.com/products/serverless-computing-edgeworkers

[23] Eventarc - google cloud services events. [Online]. Available: https://cloud.google.com/eventarc/docs

[24] Lambda@edge. [Online]. Available: https://aws.amazon.com/en/lambda/edge/

[25] Logstash. [Online]. Available: https://www.elastic.co/logstash

[26] Grafana. [Online]. Available: https://grafana.com/

[27] Prometheus. [Online]. Available: https://prometheus.io/

[28] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter, "The future of scientific workflows," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 159–175, 2018. [Online]. Available: https://doi.org/10.1177/1094342017704893

[29] 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

[30] 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

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

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

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

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

[35] 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.

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

[37] F. Yao, C. Pu, and Z. Zhang, "Task duplication-based scheduling algorithm for budget-constrained workflows in cloud computing," *IEEE Access*, vol. 9, pp. 37 262–37 272, 2021.

[38] V. Arabnejad, K. Bubendorfer, and B. Ng, "Deadline constrained scientific workflow scheduling on dynamically provisioned cloud resources," *Future Gener. Comput. Syst. This special issue*, 2017.

[39] M. Hosseinzadeh, M. Y. Ghafour, H. K. Hama, B. Vo, and A. Khoshnevis, "Multi-objective task and workflow scheduling approaches in cloud computing: a comprehensive review," *Journal of Grid Computing*, vol. 18, no. 3, pp. 327–356, 2020.

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

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

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

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

[44] Mapreduce programming model. [Online]. Available: https://en.wikipedia.org/wiki/MapReduce

[45] Hadoop distributed file system. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

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

[47] Apache beam. [Online]. Available: https://beam.apache.org/

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

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

[50] Google cloud composer. [Online]. Available: https://cloud.google.com/composer?hl=pt_br

[51] L. Schmid, M. Copik, A. Calotoiu, L. Brandner, A. Koziolek, and T. Hoefler, "Sebs-flow: Benchmarking serverless cloud function workflows," *arXiv preprint arXiv:2410.03480*, 2024.

[52] Hybrid clouds. [Online]. Available: https://cloud.google.com/learn/what-is-hybrid-cloud?hl=en

[53] 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.

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

[55] 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.

[56] L. V. João Nuno Silva and P. Ferreira, "A2ha—automatic and adaptive host allocation in utility computing for bag-of-tasks," vol. 2, no. 2, 2011, pp. 171–185. [Online]. Available: https://doi.org/10.1007/s13174-011-0033-z

[57] Z. A. Khan, I. A. Aziz, N. A. B. Osman, and I. Ullah, "A review on task scheduling techniques in cloud and fog computing: Taxonomy, tools, open issues, challenges, and future directions," *Ieee Access*, vol. 11, pp. 143 417–143 445, 2023.

[58] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman, "Serverless linear algebra," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 281–295. [Online]. Available: https://doi.org/10.1145/3419111.3421287

[59] Z. Zhang, C. Jin, and X. Jin, "Jolteon: Unleashing the promise of serverless for serverless workflows," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 167–183.

A Work Schedule

A Gantt chart showing the planned work schedule is shown in Figure 14. The tasks are separated in 4 top-level categories: understanding WUKONG internals better; extending WUKONG's components to implement our solution; preparing and evaluating our solution; and lastly writing the dissertation.

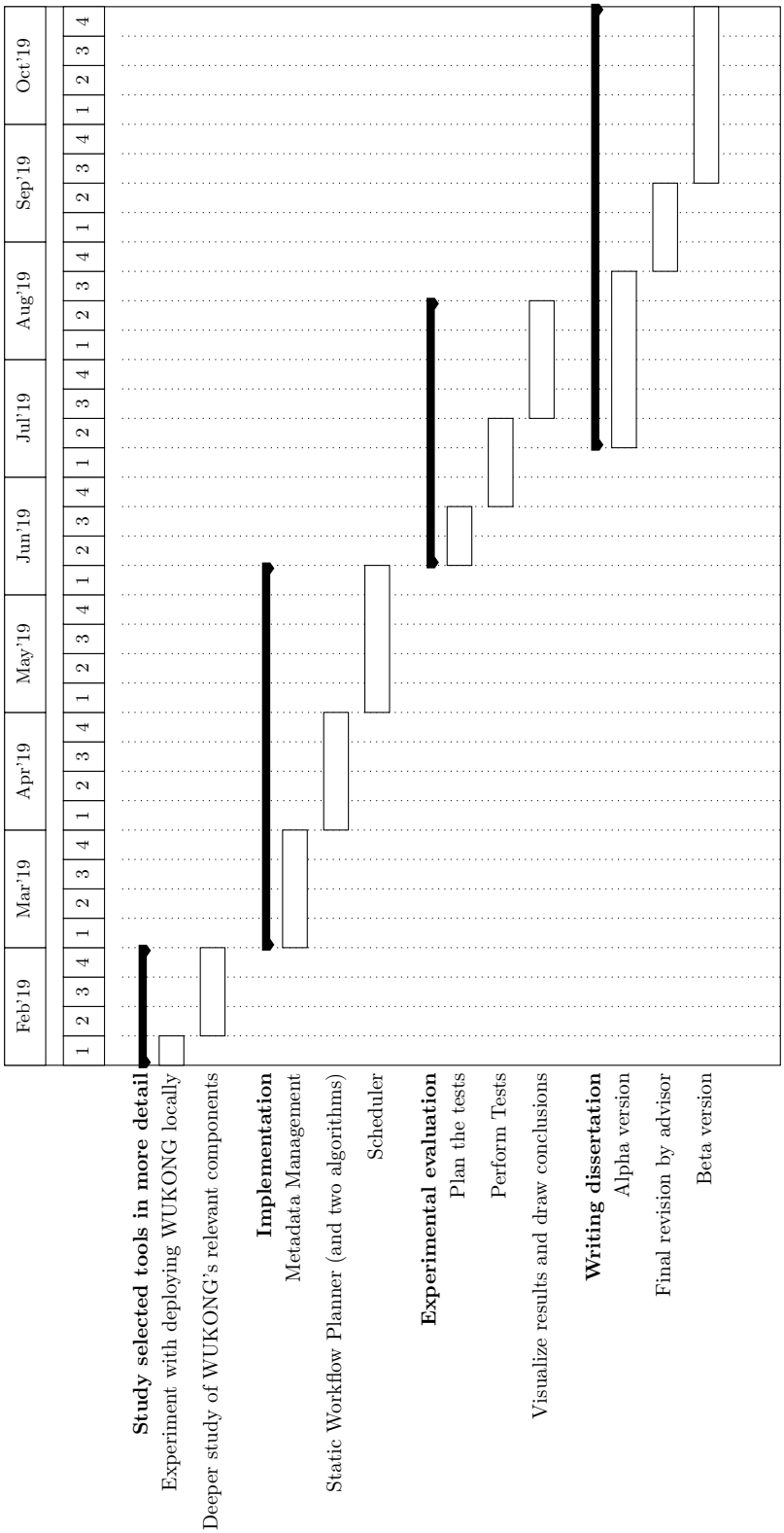

Figure 14: Planned Schedule