TÉCNICO LISBOA

Serverless Dataflows: A Decentralized Workflow Execution Engine with Predictive Planning

Diogo Alexandre Ferreira de Jesus

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Luís Manuel Antunes Veiga

Examination Committee

Chairperson: Prof. António Manuel Ferreira Rito da Silva
Supervisor: Prof. Luís Manuel Antunes Veiga
Member of the Committee: Prof. Rodrigo Fraga Barcelos Paulus Bruno

October 2025

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

Acknowledgments

I would like to thank my supervisor, Prof. Luís Veiga, for his guidance and support throughout this thesis. His advice helped me stay focused on the most important aspects of the work and provided valuable technical and non-technical insights. I am also deeply grateful to my friends and family for their constant support and encouragement throughout my academic journey.

Abstract

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

In this document, we introduce a decentralized DAG engine that leverages historical metadata to plan and influence task scheduling. Our solution encompasses metadata management, static workflow planning, and a worker-level scheduling strategy designed to drive workflow execution with minimal synchronization. We compare our scheduling approach against WUKONG, another decentralized serverless DAG engine. Our evaluation shows that our most resource-efficient planner reduces workflow execution time by **12.6%** and simultaneously lowers resource consumption by **36%** compared to the optimized WUKONG scheduling approach. For users prioritizing speed over cost, our fastest planner (Non-Uniform optimized) achieves a **57.5%** reduction in makespan compared to our most resource-efficient planner (Uniform), but this performance gain requires a **114%** increase in resource consumption.

Keywords

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

Resumo

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

Neste documento, apresentamos uma solução de *scheduling* de workflows descentralizada que utiliza metadados históricos para planear e influenciar a atribuição de tarefas a workers antes do workflow ser executado. A nossa solução abrange a gestão de metadados, o planeamento estático de workflows e uma estratégia de *scheduling* ao nível dos workers concebida para permitir a execução de workflows com sincronização mínima. Comparamos a nossa abordagem com o WUKONG, outro motor de workflows serverless descentralizado.

A nossa avaliação demonstra que o nosso algoritmo de planeamento estático mais eficiente em termos de recursos reduz o tempo de execução dos workflows em **12.6%** e, simultaneamente, diminui o consumo de recursos em **36%** em comparação com a abordagem de *scheduling* WUKONG. Para os utilizadores que priorizam a velocidade em detrimento do custo, o nosso planeador mais rápido alcança uma redução de **57.5%** no makespan em comparação com o nosso planeador mais eficiente em recursos, exigindo no entanto um aumento de **114%** no consumo de recursos.

Palavras Chave

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

Contents

1 Introduction **1**

1.1 Problem/Motivation . . . 2

1.2 Gaps in prior work . . . 3

1.3 Proposed Solution . . . 3

1.4 Document Organization . . . 3

2 Related Work **5**

2.1 Serverless Computing . . . 6

2.1.1 Advantages . . . 7

2.1.2 Limitations . . . 8

2.1.3 Research Efforts . . . 9

2.2 Workflows . . . 11

2.2.1 Workflow Definition Languages . . . 11

2.2.2 Traditional Workflow Scheduling . . . 12

2.2.3 Modern Workflow Scheduling . . . 13

2.2.3.A Stateful Serverless Functions . . . 14

2.2.4 Serverless Workflow Scheduling . . . 15

2.2.4.A Workflow Scheduling Approaches . . . 15

2.2.4.B Relevant Workflow Schedulers . . . 16

A – DEWE v3 . . . 16

B – PyWren . . . 17

C – Unum . . . 17

D – WUKONG . . . 18

2.3 Discussion/Analysis . . . 20

3 Architecture **21**

3.1 Workflow Definition Language . . . 22

3.2 Architecture . . . 24

3.3 Metadata Management . . . 26

3.4 Static Workflow Planning . . . 28
3.4.1 Workflow Simulation . . . 28
3.4.2 Planning Algorithms . . . 28
3.4.3 Optimizations . . . 35
3.5 Client . . . 39
3.6 Decentralized Workers . . . 42
4 Evaluation **47**
4.1 Emulating FaaS environment . . . 47
4.2 Research Questions . . . 49
4.3 Experimental Setup . . . 51
4.4 Results . . . 53
4.4.1 Predictions Accuracy and SLA fulfillment . . . 53
4.4.2 Overall Performance . . . 54
4.4.3 Optimizations . . . 56
4.4.4 Resource Usage . . . 57
4.5 Analysis . . . 61
5 Conclusion **63**
5.1 Achievements . . . 63
5.2 Limitations and Future Work . . . 64
Bibliography **65**
A Structure of Workflows used for Evaluation **73**

List of Figures

2.1 Serverless Computing 7
2.2 WUKONG Architecture (source: WUKONG paper [1]) 18

3.1 Simple DAG example 23
3.2 Solution Architecture 25
3.3 Planned Workflow Execution Example 36
3.4 Real Workflow Plan Visualization 39
3.5 DAG Visualization Feature for a Text Analysis Workflow 40
3.6 Workflow Real-time Dashboard 41
3.7 Choreographed Scheduling Example 44

4.1 Predictions Accuracy across all planners and SLA fulfillment rates 53
4.2 *Makespan* distribution 55
4.3 *Waiting for dependencies* distribution 55
4.4 *Worker startup time* distribution 56
4.5 Time breakdown analysis 57
4.6 Warm vs cold starts 58
4.7 Resource usage distribution (in GB-seconds) 58
4.8 *Makespan* comparison 59
4.9 Resource usage comparison 60

A.1 Matrix Multiplication Workflow (Partial) 74
A.2 Tree Reduction Workflow (Partial) 75
A.3 Text Analysis Workflow (Partial) 76
A.4 Image Transformation Workflow (Partial) 76

List of Algorithms

1 Select Relevant Samples for Predictions 29
2 Worker Assignment Algorithm (used by Uniform and Non-Uniform planners) 32
3 AssignGroup Procedure 33
4 Resource Downgrading Algorithm (used by Non-Uniform planner) 34

Listings

3.1 DAG definition example 22
3.2 Setting up and launching workflow execution 23
3.3 Task Predictions API 26
3.4 Overridable worker execution logic stages 37
4.1 Example of Delegating Tasks to Gateway 48

1

Introduction

Contents

1.1 Problem/Motivation 2
1.2 Gaps in prior work 3
1.3 Proposed Solution 3
1.4 Document Organization 3

Function-as-a-Service (FaaS) represents a serverless cloud computing paradigm that simplifies application deployment by abstracting away infrastructure management. It provides automatic, elastic scalability (potentially without limit) along with a fine-grained, pay-per-use pricing model. This has led to its widespread adoption for event-driven systems, microservices, and web services on platforms like AWS Lambda [2], Azure Functions [3], and Google Cloud Functions [4]. These applications typically benefit the most from FaaS because they are lightweight, stateless, and characterized by highly variable or unpredictable workloads, allowing them to leverage serverless platforms' on-demand scalability and cost-efficiency.

This paradigm is also increasingly used to execute complex scientific and data processing workflows, such as the Cybershake [5] seismic hazard analysis or Montage [6], an astronomy image mosaicking workflow. These applications are structured as workflows, formally represented as Directed Acyclic

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

1.1 Problem/Motivation

Despite their advantages, serverless platforms present several limitations that complicate the execution of complex workflows. Since these platforms allow scaling down to zero resources to save costs, they can also introduce unpredictable latency, known as *cold starts* [7], particularly for short-lived functions, affecting overall workflow performance. The lack of *direct inter-function communication* [8] means that tasks often have to rely on external services, such as message brokers or databases to exchange intermediate data, which can increase overhead and reduce efficiency. Interoperability between platforms is further limited by the use of platform-specific workflow definition languages, which restricts the portability of workflows across different serverless environments. Additionally, while statelessness simplifies scaling and management, it can introduce overhead and complexity for applications that require continuity or coordination across multiple function invocations. Finally, developers have limited control over the underlying infrastructure, restricting the ability to optimize resource usage or tune performance for specific workloads.

Several solutions have emerged to address the limitations of serverless platforms. Stateful functions (e.g., AWS Step Functions [9], Azure Durable Functions [10], and Google Cloud Workflows [11]) expand the range of applications that can run on serverless platforms by maintaining state across multiple function invocations, coordinating complex workflows, and providing built-in fault tolerance. Other approaches tackle limitations at the runtime level, proposing extensions to FaaS platforms (e.g., Faa$T [12], Palette [13], Lambdata [14]) or entirely new serverless architectures (e.g., Apache OpenWhisk [15]).

Other research projects focus on improved orchestration and coordination mechanisms that work on top of FaaS platforms, such as Moyer et al. [16]'s hole punching approach to allow direct inter-function communication, Pheromone [17], Triggerflow [18], FaDO [19], and FMI [20]. These solutions aim to overcome the inherent limitations of stateless functions through intelligent middleware layers that optimize function coordination, data placement, and workflow execution without requiring modifications to the underlying FaaS infrastructure.

Finally, some workflow-focused solutions (e.g., WUKONG [1], Unum [21], DEWEv3 [22]) employ scheduling strategies and workflow-level optimizations to enhance efficiency, primarily by improving data locality to bring computation closer to the data and minimize reliance on external services.

1.2 Gaps in prior work

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

1.3 Proposed Solution

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

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

1.4 Document Organization

The rest of this document is organized as follows: In Chapter 2 we do a background analysis on the serverless landscape, analyzing serverless platforms, offerings, open-source solutions and existing research work. In Chapter 3 we present our proposed solution, detailing its architecture and implementation of the core layers and components. In Chapter 4, we evaluate our proposed solution by comparing it with WUKONG's scheduling algorithm as well as with algorithms we have implemented. Finally, in Chapter 5 we conclude our work and discuss future directions for research.

2

Related Work

Contents

2.1 Serverless Computing 6
2.1.1 Advantages 7
2.1.2 Limitations 8
2.1.3 Research Efforts 9
2.2 Workflows 11
2.2.1 Workflow Definition Languages 11
2.2.2 Traditional Workflow Scheduling 12
2.2.3 Modern Workflow Scheduling 13
2.2.3.A Stateful Serverless Functions 14
2.2.4 Serverless Workflow Scheduling 15
2.2.4.A Workflow Scheduling Approaches 15
2.2.4.B Relevant Workflow Schedulers 16
A – DEWE v3 16
B – PyWren 17
C – Unum 17

D – WUKONG 18
2.3 Discussion/Analysis 20

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

2.1 Serverless Computing

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

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

The core of a typical serverless/FaaS runtime is composed of 4 layers (as depicted in Figure 2.1): 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

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

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.

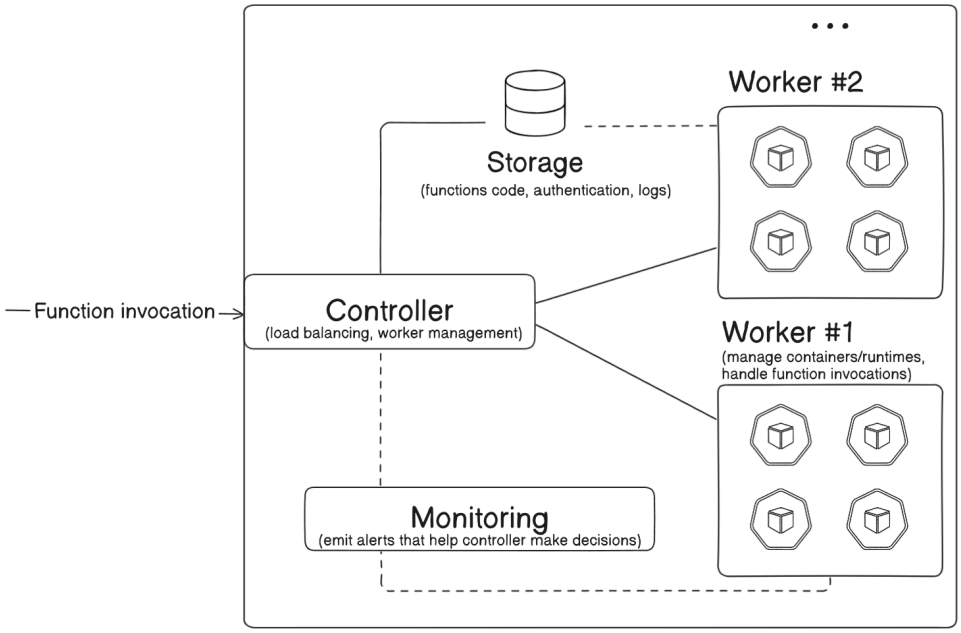

Figure 2.1: Serverless Computing

This execution model is now offered by major cloud providers, including Amazon (Lambda [2]), Google (Cloud Run Functions [4]), Microsoft (Azure Functions [3]), and Cloudflare (Workers [23]). In addition to these commercial offerings, several open-source runtimes such as OpenWhisk [15], OpenFaaS [24], and Knative [25] provide developers with alternatives for deploying FaaS in self-managed or hybrid environments.

2.1.1 Advantages

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

- **Operational Simplicity** means that developers are abstracted away from the underlying infrastructure management, without worrying about server maintenance, scaling, or provisioning. This

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

enables faster development and deployment cycles;

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

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

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

These benefits make serverless computing attractive not only for simple, event-driven applications but also for more complex workflows. Serverless workflows are a composition of multiple computational tasks that are chained together to execute applications by orchestrating individual serverless functions into a coordinated sequence. Some workflows have been successfully experimented with on FaaS. Notable examples include ExCamera [26], a highly parallel video encoding system; Montage [6], an astronomical image mosaic generator; and CyberShake [5], a seismic hazard modeling framework.

2.1.2 Limitations

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

- **Startup Latencies**: It's the time it takes for a function to start executing user code. Cold starts (explained further) can be critical, especially for functions with short execution times;

- **Isolation**: In serverless, multiple users share the same computational resources (often the same Kernel). This makes it crucial to properly isolate execution environments of multiple users;

- **Scheduling Policies**: Traditional cloud computing policies were not designed to operate in dynamic and ephemeral environments, such as FaaS;
- **Resource Configuration Complexity**: Despite the promise of abstracting infrastructure management, serverless platforms still require developers to manually configure resource allocations (such as memory limits and timeout values) for each function. This configuration significantly impacts both performance and cost, yet determining optimal settings remains a non-trivial task. While tools like AWS Lambda Power Tuning [28] have emerged to assist with empirical optimization through automated performance testing across different configurations, the burden of resource selection still falls on developers, partially undermining the serverless promise of infrastructure abstraction;
- **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. [8] portrays FaaS as a *"Data-Shipping Architecture"*, where data is intensively moved to code, through external storage services like databases, bucket storage or queues, to circumvent the limitation of inter-function direct communication. This can greatly degrade performance, while also incurring extra costs.

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

2.1.3 Research Efforts

To overcome some of the inherent limitations of traditional Function-as-a-Service (FaaS) platforms, several research initiatives have proposed architectural innovations aimed at improving performance, scalability, and orchestration. Apache OpenWhisk [15] adopts a fully event-driven, trigger-based architecture, in which functions are invoked automatically in response to events, allowing for more responsive execution and efficient resource utilization. Its design supports complex workflows and fine-grained control over function composition, making it suitable for latency-sensitive and distributed applications.

Building on similar principles, TriggerFlow [18] extends the trigger-based approach by implementing an *event-condition-action* paradigm, enabling efficient orchestration of complex workflows such as state machines and DAGs. This allows high-volume event processing, dynamic scaling, and improved fault tolerance, making it well-suited for long-running scientific and data-intensive workflows.

Another notable platform, OpenFaaS [24], fights *vendor lock-in* by emphasizing simplicity and portability, allowing developers to deploy serverless functions on a wide range of infrastructures while main-

taining an event-driven execution model. Collectively, these platforms show how architectural innovations—particularly in event handling and workflow orchestration—can mitigate many of the performance and scalability limitations found in conventional FaaS systems.

While solutions such as OpenWhisk and TriggerFlow propose completely novel serverless architectures, others such as Palette Load Balancing [13], Faa$T [12], Pocket [29], Pheromone [17], and Lambdata [14] propose extending either the FaaS runtime or the workflow definition language to address one of the most pressing limitation of the serverless paradigm: data management inefficiencies.

Palette [13] is a FaaS runtime extension that improves data locality by introducing the concept of **"colors"** as *locality hints*. These colors are parameters attached to function invocations, enabling the invoker to express the desired affinity between invocations without directly managing instances. Palette then uses these hints to route invocations with the same color to the same instance *if possible*, allowing for data produced by one invocation to be readily available to subsequent invocations, reducing the need for expensive data transfers, as it would be required in a typical FaaS runtime. This extra control that Palette provides can be used by workflow schedulers, which have insights on the data dependencies between tasks, to try co-locating tasks which share data dependencies, for example, leading to greater performance while also reducing resource utilization.

Faa$T (Function-as-a-Service Transparent Auto-scaling Cache) [12] tackles the same issue as Palette, locality, but does so on the data level, by adding a **transparent caching layer** into the FaaS runtime. Each application is assigned an in-memory *cachelet* that stores frequently accessed data, enabling subsequent invocations to reuse it without resorting to remote storage. Cachelets cooperate as a distributed cache using *consistent hashing* to share objects across instances, while pre-warming and auto-scaling mechanisms adapt the cache to workload demands. Unlike Palette, which requires user-provided hints, Faa$T operates automatically, preserving the simplicity of the serverless model, hence the "transparent" in its name.

Similarly, Lambdata [14] improves data locality by relying on explicit **data intents** provided by the developer. Functions declare which objects they will read and write, allowing the controller to co-locate invocations that share data dependencies on the same worker and reuse a local cache. This reduces remote storage accesses and data transfer overheads. Compared to Palette's flexible color hints, Lambdata's data intents are more precise but place stricter requirements on developers, while contrasting with Faa$T's fully automated and distributed approach. Contrasting with Palette, Lambdata requires less effort from the developer, but at the cost of reduced flexibility.

2.2 Workflows

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

2.2.1 Workflow Definition Languages

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

Despite the existence of other ways to express workflows, due to the simplicity of writing and interpreting DAGs, most systems and libraries use this representation. For instance, Apache Airflow [30] uses DAGs to define and schedule workflows defined in Python. Similarly, Dask [31], 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) [32] is a way HTCondor [33] (distributed computing job manager) users can organize independent jobs into workflows, also in the form of DAGs.

However, there are more flexible alternatives to define workflows. YAWL (Yet Another Workflow Language) [34] is a *workflow language* that provides a highly expressive framework for workflow management, capable of supporting a wider range of workflow patterns. YAWL uses Petri networks [35] instead of DAGs to model workflows. This allows YAWL to handle more complex control-flow structures, such as loops, parallelism, and advanced synchronization patterns, offering greater flexibility and power in defining and managing intricate workflows.

While using more capable and flexible workflow languages, such as *YAWL* (Yet Another Workflow Language) allows the representation of more complex workflow patterns, most of the tools used for

defining and running scientific workflows, like *Apache Airflow*, *Dask*, and HTCondor's DAGMan use the Directed Acyclic Graph format. This is because DAGs effectively model the majority of scientific workflows, which typically involve non-cyclic dependencies, making them simpler to compose, deploy, understand, debug, and visualize.

2.2.2 Traditional Workflow Scheduling

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

To alleviate some of the developers and researchers' pain points during these steps while scheduling workflows on more traditional *Infrastructure as a Service* (IaaS) platforms, several data processing and workflow scheduling frameworks have emerged. Among the platforms for **data processing pipelines** are Apache Spark [36], Apache Flink [37], and Apache Hadoop [38], 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 [39] and HTCondor [33], which manage the execution and coordination of complex sequences of tasks, ensuring that dependencies are handled and resources are allocated effectively. These frameworks help streamline both the data processing and the management of workflows.

Apache Hadoop provided a foundation for large-scale data processing when it introduced the MapReduce [40] paradigm, supported by HDFS and YARN [41] for reliable storage and resource management. Apache Spark provides a flexible distributed model with rich libraries for analytics and efficient task dependency management, while Apache Flink specializes in real-time stream processing with low latency and robust state handling. In terms of workflow orchestration, Apache Oozie specializes in coordinating Hadoop-based tasks, whereas HTCondor targets high-throughput scientific workflows, efficiently managing complex dependencies. This has been extended with adaptive scheduling based on the relevance of the new input received (and output produced) in each task [42, 43].

Similarly, Dask [31] and its distributed scheduler (Dask Distributed [44]) extend the familiar Python ecosystem to large-scale data and compute workloads, enabling parallel execution of array, dataframe, and machine learning operations across clusters, with a lightweight task graph scheduler that supports

both batch and interactive computation. Together, these frameworks illustrate the range of solutions available for executing and coordinating workflows on top of the IaaS model, spanning batch and streaming data as well as data-centric and scientific computing environments.

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

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

2.2.3 Modern Workflow Scheduling

The limitations of IaaS-based frameworks, such as those mentioned previously, have led to a new generation of workflow orchestrators designed for flexibility, usability, and ease of deployment. Unlike earlier systems bound to static clusters and rigid DSLs, modern platforms embrace Python APIs, containerization [3], and cloud-native [4] principles. Tools such as Prefect [45], Dagster [46], and Airflow [30] prioritize developer productivity and observability, while Argo Workflows [47] leverages Kubernetes [48] as its native execution environment. Another interesting project is Apache Beam, a cloud-native *unified programming model* for batch and streaming data pipelines designed to be executed across multiple backends, with some of the most popular implementations being Google Cloud Dataflow [49]. These solutions mark a shift from infrastructure management toward higher-level abstractions that integrate seamlessly with cloud platforms and services.

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

2.2.3.A Stateful Serverless Functions

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

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

Prominent examples include AWS Step Functions [9], Google Cloud Workflows [11], Azure Durable Functions [10], and Cloudflare Workflows [50]. Despite differences in design and integration, they share the goal of simplifying complex coordination atop serverless platforms. AWS Step Functions offers JSON- or YAML-based state machines using the Amazon States Language (ASL)[5], enabling workflows that may last up to one year. Google Cloud Workflows provides YAML- or JSON-based definitions with a maximum duration of 60 minutes per execution, tightly integrated with Google Cloud services such as BigQuery and Cloud Run. Azure Durable Functions follows a code-centric approach in which developers write an *"orchestrator function"* in languages such as C#, JavaScript, or Python, with workflow durations of up to 30 days. Finally, Cloudflare Workflows emphasizes lightweight, edge-native [6] orchestration, optimized for event-driven scenarios at the edge.

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

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

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

2.2.4 Serverless Workflow Scheduling

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

2.2.4.A Workflow Scheduling Approaches

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

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

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

- **Decentralized and Choreographed scheduling:** In this model, the responsibility for orchestration is distributed across the worker nodes rather than concentrated in a central entity. Each node

independently manages task execution, coordinates with peers, and propagates state updates as necessary. This approach eliminates the need for dedicated scheduler infrastructure, mitigates bottlenecks, and enables faster scaling to thousands of concurrent functions. However, this model introduces greater complexity in ensuring fault tolerance and consistent state propagation across distributed, ephemeral environments. Ensuring that tasks execute *exactly once* becomes particularly challenging, requiring stronger coordination and consensus mechanisms. To address these issues, algorithms such as Paxos [51], Raft [52], or coordination services like ZooKeeper [53] can be employed, along with localized snapshots or lightweight distributed state stores, to maintain workflow coherence and reliability without relying on centralized scheduling. Most existing solutions, however, assume that tasks are *idempotent*, so that repeated execution does not produce unintended side effects, simplifying failure recovery and avoiding the need for strict exactly-once guarantees.

2.2.4.B Relevant Workflow Schedulers

To illustrate the spectrum of serverless workflow scheduling strategies, we now discuss representative systems that embody each approach and inspired our work. DEWE v3 [22] demonstrates a hybrid (IaaS and FaaS) queue-based model, where tasks are distributed through dedicated queues depending on their expected execution time. PyWren [54] exemplifies a more centralized scheduling paradigm, in which the user or submission machine maintains control over task execution while workers simply carry out assigned functions. Finally, Unum [21] and WUKONG [1] showcase decentralized choreographed orchestration, distributing workflow logic across workers to achieve high scalability and reduced reliance on central schedulers. Examining these systems provides concrete examples of the trade-offs and optimizations inherent to each scheduling strategy.

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

Jobs that fail to execute on FaaS workers, for being longer than expected and exceeding execution limits imposed by the platform, are redirected to the serverful queue. This dual-execution model enables DEWE to accommodate workflows with diverse resource consumption patterns.

This system proves particularly effective for scientific workflows, such as Montage [6], where task durations and resource requirements vary significantly. However, this hybrid approach introduces specific trade-offs. Latency-sensitive workflows may be slowed by job queuing overhead. In addition, hybrid deployments often lead to resource underutilization, as serverful workers may sit idle when most tasks are executed on FaaS. Finally, the centralized workflow manager can become a scalability bottleneck when handling many short tasks.

B – PyWren PyWren [54], representing one of the pioneering pure serverless approaches, demonstrates the potential and limitations of leveraging unmodified serverless infrastructure for distributed computation. Built atop AWS Lambda, PyWren focuses on executing arbitrary Python functions as stateless serverless functions with minimal user management overhead, automatically handling function execution, dependencies, S3 bucket storage for serialized code and intermediate data. The system is ideal for *embarrassingly parallel* workloads, also known as *"bag-of-tasks"* scenarios, with many independent, parallel tasks such as simple data transformations, scientific simulations, parallel model training, and large-scale media processing. While PyWren's serverless orchestration model provides excellent scalability and removes the burden of infrastructure management, its simplicity limits its applicability. It is not well-suited for workflows with complex dependency structures or those that require sharing large intermediate results through object storage. Moreover, latency-sensitive applications are disadvantaged by function cold starts, since PyWren does not include mechanisms to mitigate their impact.

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

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

instance executing only its specific task before triggering the next function.

D – WUKONG WUKONG [1] represents the most innovative approach among these solutions, designed as a *decentralized choreographed locality-enhanced* serverless workflow engine. Built on top of Dask's programming model and DAG generation capabilities [31], WUKONG reimagines the execution layer to address the limitations of traditional serverful models like Dask Distributed while maximizing the advantages of serverless computing. The system focuses on improving scale-out speed and enhancing data locality to minimize large object movement. WUKONG's architecture, depicted in Figure 2.2, is divided into static components (operating before workflow execution) and dynamic components (operating during execution). The static scheduler includes a **DAG Generator** that leverages Dask's library to convert Python code into DAGs, a **Schedule Generator** that creates `n` static schedules for `n` root/leaf nodes (each containing every reachable task in a depth-first search starting at that node), **Initial Task Executor Invokers** that launches the first Lambda instances for each root task, and a **Subscriber Process** on the client that waits for and downloads final results.

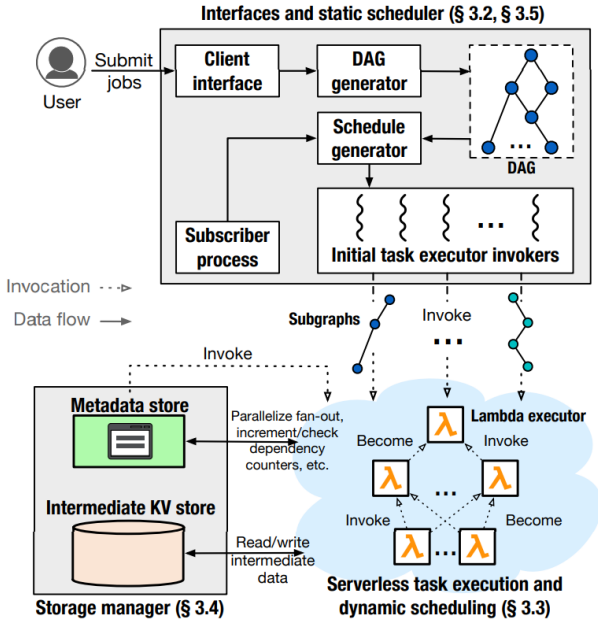

Figure 2.2: WUKONG Architecture (source: WUKONG paper [1])

After receiving the initial schedules, FaaS workers (referenced to as *AWS Lambda Executors*) drive workflow execution. Workers execute tasks until they encounter a *fan-out*, at which point they transfer data to intermediate storage, execute 1 of the `N` fan-out tasks and invoke *N - 1 new* executors for the other tasks. Then, when they find a *fan-in*, the group of executors that reach the common fan-in node cooperate using a *dynamic scheduling* model to select only one executor to

proceed. This coordination is managed through a shared **dependency counter** in a Key-Value Store (KVS). Each involved executor *atomically* updates this counter; the one whose update satisfies the final input dependency for the fan-in task will execute that task and continue along its static schedule. The other executors transfer their intermediate data to storage, and then stop their execution, decreasing the workflow parallelism.

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

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

While WUKONG represents a significant advance in serverless workflow orchestration through its decentralization, its scheduling and optimizations remain limited. The system bases decisions only on the next stage of the workflow (i.e., one-to-one, fan-in, or fan-out transitions). We refer to this as *one-step scheduling*, since it relies solely on information about the next step. Crucially, WUKONG does not exploit the global knowledge of the workflow structure, even though the entire

workflow structure is known before execution begins. Also, its optimizations rely on heuristic-based strategies that can lead to suboptimal performance when workflow behavior deviates from expected patterns.

Moreover, WUKONG exhibits an **extreme trade-off between data locality and resource contention**. Without applying its optimization strategies, WUKONG experiences *no resource contention* but also *no data locality*, since every fan-out creates new executors and most data exchanges occur through storage. When optimizations such as task clustering and delayed I/O are enabled, the system shifts to the opposite extreme, achieving *maximum data locality* at the cost of *high resource contention*, as executors hold onto large objects in memory and prolong execution to capture downstream work. This behavior highlights the absence of adaptive mechanisms to balance locality and resource utilization. We argue that the key to improving efficiency lies in achieving a **dynamic balance** between data locality and resource contention, adjusting executor behavior based on workload characteristics and runtime conditions rather than fixed and extreme heuristics.

2.3 Discussion/Analysis

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

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

3

Architecture

Contents

3.1 Workflow Definition Language 22
3.2 Architecture 24
3.3 Metadata Management 26
3.4 Static Workflow Planning 28
3.4.1 Workflow Simulation 28
3.4.2 Planning Algorithms 28
3.4.3 Optimizations 35
3.5 Client 39
3.6 Decentralized Workers 42

In this chapter, we will present the architecture, as well as some implementation details of our decentralized serverless workflow execution engine. Here we will describe how a workflow is defined and configured, we will then present an architecture overview of the solution, followed by a more in-depth description of the main layers and components. We will go over how workflow plans are created, optimizations they can apply, and how they are executed in a choreographed manner by FaaS workers.

3.1 Workflow Definition Language

We will now present the *workflow language* with which users can create tasks and compose them into workflows. It is heavily inspired by WUKONG and Dask's: the user can create workflows by composing individual Python functions, as shown in Listing 3.1. In this example, we define two tasks, `task_a` and `task_b`, and then compose them into a DAG by passing their results as arguments to the next task. The resulting workflow structure is illustrated in Figure 3.1. Python's simplicity, readability, and widespread popularity, especially among data scientists and researchers, also motivated us to choose this language for defining workflows, lowering the barrier to entry while enabling powerful parallel execution.

In Dask and WUKONG, tasks are also created by decorating [1] Python functions with a special decorator, named `@delayed` (equivalent to our `@DAGTask`), and then composed into a DAG by calling these decorated functions, similarly to our approach. The workflow execution is also triggered by calling `compute()` on the last/sink task, as shown in Listing 3.2.

Listing 3.1: DAG definition example

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

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

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

When `task_a(a1)` is invoked, it doesn't actually run the user code. It instead creates a representation of the task, which can be passed as argument to other tasks. The workflow planning and execution only happens once `.compute()` is called on the last/sink task (`a4`), as shown in Listing 3.2. When `compute()` is called, we can create a representation of the entire workflow structure by backtracking the task dependencies.

[1] https://www.geeksforgeeks.org/python/decorators-in-python/

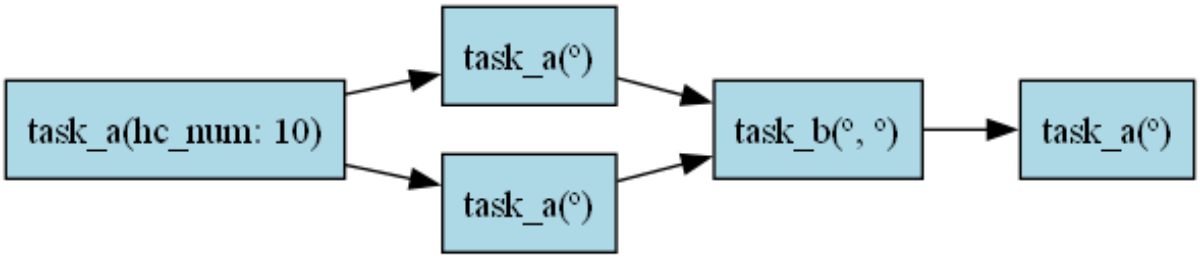

Figure 3.1: Simple DAG example

Calling `task_a(10)` generates a corresponding task representation, `DAGTaskNode`, which captures essential information about the task. This includes a unique task identifier, the tasks' dependencies, which may be other tasks or "hardcoded" data. Hardcoded data refers to values that are known prior to workflow execution; in this example, the value `10` is hardcoded. In a text processing workflow, for instance, the bytes of a file to be analyzed would be considered hardcoded as long as they are available before the workflow starts and are passed as arguments to a task. If a task function argument is itself an instance of `DAGTaskNode`, as `a1` is in `task_a(a1)`, it is treated as a dependency of `task_a`. Once a `DAGTaskNode` is created, its upstream tasks and dependencies are automatically discovered. This enables constructing a representation of the entire workflow prior to its execution by backtracking the task dependencies, starting from the *sink node*.

In serverless workflows, a common pattern is to pass storage references (e.g., cloud object storage URLs) as arguments to tasks. This behavior is not currently supported by our workflow language, as the proposed optimization and prediction mechanisms rely on analyzing the actual data produced and consumed by functions rather than indirect references. In future iterations, this limitation could be addressed through **code instrumentation**, wherein storage access APIs are intercepted to record metrics such as the volume of data transferred and the corresponding I/O latency.

To ensure portability and eliminate the need to register each user-defined function in the cloud provider's function registry, the task's executable code is directly embedded within its representation as a Python `Callable`. Later, the `DAGTaskNode` is serialized, using `cloudpickle` [2], allowing it to be transmitted to storage, and retrieved by workers for execution.

Listing 3.2: Setting up and launching workflow execution

```
result = a4.compute(
    dag_name="simpledag",
    config=Worker.Config(
        faas_gateway_address=...,
        intermediate_storage_config=(ip, port, password),
```

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

```
        metrics_storage_config=(ip, port, password),
        planner_config=UniformPlanner.Config(
            sla=sla,
            worker_resource_configuration=TaskWorkerResourceConfiguration(memory_mb=1024),
            optimizations=[PreLoadOptimization, PreWarmOptimization]
        )
    )
)
```

One limitation of this DAG definition language is that it doesn't support "dynamic fan-outs" (e.g., creating a variable number of tasks depending on the result of another task) on a single workflow. This is a powerful and expressive feature, but that is seldom supported in other DAG definition languages (e.g., Dask, WUKONG, Unum, Oozie [39] do not support it). These languages require the user to split the workflow into multiple workflows, one for each *dynamic fan-out*: one workflow runs up to the task that generates a list of results, while a second workflow starts with a number of tasks that depends on the size or contents of that list.

Apache AirFlow [3] supports this feature through an extension to their DAG language, allowing a variable number of tasks to be created at run-time depending on the number of results produced by a previous task. Implementing similar functionality is possible, but it would reduce the accuracy of predictions. This is because we would also need to predict the expected fan-out size, and any errors in that prediction could amplify inaccuracies in the predictions for the rest of the workflow.

We will now present our solution architecture overview, highlighting the core layers of our decentralized serverless workflow execution engine.

3.2 Architecture

The overall architecture and logical flow of our decentralized serverless workflow execution engine is organized into 3 high-level layers. Figure 3.2 provides an overview of this architecture. The upper part represents the components that run on the user's machine, while the lower part represents the components that run outside the user's machine.

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

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

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

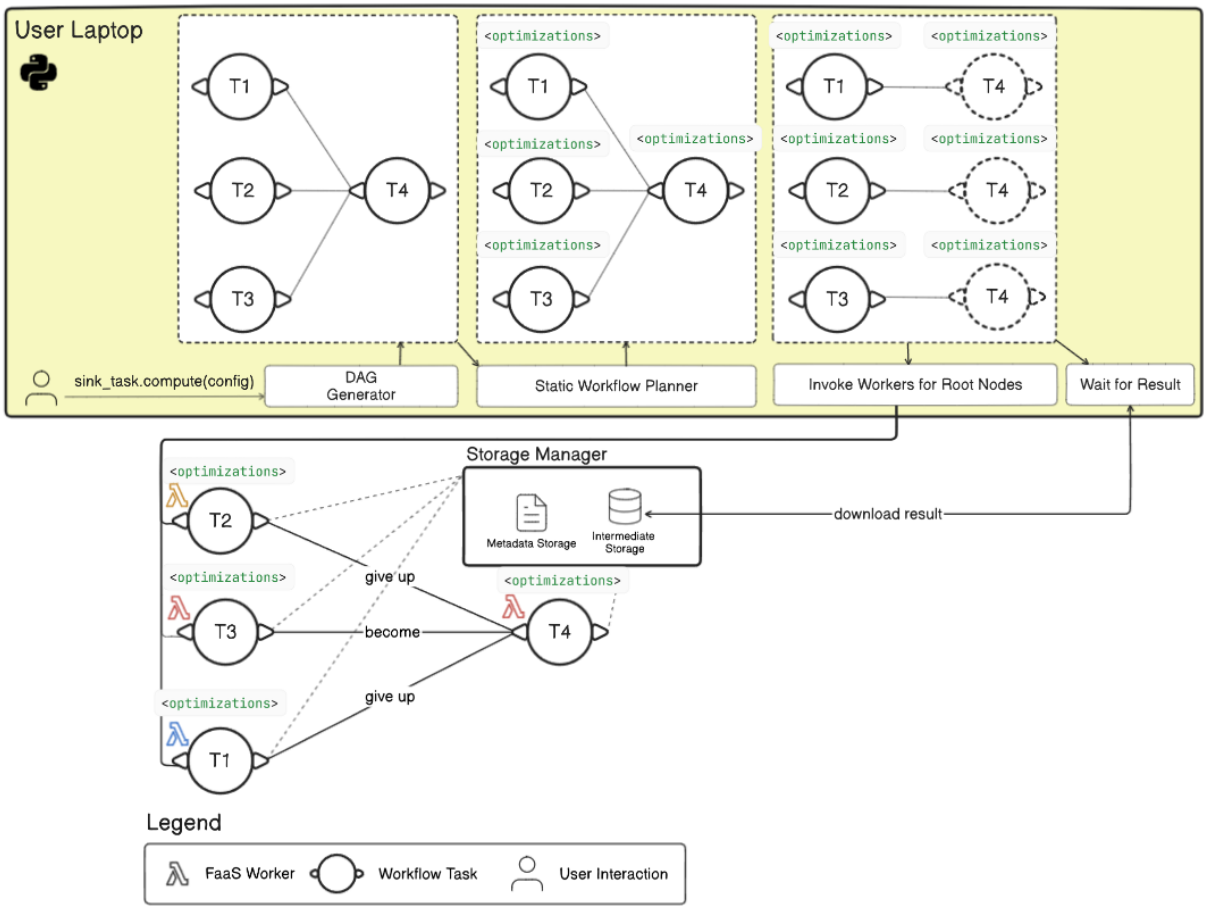

Figure 3.2: Solution Architecture

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

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

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

There are 3 distinct computational entities involved in this system:

- **User Computer**: Responsible for creating workflow plans, submitting them (triggering workflow execution), and receiving its results. The planning phase also happens on this computer, right before a workflow is submitted for execution;
- **Workers**: These are the FaaS workers (often running in containerized environments), that execute one or more tasks. The decentralization of our solution is due to the fact that these workers are responsible for scheduling of subsequent tasks, delegating tasks and launching new workers when needed without requiring a central scheduler. Lastly, they are also responsible for collecting and uploading metadata;
- **Storage**: Consists of an *Intermediate Storage* for intermediate outputs which may be needed for subsequent tasks and a *Metadata Storage* for information crucial to workflow execution (e.g., notifications about task readiness and completion).

3.3 Metadata Management

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

Storing these metrics powers a prediction API, shown in Listing 3.3. To improve accuracy, metrics are kept separate for each workflow. As a result, even if two workflows use the same function or task code, their metrics are stored independently. This design choice reflects our assumption that different workflows may follow different execution patterns. To avoid introducing runtime overhead, metrics are only uploaded before the worker shuts down.

The prediction methods take an additional parameter, `SLA` (Service-level Agreement), which is specified by the user and influences the selection of prediction samples. For example, `SLA="average"` will use the average of the historical samples, whereas `SLA=Percentile(80)` will return a more conservative estimate, where the 80th percentile of the historical samples is used, providing a higher estimate. This parameter is provided by the user, and is used by the `PredictionsProvider API` to provide estimates based on historical data.

Listing 3.3: Task Predictions API

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

```
3    def predict_worker_startup_time(state: 'cold' | 'warm', resource_config, sla) -> float
4    def predict_data_transfer_time(data_size_bytes, resource_config, sla) -> float
5    def predict_execution_time(task_name, input_size, resource_config, sla) -> float
```

When collecting metrics such as worker startup time, data transfer time, and task execution time, we associate them with the worker resource configuration used to execute the task (e.g., memory and CPU). To improve prediction accuracy, our method follows two strategies. If there are enough historical samples for the exact resource configuration being predicted, we use only those samples. Otherwise, when samples for the target configuration are insufficient, we apply a normalization approach: we adjust samples from other memory configurations to a baseline, use these normalized values to estimate execution time, and then rescale the result to the target configuration.

After filtering the samples by their resource configuration, we apply the sample selection algorithm shown in Algorithm 1, which identifies the most relevant historical samples for each prediction. Given a reference value (the value for which to predict), the method first computes a *baseline* from all available samples (line 2). The baseline is determined using the user-specified `SLA`, thereby adapting the prediction to user-defined expectations.

Next, the algorithm progressively searches for samples whose associated input sizes fall within increasing proximity thresholds relative to the reference value. Each threshold increment expands the acceptable distance range by 5% of the computed baseline, up to a maximum of 100%. Within each iteration, samples are categorized as being below, above, or exactly matching the reference value (lines 8-19), allowing the algorithm to maintain a symmetric distribution of samples around the target.

Once enough samples are found within a given threshold (line 21), they are sorted by proximity to the reference value (lines 23-24). The algorithm then selects up to a maximum number of samples (`max_samples`), prioritizing those closest to the reference and ensuring balance between samples above and below the target (lines 23-30). If the maximum capacity is not reached, the remaining slots are filled with the next-closest available samples (lines 32-34).

If even after considering all thresholds (up to 100% of the established baseline) no sufficient number of samples is found, the algorithm falls back to selecting the globally closest samples (lines 39-42). This guarantees that a prediction can always be made, even when the historical data is sparse, though it may not be as accurate.

In addition to using this algorithm to select the most relevant samples for each prediction, we restrict the samples to the same workflow and planner. For instance, data collected while executing the *Text Analysis* workflow with the *Uniform* planner will not be used to predict the execution time of the same workflow under the *Non-Uniform* planner. By limiting the sample set in this way, we ensure that the selected samples are directly comparable to the target scenario, reducing variability introduced by differing workflow behaviors or planner strategies.

This procedure ensures that the prediction leverages historical samples most similar to the target

input, which tends to be very accurate when a workflow is executed multiple times with the same or similar workloads, allowing the estimates to become accurate without requiring many previous workflow instances.

3.4 Static Workflow Planning

This layer executes on the user side, and it receives the workflow representation and a workflow planning algorithm chosen and parametrized by the user (as shown in Listing 3.2). Its job is to execute the planning algorithm, providing it access to the predictions exposed by the Metadata Management layer (Section 3.3).

3.4.1 Workflow Simulation

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

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

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

3.4.2 Planning Algorithms

For each task, the planner assigns both a `worker_id` and a resource configuration (vCPUs and memory). The `worker_id` specifies the worker instance that must execute the task—analogous to the “colors” in Palette Load Balancing [13], but in our case this assignment is mandatory rather than advisory, giving strict control over execution locality. Two tasks assigned the same `worker_id` should be executed on the same worker instance. If `worker_id` is not specified, workers will, at run-time, have to decide whether to execute or delegate those tasks, similar to WUKONG’s [1] scheduling. We refer to these workers as *"flexible workers"*.

Algorithm 1 Select Relevant Samples for Predictions

Require: $reference\_value$, $all\_samples$, SLA, $max\_samples$, $min\_samples$
Ensure: Set of values from samples closest to $reference\_value$

▷ Compute baseline based on SLA percentile
$baseline \leftarrow$ PERCENTILE($\{comparable_to_ref \mid (value, comparable_to_ref) \in all\_samples\}, SLA$)
▷ Try increasing thresholds to find enough samples
for $threshold\_pct \leftarrow 5$ **to** 100 **step** 5 **do**
  $distance\_threshold \leftarrow baseline \times (threshold\_pct/100)$
  $below\_samples \leftarrow \emptyset$, $above\_samples \leftarrow \emptyset$, $exact\_samples \leftarrow \emptyset$
  ▷ Categorize samples by their position relative to reference
  for all $(value, comparable_to_ref) \in all\_samples$ **do**
    $signed\_dist \leftarrow comparable_to_ref - reference\_value$
    if $|signed\_dist| \leq distance\_threshold$ **then**
      if $signed\_dist < 0$ **then**
        $below\_samples$.ADD($(|signed\_dist|, value)$)
      else if $signed\_dist > 0$ **then**
        $above\_samples$.ADD($(|signed\_dist|, value)$)
      else
        $exact\_samples$.ADD($value$)
      end if
    end if
  end for
  $total\_found \leftarrow |below\_samples| + |above\_samples| + |exact\_samples|$
  if $total\_found \geq min\_samples$ **then**
    ▷ Sort by distance and select balanced samples
    $below\_samples$.SORTBYDISTANCE()
    $above\_samples$.SORTBYDISTANCE()
    $remaining \leftarrow max\_samples - |exact\_samples|$
    $per\_side \leftarrow \lfloor remaining/2 \rfloor$
    ▷ Select closest samples from each side
    $selected \leftarrow exact\_samples$
    $selected \leftarrow selected \cup below\_samples[0 : \min(per\_side, |below\_samples|)]$
    $selected \leftarrow selected \cup above\_samples[0 : \min(per\_side, |above\_samples|)]$
    ▷ Fill remaining budget from available samples
    if $|selected| < max\_samples$ **then**
      FILLREMAINING($selected$, $below\_samples$, $above\_samples$, $max\_samples$)
    end if
    return $\{value \mid value \in selected\}$
  end if
end for
▷ Fallback: not enough samples even at 100% threshold
$samples\_with\_dist \leftarrow \{(|comparable_to_ref - reference\_value|, value) \mid (value, comparable_to_ref) \in all\_samples\}$
$samples\_with\_dist$.SORTBYDISTANCE()
return $\{value \mid (dist, value) \in samples\_with\_dist[0 : min\_samples]\}$

In our planners and implementation, we primarily focus on fixed `worker_ids`, as this approach provides planners with finer-grained control over task execution locality. It also reduces network overhead, since workers are aware of the specific tasks assigned to them and can simply subscribe to the corresponding `TASK_READY` events, eliminating the need for broad or repeated coordination messages.

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

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

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

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

Both the **Uniform** and **Non-Uniform** planners follow a two-phase approach for task allocation: resource configuration assignment followed by worker ID assignment. The planners differ in their resource allocation strategies. The **Uniform planner** applies a single, user-specified CPU and memory configuration to all tasks, while the **Non-Uniform planner** selects the most powerful configuration from the user-specified options for each task. After resource configuration, both planners employ the logic detailed in Algorithm 2 for worker ID assignment. This algorithm presents the worker assignment procedure used by our planner to distribute workflow tasks across workers, trying to achieve a **balance between data locality and resource contention**. The algorithm iterates over the DAG nodes in topological order, ensuring that upstream tasks are always processed before their dependents. Each node is assigned to a worker depending on the structure of its neighborhood (fan-in or fan-out pattern) and on predictions of task execution time and output size.

In this algorithm, root nodes (those without upstream dependencies), are grouped together for assignment (lines 7-8). For simple linear dependencies (1→1 relationships), the downstream task inherits the same worker as its upstream task, allowing direct data reuse without intermediate storage (lines 11-12). When a node fans out to multiple downstream tasks (1→N), those downstream tasks are clustered together and assigned as a group (lines 14-15). Conversely, for fan-in patterns (N→1), the algorithm

assigns the downstream node to the worker responsible for producing the largest upstream output (lines 18-20), maximizing data locality by placing the consumer task where the majority of input data already resides.

The AssignGroup function, presented in Algorithm 3, is responsible for clustering tasks into execution groups and assigning them to workers. After retrieving the predicted execution times and output sizes for each task (lines 4-5), the algorithm classifies tasks as either *long* or *short* depending on whether their predicted execution time is above or below the median (lines 7-8). This categorization allows the planner to balance worker load.

The function proceeds in four stages:

1. **Local clustering (lines 10-14):** If an upstream worker is available, a subset of *short* tasks with larger output sizes (up to a predefined maximum cluster size) is assigned to the same worker. This promotes *data locality*, reducing data transfer costs by executing tasks near their data sources;
2. **Balanced pairing (lines 16-22):** The algorithm pairs each remaining *long* task with a few *short* ones, creating mixed groups, distributing workload more evenly;
3. **Grouping remaining short tasks (lines 24-28):** Any remaining unassigned *short* tasks are grouped together and allocated to new workers, up to the maximum cluster size;
4. **Grouping remaining long tasks (lines 30-35):** This final phase is only reached when there are more long tasks than short ones, meaning some long tasks could not be paired in the previous phase. In such cases, the remaining long tasks are grouped into smaller clusters (half the normal cluster size) to try reducing resource contention.

After this, the **Non-Uniform** planner runs an additional algorithm, shown in Algorithm 4, that attempts to downgrade resources of workers *outside the critical path* as much as possible without introducing a new critical path, by iteratively simulating the effect of downgrading resources of workers *outside the critical path* with different configurations.

The algorithm begins by identifying workers whose tasks are outside the critical path (lines 3-9). Then, for each of those workers, we attempt to downgrade its resources by iterating through the list of use-provided resource configurations, simulating progressively weaker configurations (lines 11-41).

After a configuration is added to all tasks of the target worker (lines 17-21), the simulation layer (explained in Subsection 3.4.1) is used to simulate the critical path duration (line 23). If the downgrade does not increase the critical path time, it is accepted. This process repeats for each worker until no more configurations are available to experiment or the critical path time is exceeded, in which case it reverts to the previous *acceptable* resource configuration (line 35).

With the information they have access to, planners can estimate whether it is worthwhile to offload a task to a more powerful worker. This involves weighing the overhead of uploading the input data, waiting

Algorithm 2 Worker Assignment Algorithm (used by Uniform and Non-Uniform planners)

Require: $nodes$, $predictions$, $MAX\_CLUSTERING$

$assigned \leftarrow \emptyset$

▷ nodes are topologically sorted

for all $n \in nodes$ **do**
  if $n \in assigned$ **then**
    continue
  end if
  if $n.upstream = \emptyset$ **then** ▷ root nodes
    $roots \leftarrow \{r \in nodes \mid r.upstream = \emptyset \wedge r \notin assigned\}$
    AssignGroup(*null*, $roots$)
  else if $|n.upstream| = 1$ **then** ▷ 1→1 or 1→N
    $u \leftarrow n.upstream[0]$
    if $|u.downstream| = 1$ **then**
      AssignWorker($[n]$, $u.worker$) ▷ reuse worker
    else ▷ 1→N
      $fanout \leftarrow \{d \in u.downstream \mid d \notin assigned\}$
      AssignGroup($u.worker$, $fanout$)
    end if
  else ▷ N→1 (assign to worker of upstream task with the largest total output)
    $outputs \leftarrow \{u.worker : predictions.output\_size(u) \mid u \in n.upstream\}$
    $worker\_w\_greatest\_acc\_output \leftarrow \arg\max_{w \in outputs} outputs[w]$
    AssignWorker($[n]$, $worker\_w\_greatest\_acc\_output$)
  end if
end for

for the worker to be provisioned, and then executing the task, against the alternative of simply executing the task on the current, less powerful worker.

To facilitate the implementation of new planners, the Python class `AbstractDAGPlanner`, from which all planners must inherit, provides a set of utility methods: a method to perform a topological sort of the DAG nodes, ensuring that we don't iterate on a node without first iterating on its upstream nodes; a method to estimate the input size of a task by summing the sizes of its hardcoded inputs and the predicted outputs of its upstream tasks; a method to simulate the execution of an entire workflow, returning predicted metrics for all tasks; and a method to compute the critical path of the workflow using the results of the simulation.

One of these validations ensures that *all tasks assigned to the same worker form a continuous, uninterrupted branch* within the workflow graph. In other words, a worker cannot be assigned to a group of tasks, skipped for intermediate ones along the dependency chain, and later reassigned to a downstream task. This rule introduces structural predictability, ensuring that task assignments for a given worker remain contiguous in the DAG. As a result, when delegating tasks, the system can determine whether a worker is already running and can simply receive a `TASK_READY` event (as described in Section 3.6), or whether a new worker launch is required. This constraint implicitly encourages planners to allocate tasks in ways that minimize worker idle time.

Algorithm 3 AssignGroup Procedure

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

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

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

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

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

Algorithm 4 Resource Downgrading Algorithm (used by Non-Uniform planner)

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

$workers\_outside \leftarrow \emptyset$

▷ 1) Identify workers outside the critical path

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

  $wid \leftarrow n.worker\_id$

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

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

  end if

end for

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

▷ 2) Attempt downgrade for each worker outside critical path

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

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

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

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

    $trial \leftarrow configs[i].\textsc{Clone}(wid)$

▷ Apply trial configuration to all nodes of this worker

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

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

        $n.config \leftarrow trial$

      end if

    end for

▷ Recompute workflow timing with predictions

    $cp\_time \leftarrow \textsc{SimulateCriticalPathTime}(dag)$

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

▷ Downgrade acceptable, record as last acceptable state

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

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

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

        end if

      end for

    else

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

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

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

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

        end if

      end for

      break ▷ move to next worker

    end if

  end for

end for

3.4.3 Optimizations

Aside from `worker_id` and resource assignments, planners can also apply different **optimizations** to further improve the workflow execution. The optimizations a planner can use are selected by the user, as shown in Listing 3.2. We provide two optimizations: **pre-warm** and **pre-load**, but it is also possible to create new optimizations and define how workers should react to them. Now, we will describe the two base optimizations and how they are assigned to tasks:

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

   - *Interpretation*: Tasks/Nodes with this optimization should perform a special invocation to the FaaS gateway that forces it to launch a new worker with the specified resource configuration `worker_config`. This can be used to warm up workers ahead of time and mask cold start latencies.
   - *Assignment Logic*: For workers that are predicted to experience a cold start, the algorithm identifies an optimal worker to perform the pre-warming action. It searches for a task whose execution window aligns with a calculated pre-warming interval. This process balances two objectives: ensuring the pre-warmed worker does not go cold before it is required, while also not being warmed too late to be effective.
   - *Integration with Task Handling Logic*: The optimization is attached to the identified optimal worker's task and includes a `delay_s` parameter. As soon as this optimal worker begins its own execution, it launches a concurrent Python coroutine. This coroutine waits for the specified delay before making a special "empty invocation" to the FaaS gateway, which in turn initializes the target worker.

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

   - *Interpretation*: Workers assigned to a task with this optimization will proactively download the task's dependencies as soon as they become available. This is achieved by subscribing to completion notifications from the *Metadata Storage* for the task's upstream dependencies. When an upstream task finishes, the worker is notified and can immediately start downloading the result in the background, parallel to other ongoing computations. This strategy aims to reduce data-fetching latency when the task is finally ready to execute, as its inputs may already be present locally;
   - *Assignment Logic*: The optimization is applied using a simple, single-pass heuristic. The logic iterates through the tasks and assigns the `pre-load` optimization to tasks that depend on at least two upstream tasks that are assigned to a different worker;

- *Integration with Task Handling Logic*: Before a worker starts fetching a task dependencies, it prevents any new `pre-loads` from starting, and gathers a list of all ongoing `pre-loads`. It then fetches all other dependencies from storage and waits for all preloads to complete before executing the task.

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

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

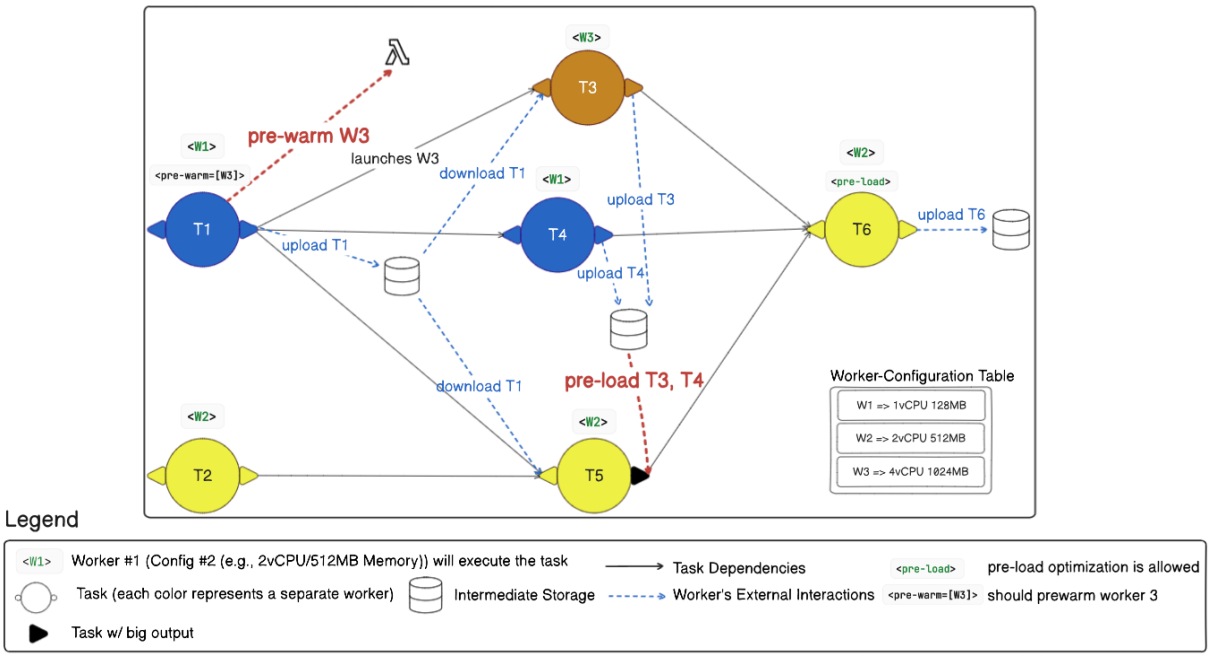

Figure 3.3: Planned Workflow Execution Example

To illustrate this execution model, Figure 3.3 provides a visual trace of how a planned workflow would be executed. The diagram depicts the workflow with the optimizations and `worker_id` assignments for each task. The non-dashed arrows represent task dependencies, while the dashed arrows represent interactions with the *Intermediate Storage* to either upload or download task data. We can see that task outputs are only uploaded to storage when there is at least one downstream task that depends on it and

is assigned to another worker.

It is also worth noting that the planner assigned `Task 6` to `Worker 2`. This decision might be due to `Worker 2` being more powerful than `Worker 1`, and because the output of `Task 5` is larger than that of `Tasks 4` and `5`. Therefore, even if the task were executed on a more powerful worker (such as `Worker 3`, which handled `Task 3`), the potential performance gain would not offset the additional time or resources required. This is an example of a planner deciding to co-locate `Tasks 5 and 6` on the same worker to reduce data movement.

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

Optimizations are implemented by re-defining worker execution logic at certain stages. The methods that a planner can implement in order to override default worker execution logic are shown in Listing 3.4: `wel_on_worker_ready` method is called once the worker starts executing, and the provided optimizations use it to subscribe to `TASK_READY` and `TASK_COMPLETED` events for specific tasks, and schedule *pre-warm* coroutines; `wel_before_task_handling` is called before task's dependencies are downloaded; `wel_override_handle_inputs` is called after discovering which tasks dependencies are not yet present locally and need fetching; `wel_before_task_execution` is called after all dependencies have been gathered and right before task's execution. `wel_override_should_upload_output` is called before task's execution and its output determines whether the task output should be uploaded to storage; `wel_update_dependency_counters` is called after a task's output is handled and the default behavior updates the *Dependency Counters* (explained in the next Section 3.6) of the downstream tasks; lastly, `wel_override_handle_downstream` is called, and the objective of the default implementation is to decide which of the downstream tasks that became ready should this worker execute, and which tasks should be delegated to other workers.

Listing 3.4: Overridable worker execution logic stages

```
1    async def wel_on_worker_ready(planner, storage, dag, this_worker)
2
3    async def wel_before_task_handling(planner, storage, dag, this_worker,
         current_task)
4
5    async def wel_override_handle_inputs(
6        planner, storage, dag, this_worker, current_task,
```

```
            upstream_tasks_without_cached_results, worker_resource_config,
            task_dependencies: dict
    ) -> tuple[list, list[str], CoroutineType | None] | None

    async def wel_before_task_execution(planner, storage, dag, this_worker,
        current_task)

    async def wel_override_should_upload_output(
        planner, storage, dag, this_worker, current_task
    ) -> bool | None

    async def wel_update_dependency_counters(
        planner, storage, dag, this_worker, current_task
    ) -> list | None

    async def wel_override_handle_downstream(
        planner, storage, dag, this_worker, current_task, downstream_tasks_ready
    ) -> list | None
```

After a plan is created and validated, the workflow representation is serialized using *cloudpickle* (as mentioned before) and uploaded to storage. The plan is embedded directly within this representation, allowing each worker to identify both the tasks it is responsible for executing and the ones it must delegate to other workers. When a plan is created, visual plan data is stored on the client machine, as illustrated in Figure 3.4, to help the user interpret scheduling decisions and identify potential inefficiencies. In this example, the visualization highlights the critical path through tasks outlined in bold, along with worker assignments, resource configurations (color-coded), applied optimizations, and predictive metrics such as earliest start time, estimated completion time, and I/O characteristics. The visualization also indicates each worker's state (warm or cold), providing an intuitive overview of the planned execution, which was very useful while developing and testing the system.

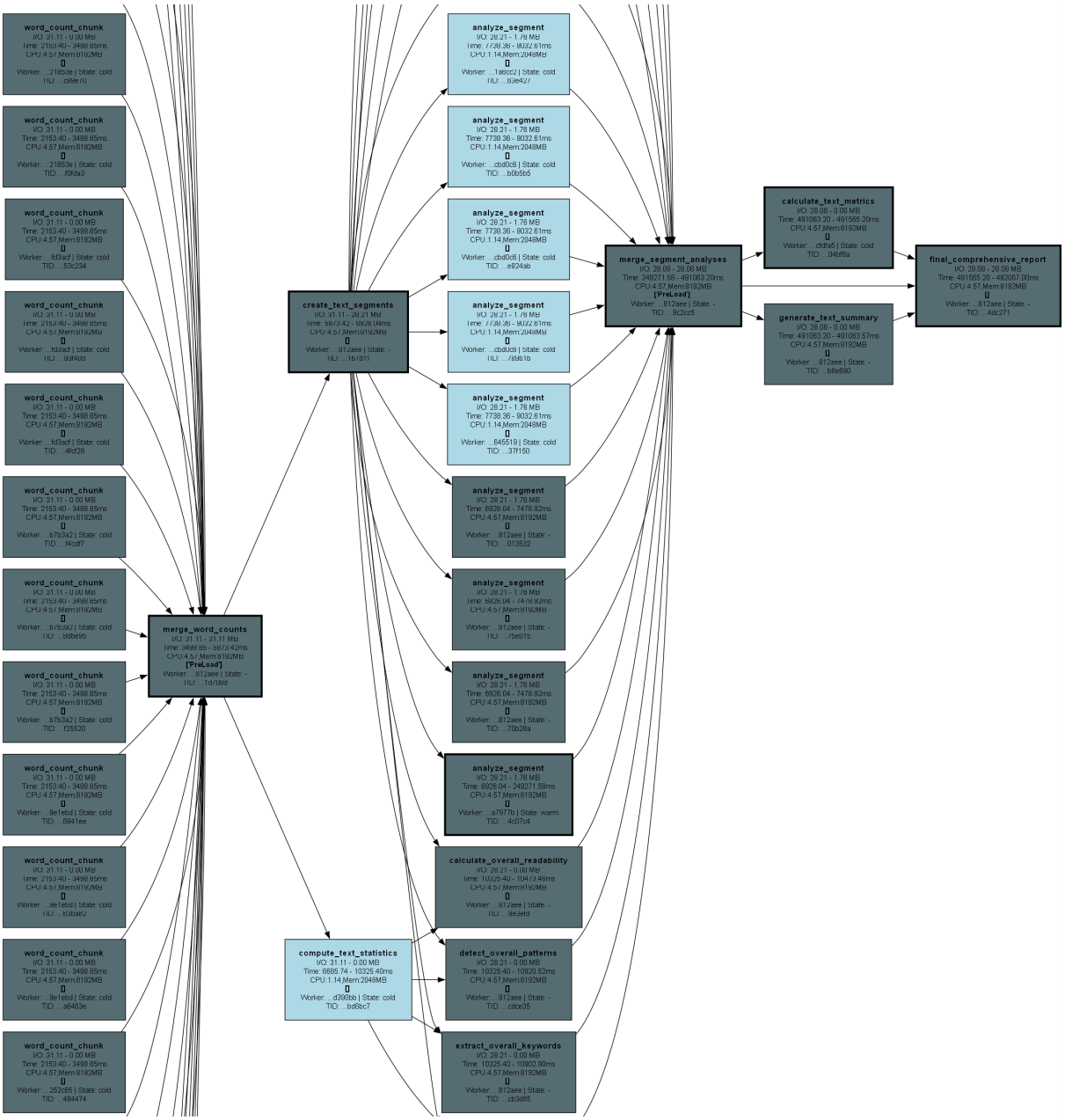

Figure 3.4: Real Workflow Plan Visualization

Now, we will explain the process of submitting the workflow and initiating its execution.

3.5 Client

In this section we will cover the role of the client application, which is where the workflow tasks are created and composed, as well as where the workflow execution is parametrized and initiated.

The `.compute(configuration)` method, which can be called on any `DAGTaskNode` reference, re-

ceives the configuration object, which contains information on how to connect to both *Intermediate Storage* and *Metadata Storage*, as well as the planner to be used, along with its configuration. The planner configuration includes the **SLA**, which will be used for the predictions made by the `Predictions API` (described in Section 3.3), the **resource configurations** that the planner is allowed to assign workers, and the list of **optimizations** that the planner should try to apply.

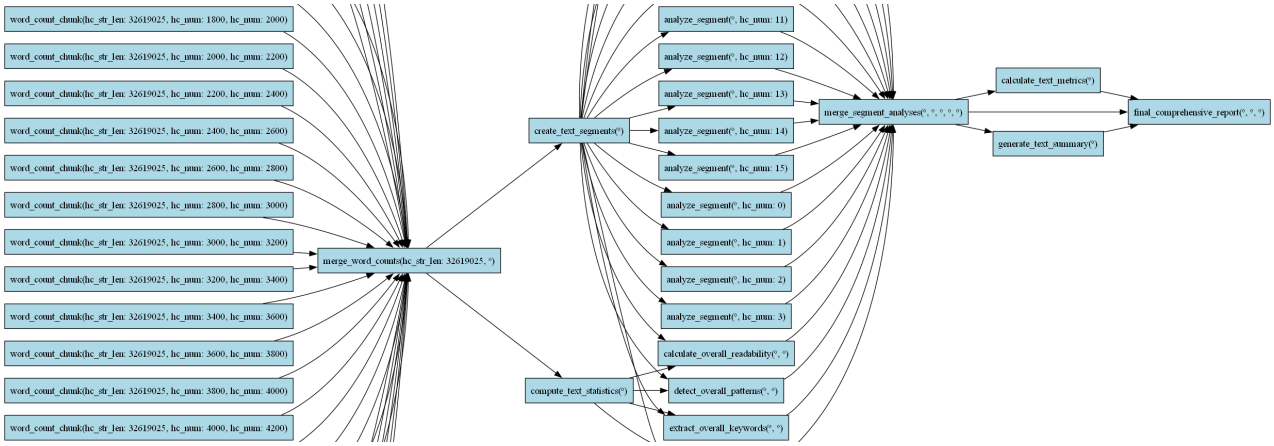

Figure 3.5: DAG Visualization Feature for a Text Analysis Workflow

Before the client calls `compute()`, it can first generate a visualization of the DAG structure. Figure 3.5 shows an example of what this visualization looks like for a Text Analysis workflow. Nodes in the graph are connected to its dependencies and each node contains the task name and its hardcoded arguments. This feature allows users to inspect and verify the workflow's dependencies and overall structure in advance, before submitting it for execution.

When the client calls `compute()`, there are a couple of steps before the initial serverless workers are invoked. First, since `compute` is called on the sink node of the workflow, we discover the entire workflow by backtracking until we find the root nodes. Then, some basic validation happens to check if there are no other nodes without downstream nodes, ensuring there's only one sink node. At this stage, a few optimizations are made in order to reduce the size of the workflow representation. This is important because if the workflow representation is below a threshold we can send it in the invocation to FaaS workers without having to rely on storage, saving on both performance and storage costs.

One of these optimizations involves special handling of large hardcoded inputs. When a task receives a large input that is already known before workflow execution begins, this input would normally be embedded directly within the `DAGTaskNode` representation. However, because workers exchange this representation among themselves, embedding large inputs would lead to unnecessary data transfers and inefficiency. To address this, we detect all large hardcoded input objects and replace them with unique references. During execution, workers can use these references to retrieve the corresponding inputs from storage only when needed. The client is responsible for uploading these large input objects

to *Intermediate Storage* prior to workflow execution.

Before uploading hardcoded data, the client first loads all relevant metrics required for workflow prediction and then invokes the selected planner algorithm. This process updates the workflow representation with worker assignment information and optimization details. As mentioned earlier, predictions are based on samples collected from the same type of workflow and planner to improve their accuracy. To determine which samples correspond to which workflow, we differentiate workflows by appending a *hash* of their graph composition, defined by the task names and their dependency relationships, to the workflow's unique identifier. Workflow identifiers sharing the same hash are thus considered to represent the same workflow type. After a plan is created, an image of the predicted metrics, worker assignments and optimizations is generated and stored on the client machine, which can help the user to better understand the plan created by the planner.

After a plan is created and embedded in the DAG representation, the client invokes the workers assigned to the root nodes. It provides each worker with the corresponding task identifiers, enabling them to begin executing their respective branches of the workflow. Afterward, the client uses Pub/Sub to subscribe to the `TASK_COMPLETED` event associated with the workflow's final task. Once this event is received, the client retrieves the result from *Intermediate Storage* and returns it to the user. From the user's perspective, invoking `compute()` behaves like calling a regular function, returning the expected result, while the actual execution is transparently handled by serverless workers.

During workflow execution, the client can monitor the workflow's progress through a simple web dashboard, showcased in Figure 3.6, implemented with Streamlit [4]. This dashboard can be used to visualize the workflow graph in real-time, as it indicates which tasks have been completed (in green) and which remain pending (in gray).

Figure 3.6: Workflow Real-time Dashboard

[4] https://streamlit.io/

In the following section, we describe in detail how workers operate, including how they follow the plan, handle task dependencies, and coordinate through event-based communication to achieve decentralized workflow execution.

3.6 Decentralized Workers

Since our target execution platform is FaaS, each worker is implemented as a FaaS handler. Owing to the decentralized design, workers are responsible not only for executing tasks but also for coordinating scheduling decisions in a choreographed manner. Upon invocation by the FaaS gateway, a worker receives a **configuration** containing connection details for storage and the gateway, along with SLA parameters and related metadata. It also receives a list of **immediate task identifiers** that define the workflow branches it should manage (executing its own tasks and delegating the rest to other workers) as well as the corresponding **upstream outputs**, which may be complete or partial depending on whether the worker that spawned this worker already had this data available locally or not. Finally, the invocation includes either the **workflow representation** itself or a **workflow instance identifier** used to fetch it from *Metadata Storage* if it exceeds a size threshold (300KB in our setup).

In this initial stage, the worker calls the `wel_on_worker_ready` method on all optimizations. Then, it subscribes to `TASK_READY` events for all tasks assigned to it that depend on tasks assigned to other workers. To begin executing tasks, the worker launches a Python *asyncio* [5] *coroutine* for each branch, starting from the provided **immediate task identifiers**. From this point, the main *coroutine* simply waits for all assigned tasks to complete, and then runs the cleanup logic, which deletes intermediate data from storage, flushes metrics and metadata to *Metadata Storage*, and closes storage connections.

Each branch of task handling runs concurrently in its own *coroutine*, and follows these sequential phases when running without optimizations:

1. **Gathering Dependencies**: Check which dependencies are missing (not downloaded yet) and download them from storage. Some dependencies might be present in the local representation of the DAG if `pre-loading` happened, if the same worker executed some of the upstream tasks, or if the worker already downloaded large hardcoded data (explained previously in Section 3.5) because previous tasks needed it;

2. **Executing Task**: After gathering all task dependencies (upstream task outputs and hardcoded data), it executes the task's code. Its function code is embedded within the workflow representation so it can be easily executed by the workers, similarly to WUKONG. This enables the worker to remain generic, capable of receiving and executing arbitrary task code (excluding async functions, since *cloudpickle* can't serialize them). Tasks' code execute on a separate thread, to avoid slowing

[5] https://docs.python.org/3/library/asyncio.html

down or even blocking (e.g., if the task code calls `time.sleep()`) the main thread, where the other coroutines are running: both task branch execution and other internal coroutines (i.e., Redis pub/sub listener, and delayed `pre-warm` requests);

3. **Handling Output**: This phase is responsible for evaluating whether it's necessary to upload the task's output to storage and emitting a `TASK_COMPLETED` event. A tasks' output is uploaded if there is at least one downstream task assigned to a different worker, or if it's the sink/last task of the workflow;

4. **Updating Dependency Counters**: For each downstream task, the worker performs an *atomic increment and get* operation on a *"Dependency Counter"* (inspired by WUKONG [1]) stored in *Metadata Storage*, which tracks how many dependencies of a task have been satisfied. If the counter sees that the value returned is the same as the number of dependencies for a downstream task, the worker emits a `TASK_READY` event for that task, signaling other workers or workers that aren't active yet that the task became ready to execute;

5. **Delegating Downstream Tasks**: After updating dependency counters, the worker knows which tasks became *ready*. The worker then consults the execution plan to determine how to proceed for each of those tasks: for tasks assigned to the same worker and has no remaining unfulfilled dependencies, the worker will execute it (one *coroutine* for each task); for tasks assigned to another worker **that is already active**, a `TASK_READY` event is emitted; for tasks assigned to another worker **that is not active**, this worker launches the target worker, indicating the branches (*immediate task identifiers*) it should execute. Because of the planner validation mentioned in Section 3.4.2, it is possible for a worker to know, at any point in time, whether another worker is already active or not just by looking at the workflow representation and respective plan.

For exchanging events among workers, we use Redis's Pub/Sub [6]. Both *Intermediate Storage* and *Metadata Storage* are also implemented in Redis for deployment simplicity. These events are `TASK_READY` and `TASK_COMPLETED`, and are implemented as Pub/Sub channels, where the channel name is a composition of `"notification-task-ready-"` and the unique Task ID. Since the same worker can subscribe to the same event multiple times, we assign unique subscription identifiers that are local to each worker. This allows the worker to unsubscribe from individual event subscriptions safely, without inadvertently removing other listeners that might depend on the same event.

We will now go over a practical example of how we use these events are used to coordinate decentralized scheduling. Figure 3.7 presents an example of a workflow with four workers (A, B, C, and D) and eight tasks (T1-T8).

[6] https://redis.io/glossary/pub-sub/

In this example, tasks complete in the following order: T1, T2, T3, T4, T5, T6, T7, and T8. The text in the arrows represent the order of actions performed by the worker who executed the task from which the arrow starts, where each action is separated by a semicolon. When T1 finishes, its worker will increment the dependency counter of T5 because it knows, by looking at the workflow representation, that T5 depends on other tasks from another worker. This storage operation returns "1", but T5 has 2 dependencies (T1 and T2) and as such, it isn't READY to execute. Then T1's worker sees T4 and since it's assigned to it, and it doesn't have any other dependencies, it schedules it for execution locally. Lastly T1 will see that T3 doesn't depend on any other tasks, meaning it is also ready for execution and, since it is assigned to another worker (Worker C), it sends a request to the FaaS gateway to launch Worker C, providing the branches it should execute.

Then, when T3 finishes execution, its worker will see that T7 depends on external tasks, so it updates the dependency counter of T7 and remains idle waiting for the `TASK_READY` event for T7. Later, T4 finishes, realizes that T7 is meant to execute on another worker, increments its dependency counter, realizes all dependencies are met, and emits a `TASK_READY` event for T7, to which Worker C will react to by start executing T7. Once T5 finishes execution, it realizes that T6 depends on external tasks, increments its dependency counter, realizes all dependencies are met, and since it's assigned to itself, it schedules it for execution locally.

Finally, T6 finishes, increments the dependency counter of T8 (because it has other dependencies), but since it's still missing 1 dependency, it exits. Later, then T7 finishes, it increments the dependency counter of T8 and realizes all dependencies are met and that the worker assigned to T8 (Worker D) isn't active yet, so it makes a request to the FaaS gateway to invoke this worker. Once T8 finishes, it emits a `TASK_COMPLETED` event which is received by the client, who then downloads the result from *Intermediate Storage*.

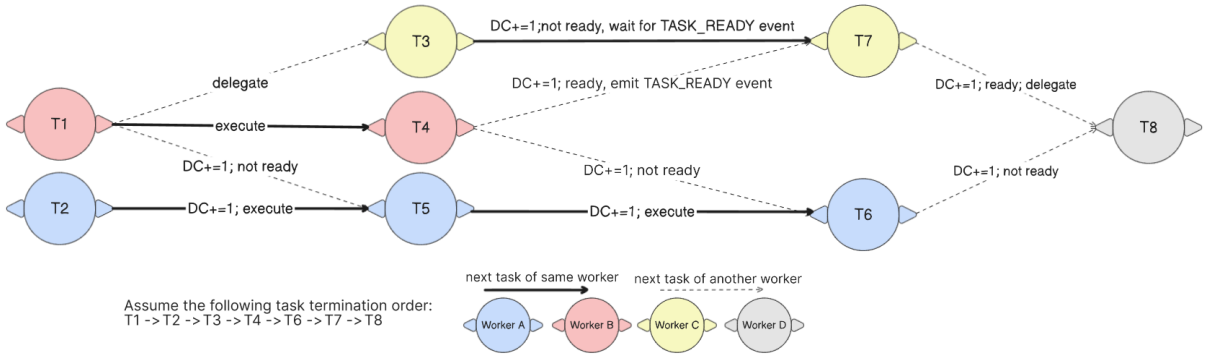

Figure 3.7: Choreographed Scheduling Example

A workflow is considered complete once the output of the final (sink) task is available in storage. The worker that uploads this final result is also responsible for cleaning up all intermediate results before

shutting down. This worker emits a `TASK_COMPLETED` event for the sink task, triggering the client to retrieve the final result from *Intermediate Storage*.

By delegating downstream tasks to workers, our approach eliminates the need for a central scheduler, a common component in many existing FaaS-based workflow engines. This decentralized approach presents several key advantages over traditional centralized scheduling systems. By allowing each worker to dynamically schedule downstream tasks, the system avoids the need for a continuously active central scheduler during workflow execution, reducing coordination overhead and eliminating the potential bottleneck of a single point of failure. Additionally, workers can make scheduling decisions immediately after completing their tasks, minimizing additional round-trip delays that would occur if a central scheduler were responsible for assigning subsequent tasks. This model also improves scalability, which mostly depends on the capability of both the FaaS platform and storage to scale horizontally. However, this design relies on the client to initiate the first set of workers, requires workers to have knowledge of the workflow structure to properly follow the scheduling plan (including optimizations), and still involves a minimal level of coordination among workers to ensure that every task in the workflow is eventually executed.

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

4

Evaluation

Contents

4.1 Emulating FaaS environment 47
4.2 Research Questions 49
4.3 Experimental Setup 51
4.4 Results 53
4.4.1 Predictions Accuracy and SLA fulfillment 53
4.4.2 Overall Performance 54
4.4.3 Optimizations 56
4.4.4 Resource Usage 57
4.5 Analysis 61

4.1 Emulating FaaS environment

To develop, test and evaluate the proposed solution under realistic and controlled execution conditions, a simple and lightweight Function-as-a-Service (FaaS) platform was developed in ~300 LOC of Python.

This setup reproduces the fundamental behavior of a typical serverless platform, such as those mentioned in Chapter 2, while remaining simple and fully controllable for testing purposes. This emulation consists of two main components: a **gateway service** implemented as an HTTP server that receives invocations and manages containers, dynamically provisioning and reusing Docker [1] containers with the requested resource configuration to execute arbitrary code, and the **worker code** itself, which is packaged into a docker image.

The Dockerfile of the worker is packaged with the worker logic code, and it simply remains alive forever, running a simple bash script that prevents the container from exiting. The gateway is then able to launch containers with this image and invoke the worker code at any time to execute the desired tasks. Both the client and the workers can make requests to a `/job` endpoint to launch new workers, as shown in Listing 4.1, where the caller specifies the required resource configuration for the workers' container. If there are no idle containers with the desired resource configuration and the maximum amount of concurrent containers has been reached, this request will wait until there are conditions to run the worker code. To enforce resource configurations, we use Docker's built-in support for CPU and memory limits [2].

Listing 4.1: Example of Delegating Tasks to Gateway

```
async with await session.post(
    gateway_address + "/job",
    data=json.dumps({
        "resource_configuration": ...,
        "dag_id": ...,
        "fulldag": optimized_dag if optimized_dag and fulldag_size_below_threshold else None,
        "task_ids": [...],
        "relevant_cached_results": {...},
        "config": ...,
    })
)
```

To provide isolation and avoid resource contention, each Docker container can only run one instance of the worker code at a time. This is enforced using a file-based locking mechanism that prevents concurrent execution within the same container, ensuring exclusive access to container resources and maintaining predictable performance characteristics. However, since the Gateway keeps track of which containers exist, which are being used, and which are idle, it shouldn't invoke worker commands on a container that is already executing another command.

The Gateway limits the maximum worker concurrency to 32, meaning there can only be 32 active

[1] https://www.docker.com/
[2] https://docs.docker.com/engine/containers/resource_constraints/

workers at a time. This is because this FaaS platform emulator is designed to run on a single host, but this value can be increased on more capable hosts. This is also a simple implementation, but it would be possible to implement a distributed version by using something like Kubernetes [48] that would provide a more realistic environment. The Gateway also has a background job that removes containers that are idle for more than a configurable amount of time (we used 7 seconds in our experiments because it allowed to test *pre-warming* without making workflows too long). A container is considered idle once no worker code is running on it.

For simplicity, the Gateway also contains an HTTP endpoint for *pre-warming* (`/warmup`), which receives a worker configuration list and creates one container for each resource configuration specified without executing any commands on it, leaving them available for future `/job` invocations to use. Similarly to other containers, these containers are also subject to the idle timeout and will be removed if they are idle for too long.

To improve observability and allow for easier debugging, the worker standard output and standard error are streamed to the Gateway process, where they are captured and logged in real time. This design enables detailed inspection of function execution without accessing the containers directly.

The worker code packaged in the Docker image is designed to work specifically with this simplified FaaS emulator and implementing worker logic for platforms like AWS Lambda [2] or OpenFaaS [24] would require implementing a `Worker` abstract class, defining how to delegate tasks to the FaaS platform, as well as re-write parts of the worker code logic. This would include grabbing invocation data differently (from HTTP request instead of process standard input). Detecting whether the worker code is running on a cold or warm container would also be different. Instead of using a file-system file, as our emulator does, we would simply use a global variable, since their values are persisted across different invocations on the same container/underlying runtime. The remaining worker logic should work as intended.

4.2 Research Questions

With the evaluation of this work, we aim to address the following research questions:

- **RQ1:** To what extent can historical metrics from previous workflow executions improve the accuracy of predicting serverless workflow behavior, specifically regarding execution time, container startup latency, data transfer performance, and function I/O characteristics?
- **RQ2:** What are the main advantages and limitations of applying prediction-based scheduling strategies in serverless workflow environments, compared to traditional reactive or static scheduling approaches? Can the proposed system achieve a lower *makespan* and reduced overall resource consumption compared to WUKONG [1], a decentralized serverless workflow engine that employs its own set of data-locality optimizations?

- **RQ3:** How effective are the proposed optimizations in practice? In particular, how much does `pre-load` contribute to hiding latency and enabling earlier task execution, and how beneficial is `pre-warming` in reducing cold-start delays and improving overall workflow performance?
- **RQ4:** How much performance improvement can be achieved by adopting a non-uniform worker resource allocation strategy, compared to a uniform scheduling approach that assigns identical resource configurations to all tasks? And what is the trade-off between the achieved performance gains and the additional resource consumption introduced by this adaptive allocation strategy?

To help answer these questions we had to collect a wide range of metrics:

- For each **task**:
  - Timestamp of when it started being handled (before executing its tasks);
  - Time to download all dependencies;
  - Size and time to download each of the dependencies;
  - Task execution time;
  - Size and time to upload output;
  - Optimization-specific metrics of all optimizations applied to a given task on a particular workflow instance.
- For each **workflow instance**:
  - Entire workflow plan, with worker resource and ID assignments, as well as optimizations;
  - Time at which user submitted the workflow;
  - Monitor Docker containers during the workflow, recording total run-time and memory allocation in GB-seconds, similar to AWS Lambda's cost calculation, which is based on memory (GB) * run-time (seconds).
- For each **worker**:
  - Resource Configuration (CPUs and Memory);
  - Time at which a worker invocation was made to the FaaS Gateway and time at which the worker script started executing (used to calculate worker startup latency);
  - Worker state, indicating whether the worker script was executed in a *cold* or *warm* environment. A cold start occurs when a new container must be created to run the worker, while a warm start reuses an existing, previously initialized container. The system determines this state using a file-based locking mechanism: during startup, the worker attempts an atomic

"create-if-not-exists" operation on the file `/tmp/worker_startup.atomic`. If the file already exists, it implies that the container has been used before, and the current execution is therefore a warm start; otherwise, it is a cold start. When a container is *pre-warmed*, this file is proactively created during initialization, even though the worker script itself is not yet executed.

In the following section, we describe the experimental setup used to evaluate the proposed system.

4.3 Experimental Setup

To evaluate our proposed solution, we deployed the FaaS emulator described in Section 4.1 on an AMD EPYC 7282 16-Core (32 logical threads, via AMD SMT) processor with 125GiB of RAM, running Ubuntu 22.04.5 LTS. Both *Metadata Storage* and *Intermediate Storage* were deployed as Redis Docker containers. The client, responsible for submitting the workflow and uploading hardcoded dependencies, was also run on the same machine. We added some artificial delay in both storage and gateway interactions, to try simulating a more realistic environment. This delay is applied before requests are made and the value we used was 30ms of round-trip time. This value was chosen based on observations of median latency between AWS Europe data centers in the same region being around 8ms and around 15ms across different data centers [3].

We tested four different workflows, three SLAs, six planners, and two optimization strategies. To limit the number of experimental combinations, the optimizations were not evaluated in isolation; instead, both *pre-load* and *pre-warm* optimizations were applied together.

The planners that used uniform worker resource configurations (*Uniform*, *Uniform w/ optimizations*, *WUKONG*, and *WUKONG w/ optimizations*) used 2GB memory and the vCPUs were calculated similarly to AWS Lambda, proportionally to the function's memory size [4] (*1 vCPU per 1,769 MB*).

The workflows used for evaluating our solution were the following:

- **Matrix Multiplication**: A workflow with a straightforward structure, shown in Figure A.1 (partially), that performs distributed matrix multiplication. The input matrices are first divided into smaller chunks, and each pair of compatible chunks is multiplied independently in parallel. The resulting partial products are then aggregated by a final task, which reconstructs the full matrix from the computed blocks, producing the complete multiplication result. WUKONG's evaluation also includes a comparable workflow that follows the same computational pattern;
- **Tree Reduction**: A workflow that performs a hierarchical reduction over a list of numeric values, depicted in Figure A.2 (partially). In each stage, pairs of elements are summed in parallel, forming

[3] https://www.cloudping.co/
[4] https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html

a binary reduction tree where the number of active tasks halves at each level. This process continues recursively until a single task (the sink) produces the final aggregated result. This workflow is useful for evaluating how well the system handles iterative task dependencies and parallel aggregation. WUKONG's evaluation includes an identical workflow, featuring the exact same structure;

- **Text Analysis**: A workflow with a more complex structure, shown in Figure A.3 (partially), designed to simulate a multi-stage text processing pipeline, combining both fan-out and fan-in patterns. It includes several interdependent analytical tasks that operate on an input text (a 750,000 lines text file). This workflow aims to highlight scenarios where our decentralized execution model can better exploit task parallelism and reduce scheduling overhead compared to WUKONG;
- **Image Transformation**: A complex, highly parallel workflow that applies multiple transformations to an input image, shown in Figure A.4 (partially). The image is first divided into chunks, each processed independently through two parallel transformation branches of differing complexity: one performing color-based operations such as resizing, blurring, normalization, and sepia filtering, and another applying edge detection and sharpening. The results from both branches are then combined and merged back into a complete image. This workflow is bigger, composed of 130 tasks large fan-outs and heterogeneous task execution times, making it ideal for evaluating our solutions' efficiency. We also expect better performance compared to WUKONG, as our scheduling model requires fewer worker launches and thus less downstream coordination overhead.

As mentioned in Section 3.3, the client specifies an SLA which determines how conservative the predictions are. We ran our experiments with three different SLAs: median (50th percentile), 75th percentile, and 90th percentile. We will then analyze the fulfillment success rates of each SLA, the prediction's accuracy and whether different SLAs yield different performance and resource usage results.

To assess our research questions, we implemented three core planners: our proposed **Uniform** and **Non-Uniform** models, and a replication of the **WUKONG** scheduling model. We evaluated six total configurations by testing baseline versions against *optimized* versions.

Specifically, our **Uniform** and **Non-Uniform** planners were *optimized* with our proposed *pre-load* and *pre-warm* optimizations. For comparison, the **WUKONG** planner was enhanced with its own native optimizations (*Task Clustering* and *Delayed I/O*, explained in Chapter 2). The six evaluated variations were:

- **Uniform** (baseline)
- **Optimized Uniform** (with *Pre-Load* and *Pre-Warm*)
- **Non-Uniform** (baseline)
- **Optimized Non-Uniform** (with *Pre-Load* and *Pre-Warm*)

- **WUKONG** (baseline)
- **Optimized WUKONG** (with *Task Clustering* and *Delayed I/O*)

The experimental process was automated with a script that iterated through every combination of planner variation, SLA, and workflow. Each unique configuration was executed ten times to account for performance variability, resulting in 720 total experiments for analysis. Before starting a new workflow instance, this script waits for all FaaS gateway containers to become idle and then removed, causing the workers of the root nodes of the next workflow to experience cold starts. This ensures consistency in the experimental process.

4.4 Results

This section presents the evaluation results of all proposed planners, including their *optimized* ("-opt") versions. We analyze prediction accuracy, SLA fulfillment rates, overall execution metrics, and resource efficiency, while also highlighting how the two proposed optimizations impact workflow execution and cost.

4.4.1 Predictions Accuracy and SLA fulfillment

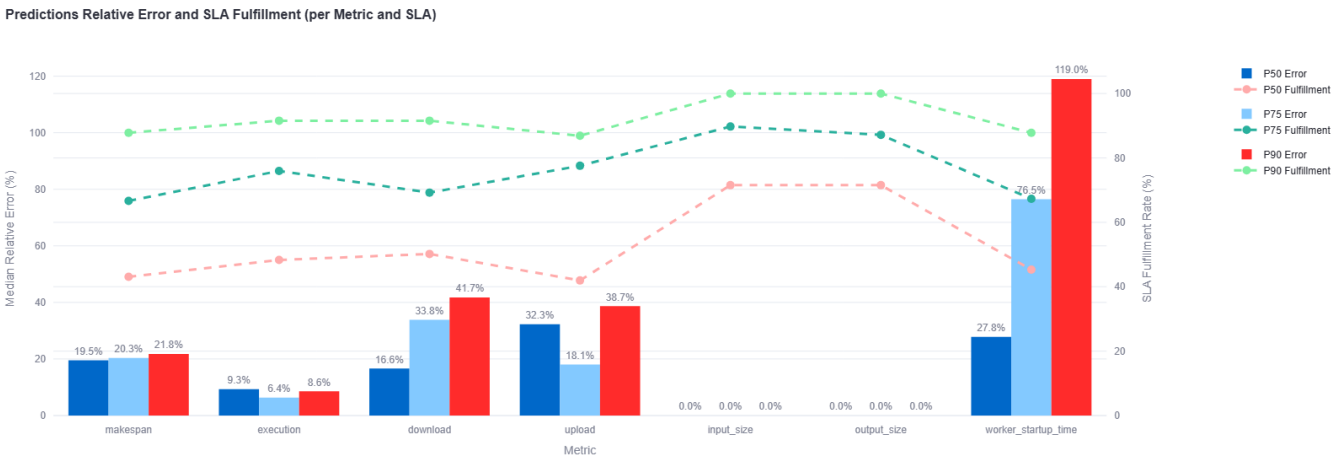

Figure 4.1: Predictions Accuracy across all planners and SLA fulfillment rates

The chart in Figure 4.1 evaluates the model's predictive performance across various metrics for all tested workflows and all planners (excluding WUKONG, which doesn't use predictions). It displays two key indicators:

- **Prediction Accuracy**: This measures the error, or deviation, between the model's predictions and the actual measured outcomes;
- **SLA Fulfillment Rate**: This represents how consistent the workflow performance was with respect to the chosen SLA.

Regarding prediction accuracy, we observe that (except for execution and upload times) **higher SLAs lead to larger prediction errors**. This outcome is expected, as higher SLA levels adopt a more conservative approach, which tends to overestimate values. The **median relative error** for input and output sizes is **0%**, since identical workloads were used throughout the experiments, making these metrics easier to predict.

Across all four workflows tested, **execution time predictions were the most accurate**, with median relative errors below **9.3%**. In contrast, **data transfer times** (upload and download) were less precise, showing median relative errors of around **35%**.

When examining **SLA fulfillment rates**, we find that they remained stable and aligned with expectations across all SLA levels. Specifically, **P50, P75, and P90 SLAs** achieved fulfillment rates in the ranges of **41.9–71.5%**, **66.7–89.7%**, and **86.9–100%**, respectively.

4.4.2 Overall Performance

The box plot shown in Figure 4.2 compares the makespan distribution between planners. All versions of our planners performed slightly better than both versions of the **WUKONG** planner when it comes to *makespan*, with the non-optimized version of our **Uniform** planner performing **12.6%** faster when compared with the optimized version of WUKONG. We can also see a clear improvement from the **Non-Uniform** to **Non-Uniform w/ optimizations** (with the best *makespan* observed), but the same doesn't happen when comparing the **Uniform** planner against it's optimized version. This could be due to the Uniform planner using less resources per worker, leading to higher **resource contention** when *pre-loading* dependencies. This could also explain higher times waiting for dependencies, in the plot shown in Figure 4.3.

In Figure 4.3, we can also observe that *pre-loading* had a significant impact on the **Non-Uniform** planner, resulting in a difference of **1.34 seconds**. A notable contrast is also evident between **WUKONG** and **WUKONG w/ optimizations**, with the latter exhibiting a substantial reduction in waiting times for dependencies. This behavior is expected, as WUKONG's optimizations (Task Clustering for Fan-ins and Fan-outs, and Delayed I/O) prioritize locality for large dependencies, consistently attempting to execute tasks where the required inputs are already available.

The chart in Figure 4.4 compares the worker startup time distribution across planners, clearly showing higher values for WUKONG. This is expected, as WUKONG's scheduling strategy launches more

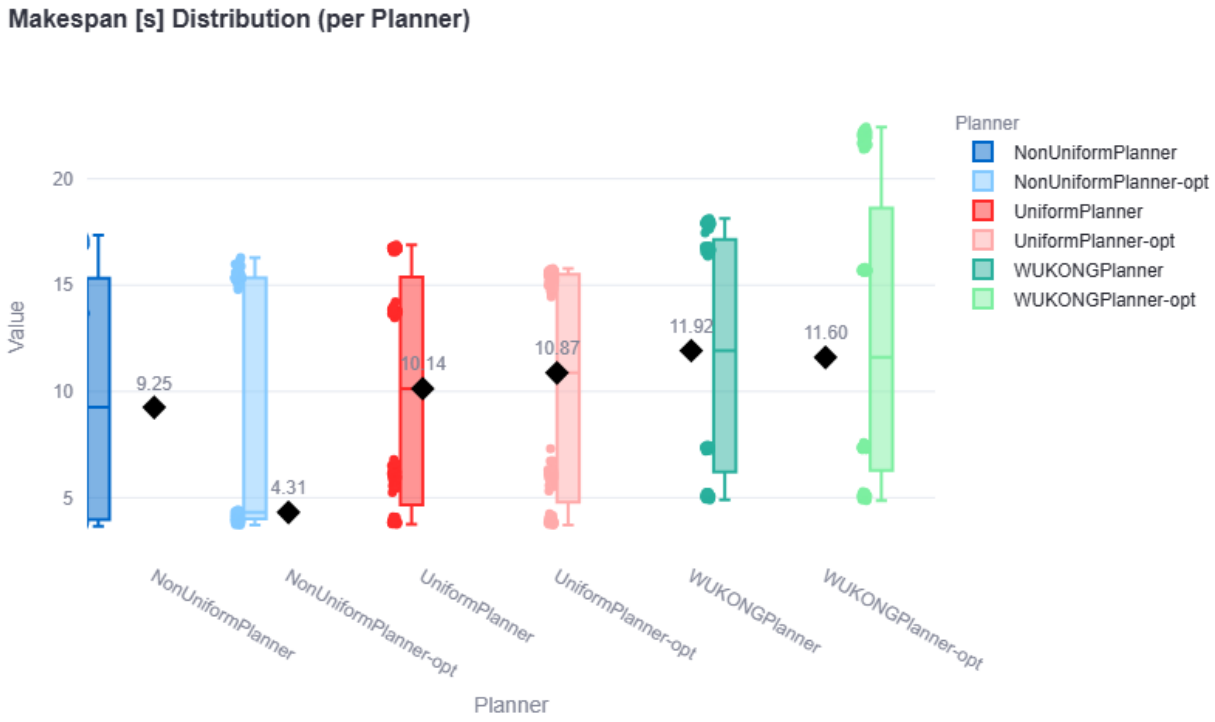

Figure 4.2: *Makespan* distribution

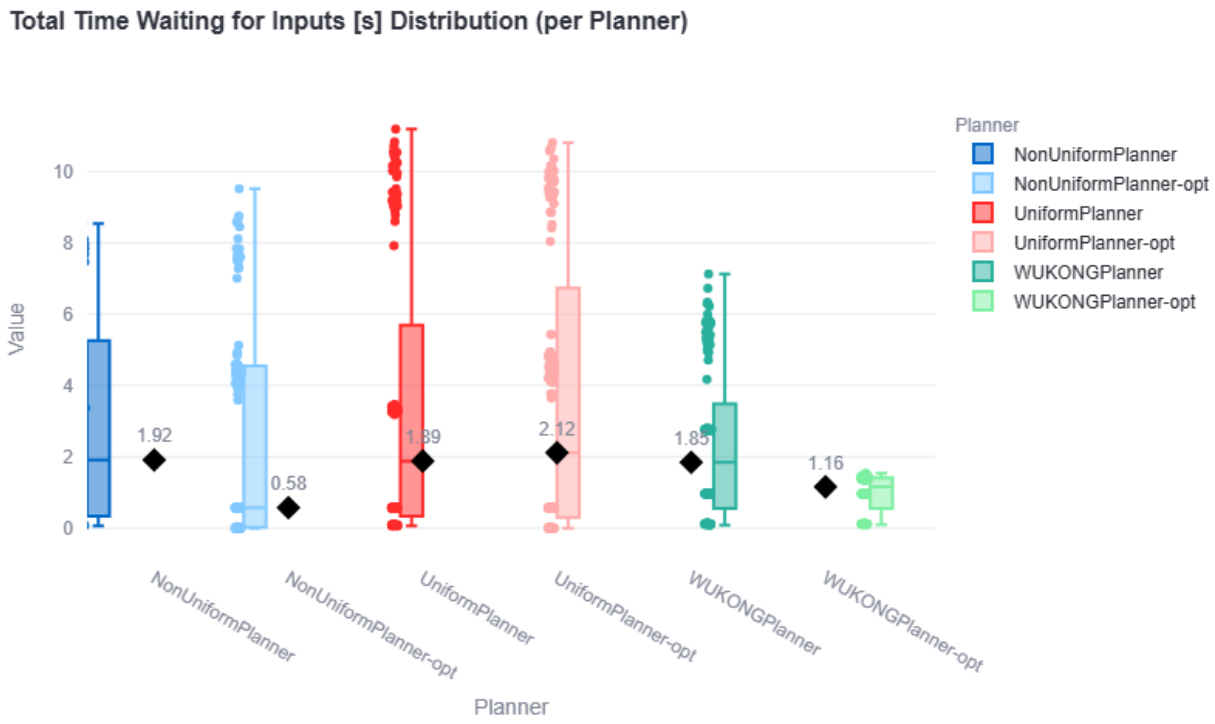

Figure 4.3: *Waiting for dependencies* distribution

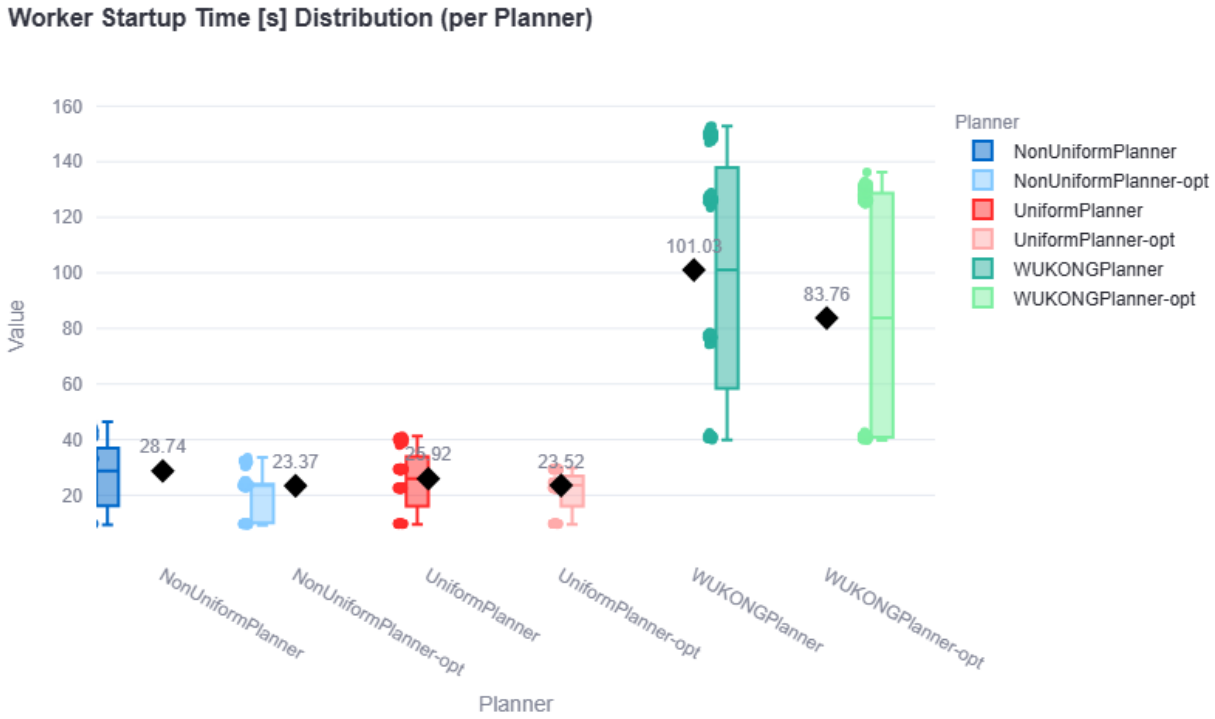

Figure 4.4: *Worker startup time* distribution

workers than our planners. For fan-outs of size `N`, WUKONG launches `N-1` workers (except in the optimized version, which attempts to avoid launching workers when the output is large). In our experiments, WUKONG launched a median of 32 workers (with optimizations, a median of 24), while our planners launched a median of only 13 workers.

4.4.3 Optimizations

The stacked bar chart in Figure 4.5 shows a time breakdown analysis between planners. It shows the median of how much time each planner spent on different tasks: waiting for workers to start, time downloading dependencies, task execution time, and time uploading results. Here, we can see that WUKONG has the highest time spent waiting for workers to start, as it launches more workers than our planners. The optimized version of WUKONG spends less time waiting for dependencies than the non-optimized version, showing that, for the tested workflows, it compensates to delay large task output uploads to try and execute downstream tasks locally.

WUKONG's execution time is slightly lower than our **Uniform** planner (~12.5s vs ~20s) which shows the slowdown caused by resource contention, as **Uniform** can execute multiple tasks in parallel, whereas WUKONG workers execute only one task at a time. Despite using more capable workers, our **Non-**

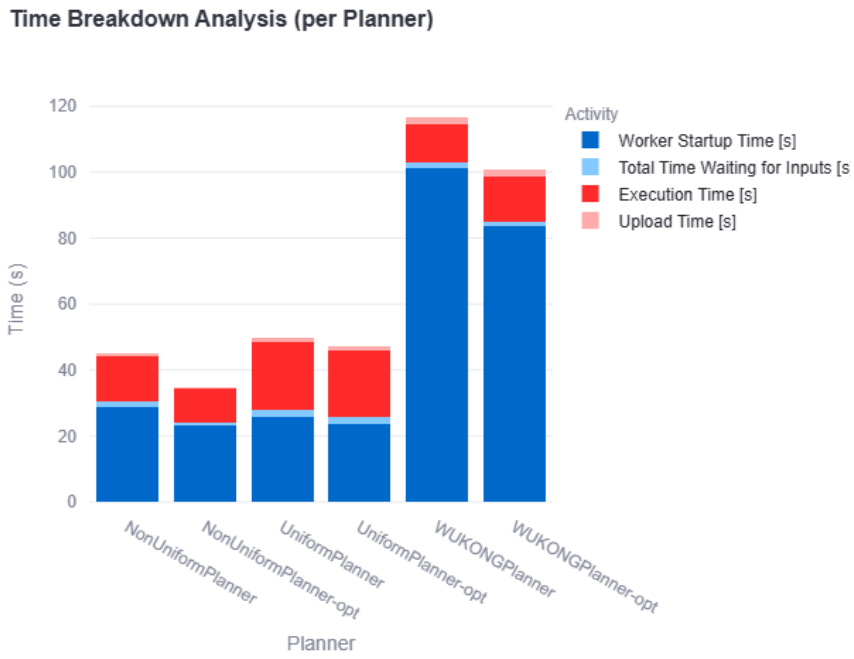

Figure 4.5: Time breakdown analysis

Uniform planner shows an execution time similar to **WUKONG**, a result that probably can also be attributed to resource contention.

Similarly to Figure 4.3, we can see the positive impact of *pre-loading* on the **Non-Uniform** planner, as well as the positive impact of *pre-warming* on both "optimized" versions of our planners, saving between **2 to 6 seconds**.

The chart in Figure 4.6 illustrates the impact of the *pre-warm* optimization, supporting the findings in Figure 4.5 regarding worker startup times. The optimized versions of our algorithms achieve, on average, **25%** more warm starts compared to their respective base versions.

4.4.4 Resource Usage

The chart in Figure 4.7 shows the resource usage distributions in *GB-seconds* (used by AWS Lambda to charge users [5]) across all planners. The optimized version of WUKONG clearly uses fewer resources than the non-optimized version, which is attributable to its reduced worker count. Once again, we believe that the reason for the optimized version of the **Uniform** planner to be using more resources than its base version is due to resource contention, which increases task execution time, consequently increasing overall resource usage. Despite this, if we compare the baseline **Uniform** planner implementation, it consumes **36%** less resources than the optimized version of **WUKONG**.

[5] https://aws.amazon.com/lambda/pricing/

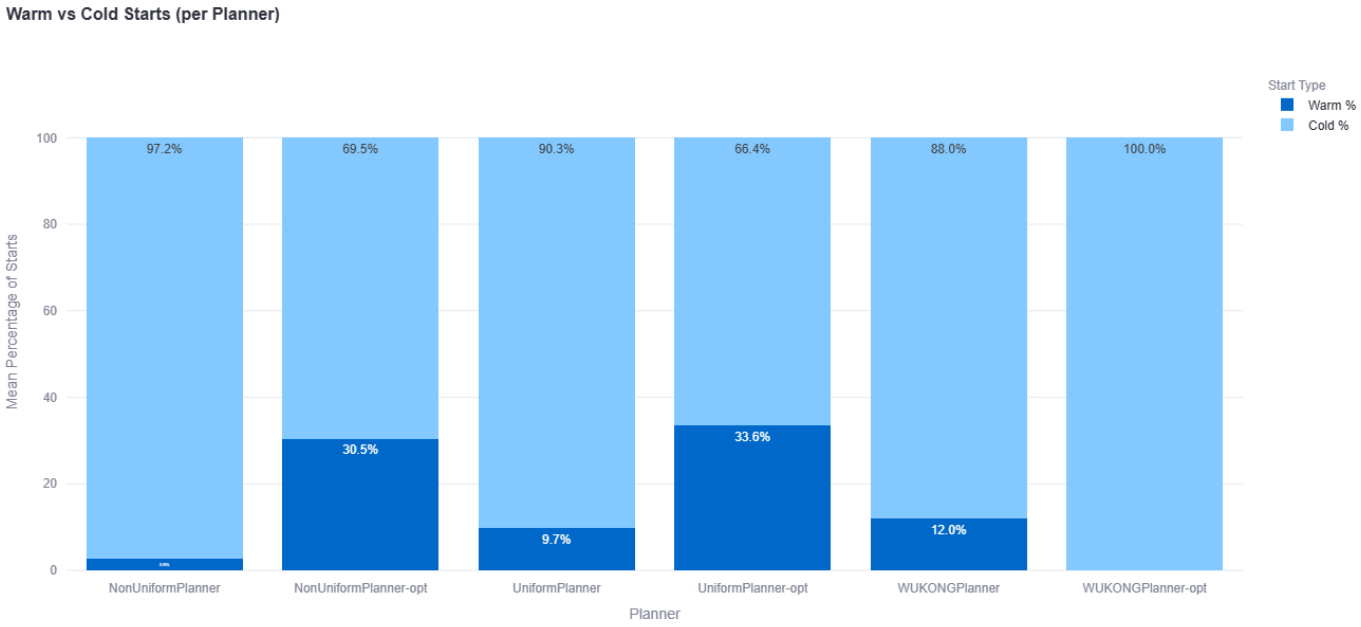

Figure 4.6: Warm vs cold starts

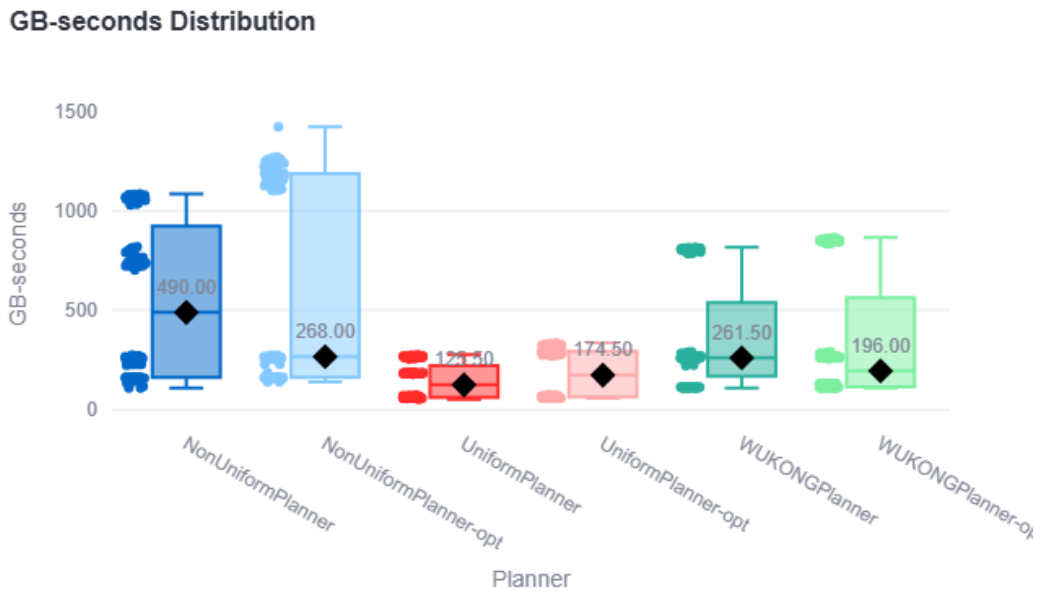

Figure 4.7: Resource usage distribution (in GB-seconds)

In contrast, the **Non-Uniform** planner shows a substantial improvement: the optimized version consumes slightly more than half the resources of the non-optimized version and nearly matches the resources used by **WUKONG**, despite using workers up to four times more powerful (2GB memory versus up to 8GB per worker). This highlights the effectiveness of the *pre-load* and *pre-warm* optimizations.

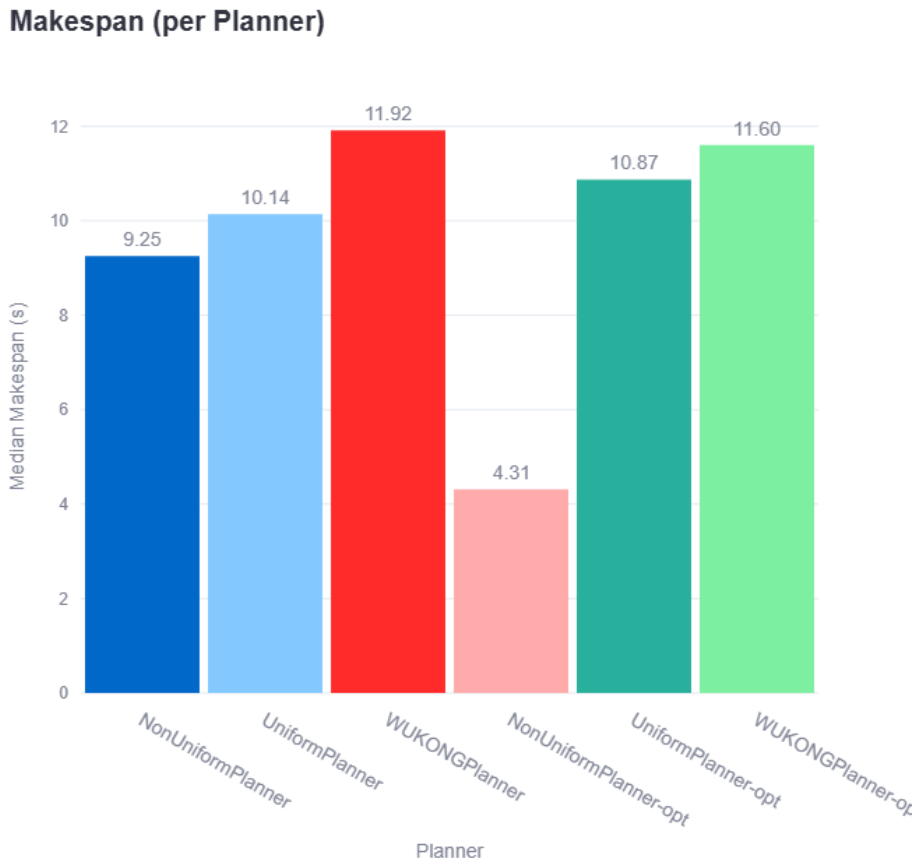

Figure 4.8: *Makespan* comparison

The charts presented in Figures 4.8 and 4.9 highlight the trade-off between workflow execution speed (makespan) and resource efficiency (GB-seconds), respectively, of the evaluated algorithms. Choosing the **Non-Uniform** planner over the **Uniform** planner yields an **8.8%** performance gain (i.e., a shorter makespan), but this speed comes at a significant cost: a **290%** increase in memory resource consumption (GB-seconds).

The **WUKONG** planner represents a less favorable option in this comparison. It is **14.9% slower** than the **Uniform** planner while also consuming **108% more** memory resources, making the **Uniform** planner the most resource-efficient of the three.

The introduction of optimized planners dramatically alters the performance landscape. The optimized **Non-Uniform** planner emerges as the clear frontrunner in execution speed, achieving a median

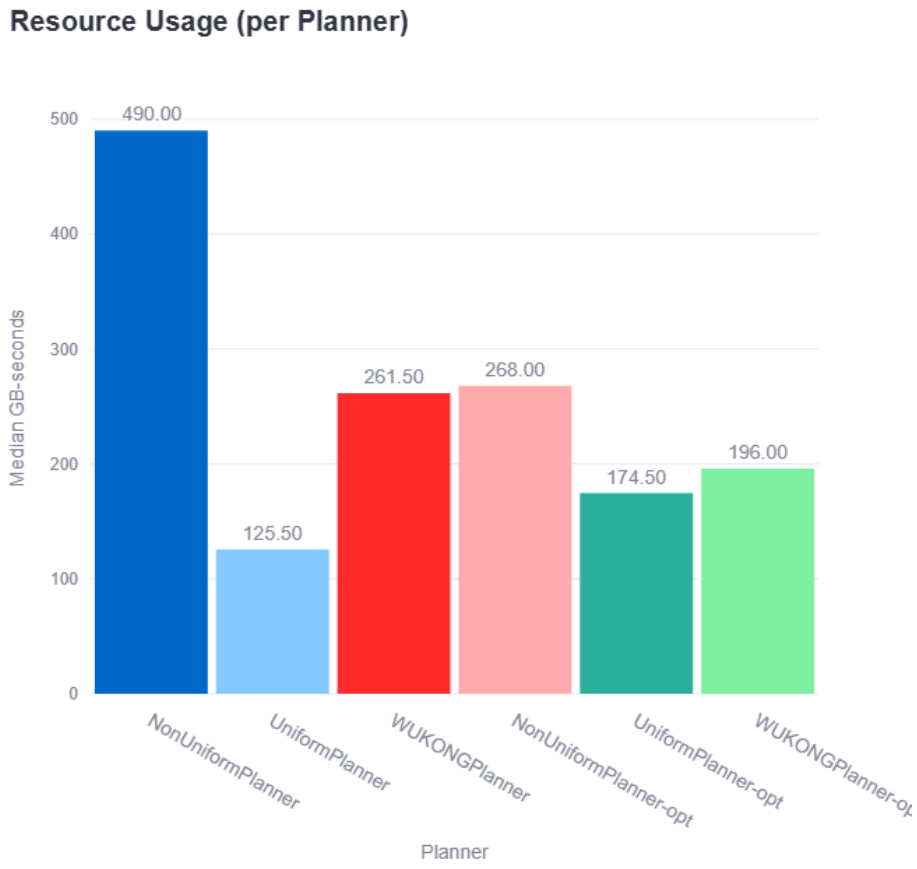

Figure 4.9: Resource usage comparison

makespan of just 4.31s. This represents a **53%** reduction in makespan and a **45%** decrease in memory usage compared to its non-optimized version.

A new trade-off becomes evident among the optimized planners. While the **Non-Uniform** planner is the fastest, the optimized **Uniform** planner is the most memory-efficient, consuming only 174.5 GB-seconds. To achieve its superior speed, the optimized **Non-Uniform** planner is **60%** faster than the optimized **Uniform** planner but requires **54%** more memory resources to do so.

The optimization had mixed results for the other planners. The optimized **WUKONG** planner saw a modest **2.7%** improvement in makespan and a significant **25%** reduction in memory usage. However, it remains a less efficient choice, as it is both slower and more memory-intensive than the **Uniform** planner.

4.5 Analysis

In summary, our results demonstrate that our proposed planners, by managing the trade-off between data locality and resource contention, achieved better performance than both versions of the **WUKONG** planner, with the optimizations significantly enhancing both performance and resource usage.

The **Non-Uniform** planner consistently delivers the shortest *makespan*, particularly in its optimized form, which achieves over **50%** faster execution while reducing resource consumption by **45%** compared to its base version. The **Uniform** planner, while slightly slower, remains the most resource-efficient, achieving the lowest median *GB-seconds* usage among all tested approaches, while keeping workflow execution times lower than both **WUKONG** implementations (non-optimized and optimized).

With its task assignment policy on fan-outs, **WUKONG**'s scheduling strategy reduces resource contention but suffers on worker startup times and worker synchronization, leading to higher resource consumption and making it less suitable for workflows with big fan-outs.

Our optimizations, *pre-loading* and *pre-warming*, proved highly effective in reducing both *makespan* and resource consumption, making them valuable additions to the proposed planners. Resource contention, however, remained a challenge for the **Uniform** planner, which can execute multiple tasks in parallel, often causing different tasks to compete for the same resources. Although the **Non-Uniform** planner follows a similar scheduling approach, its use of more powerful workers mitigated the effects of contention. In our experiments, we limited the number of fan-out tasks assigned to a single worker to **3**, though in practice a worker may execute more tasks from other branches simultaneously. Future work could include a deeper analysis of the impact of resource contention and potentially incorporating contention modeling into the simulation and prediction layers.

The analysis of our results on Service-Level Agreement (SLA) fulfillment reinforces its importance as a cornerstone of our predictive planning engine. This parameter proved to be a highly effective and

practical mechanism for managing user expectations. As demonstrated by the results, by allowing users to select their desired tolerance for performance variability, ranging from a "median" (P50) to a more conservative (P90) estimate, the system consistently delivered accurate predictions. This directly contrasts with other works, which often use complex prediction models [55–58], which would introduce significant computational overhead if used in our solution. This approach would increase prediction times and scale badly, especially as the research configuration space expands with more potential configurations to evaluate. Our lightweight, SLA-driven approach avoids this trade-off, providing decent predictions without compromising the engine's agility. This low overhead is also a key enabler for a potential evolution towards a more dynamic scheduling model, where workers themselves could feasibly make predictions and adjust scheduling decisions on-the-fly as the workflow executes, rather than adhering to a purely static plan.

Overall, the results validate the design goals of our proposed planners: **improving execution efficiency** through smarter task placement and resource utilization, while the introduced optimizations further enhance **performance** and **cost-effectiveness**.

5

Conclusion

Contents

5.1 Achievements 63
5.2 Limitations and Future Work 64

5.1 Achievements

We believe this work successfully addressed some of the key inefficiencies of executing complex workflows in serverless environments identified in other similar works which use uniform worker resources for all tasks and don't take into account a global-view of the workflow structure. The central achievement of this thesis was to answer its primary research question: whether historical metrics could be leveraged to make smarter scheduling decisions that minimize makespan and maximize resource efficiency. We demonstrated this by designing, implementing, and evaluating a decentralized serverless workflow engine that utilizes predictive planning. This approach successfully overcomes the limitations of prior work, such as "one-step scheduling," by leveraging global knowledge of the workflow structure to inform task placement and resource allocation.

Our evaluation demonstrated the clear benefits of a predictive scheduling approach. In direct comparison, our planners consistently outperformed the scheduling model of WUKONG, a state-of-the-art decentralized engine, across four different workflows. The results highlighted a combination of speed and efficiency: our optimized **Non-Uniform** planner delivered the fastest execution, achieving a makespan approximately **63%** shorter than the optimized WUKONG implementation. Simultaneously, our **Uniform** planner proved to be the most resource-efficient, consuming **36%** fewer resources than WUKONG's optimized version. These substantial performance gains are a direct result of our task co-location strategies, which led to a significant reduction in the number of worker invocations compared to WUKONG. This advantage was further amplified by our novel **pre-warming** and **pre-loading** optimizations, which successfully mitigated cold-start latency and halved the time spent waiting for data compared to WUKONG's model.

Beyond these specific results, this thesis also contributes a flexible and extensible framework for future research. The solution's modular architecture, with its clear separation between the **prediction**, **planning**, and **decentralized execution** layers, should allow other researchers to easily implement and comparatively evaluate novel scheduling strategies and prediction models. The project code is available on GitHub, at `https://github.com/48302-DiogoJesus/octoflows`.

5.2 Limitations and Future Work

While this thesis successfully presents and validates a novel solution for efficiently scheduling serverless workflows using prediction-based scheduling strategies, it also has several limitations, which opens opportunities for future research. Such opportunities can be broadly categorized into core solution improvements, usability enhancements, and further experimentation.

The current architecture, though functional, can be refined. A main limitation is the unbounded growth of stored prediction samples. Implementing a **data summarization or downsampling strategy** would condense historical data while preserving key statistics, improving both scalability and prediction speed. The **optimization system** could also benefit from a stronger abstraction layer to handle **conflicting optimizations** that override the same execution phases. Finally, adding robust **error handling**, including **exponential backoff** retries and clearer client-side error reporting, would improve reliability and move the system closer to production readiness.

Another promising direction for improvement lies in encouraging FaaS platforms to support **decoupled memory and CPU configurations**. Allowing these resources to be configured independently would enable a more precise alignment between task requirements and allocated resources, thereby enhancing efficiency, adaptability, and enabling **finer-grained pricing models**. However, introducing such flexibility would also considerably increase the complexity of **scheduling, resource allocation,**

and locality optimization from the provider's perspective. If this were to be implemented in existing cloud providers, our solution would **naturally extend** to such environments, as its design is not constrained by specific resource coupling assumptions. Nevertheless, the increased dimensionality of possible resource configurations would significantly expand the prediction search space, potentially leading to slower planning times.

More aggressive optimizations could also be explored. For instance, *task duplication* [59], represents a different class of optimization from those currently implemented, as it intentionally increases resource usage. Such an approach could be used to hide latency and enable earlier task execution, at the cost of higher resource consumption. Evaluating this trade-off would provide valuable insight into the balance between performance gains and resource efficiency.

Enhancing user control and interaction would broaden the solution's applicability and appeal. The system's expressiveness could be enhanced by incorporating support for **dynamic fan-outs**. Despite hurting predictions accuracy, this would enable more complex and adaptive workflow patterns by allowing workflows to generate a variable number of parallel tasks based on runtime data. Additionally, the DAG definition language could also be extended to allow users to specify a **minimum resource configurations** for specific tasks. This is important, as some tasks may not be able to run on a run-time with less resources than required. The real-time dashboard could also be transformed it into a fully interactive control panel that would provide greater observability over the workflow executions. This includes making it more informative with richer visualizations and adding capabilities for users to **manage, launch, and even simulate workflows directly from the user interface**.

To better understand the solutions' performance generalizability, future work could include a comprehensive **comparative analysis of alternative prediction strategies**. Evaluating their accuracy, efficiency, and overhead across diverse scenarios would help identify the most effective approaches for different use cases. Additionally, with more time and detailed metrics, it would be valuable to **quantify the individual contribution of each optimization** to overall performance improvements. The current solution was validated in a controlled, semi-predictable environment; therefore, a key next step is to deploy and assess the worker logic on **commercial Function-as-a-Service (FaaS) platforms** (e.g., AWS Lambda, Google Cloud Functions). This would test its robustness and scalability under the variable and unpredictable conditions of real cloud environments, or even on edge computing environments [60, 61]. Finally, evaluating the system on a **broader set of real-world workflows**, with more complex dependency graphs and longer execution chains, would provide deeper insights into its strengths, limitations, and potential areas for improvement.

Bibliography

[1] 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.

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

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

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

[5] 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

[6] 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

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

[8] 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.

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

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

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

[12] 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

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

[14] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

[28] A. Casalboni, “AWS Lambda Power Tuning,” https://github.com/alexcasalboni/aws-lambda-power-tuning, 2024.

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

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

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

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

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

[34] 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.

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

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

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

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

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

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

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

[42] S. Esteves and L. Veiga, "Waas: Workflow-as-a-service for the cloud with scheduling of continuous and data-intensive workflows," *Comput. J.*, vol. 59, no. 3, pp. 371–383, 2016. [Online]. Available: https://doi.org/10.1093/comjnl/bxu158

[43] S. Esteves, H. Galhardas, and L. Veiga, "Adaptive execution of continuous and data-intensive workflows with machine learning," in *Proceedings of the 19th International Middleware Conference, Middleware 2018, Rennes, France, December 10-14, 2018*, P. Ferreira and L. Shrira, Eds. ACM, 2018, pp. 239–252. [Online]. Available: https://doi.org/10.1145/3274808.3274827

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

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

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

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

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

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

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

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

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

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

[54] 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.

[55] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "Cose: Configuring serverless functions using statistical learning," in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, 2020, pp. 129–138.

[56] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "{ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 303–320.

[57] 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.

[58] S. K. Koney, "Predictive and adaptive scheduling of serverless ml pipelines for cost-efficient and low-latency execution," *Journal Of Engineering And Computer Sciences*, vol. 4, no. 7, pp. 620–629, 2025.

[59] 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.

[60] P. Kathiravelu, P. V. Roy, and L. Veiga, "SD-CPS: software-defined cyber-physical systems. taming the challenges of CPS with workflows at the edge," *Clust. Comput.*, vol. 22, no. 3, pp. 661–677, 2019. [Online]. Available: https://doi.org/10.1007/s10586-018-2874-8

[61] P. Kathiravelu, Z. Zaiman, J. Gichoya, L. Veiga, and I. Banerjee, "Towards an internet-scale overlay network for latency-aware decentralized workflows at the edge," *Comput. Networks*, vol. 203, p. 108654, 2022. [Online]. Available: https://doi.org/10.1016/j.comnet.2021.108654

A

Structure of Workflows used for Evaluation

Figure A.1: Matrix Multiplication Workflow (Partial)

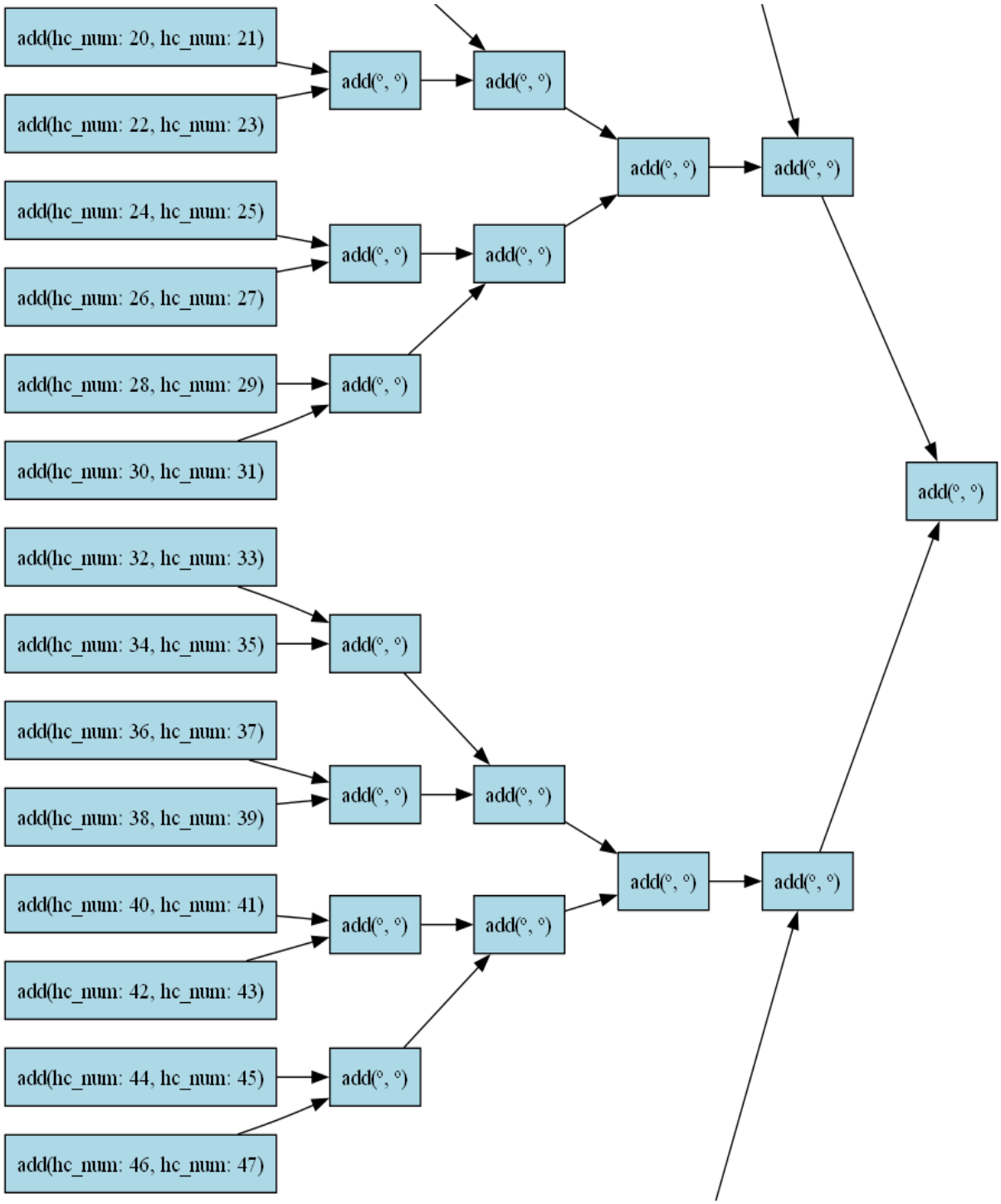

Figure A.2: Tree Reduction Workflow (Partial)

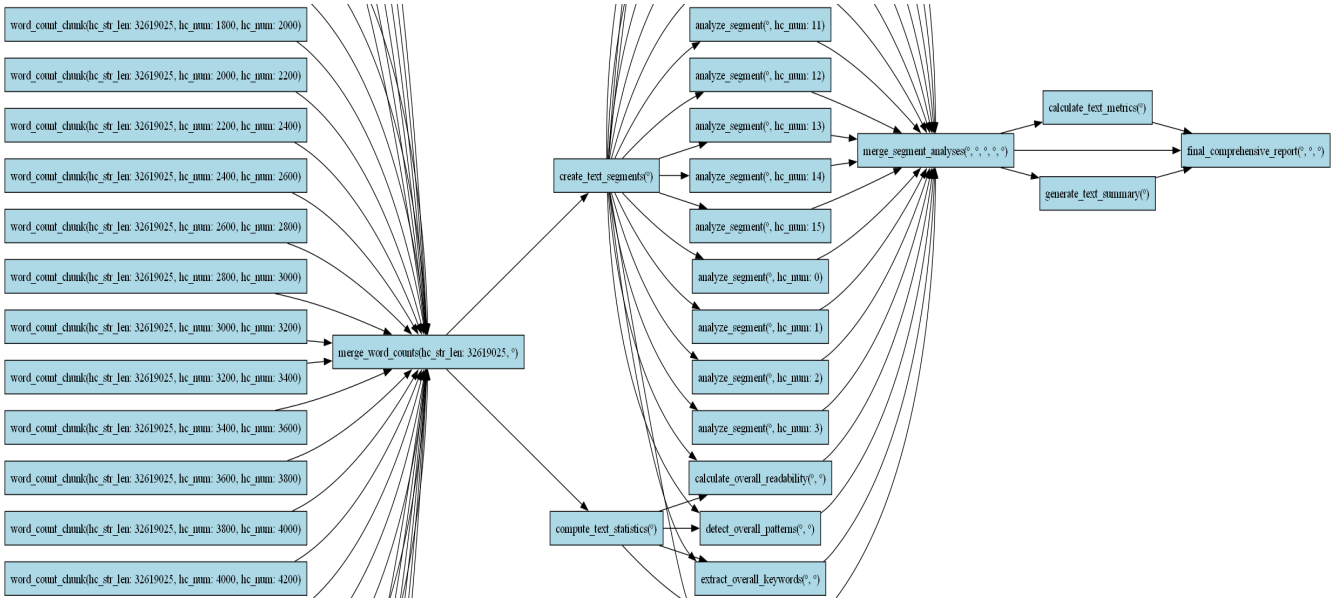

Figure A.3: Text Analysis Workflow (Partial)

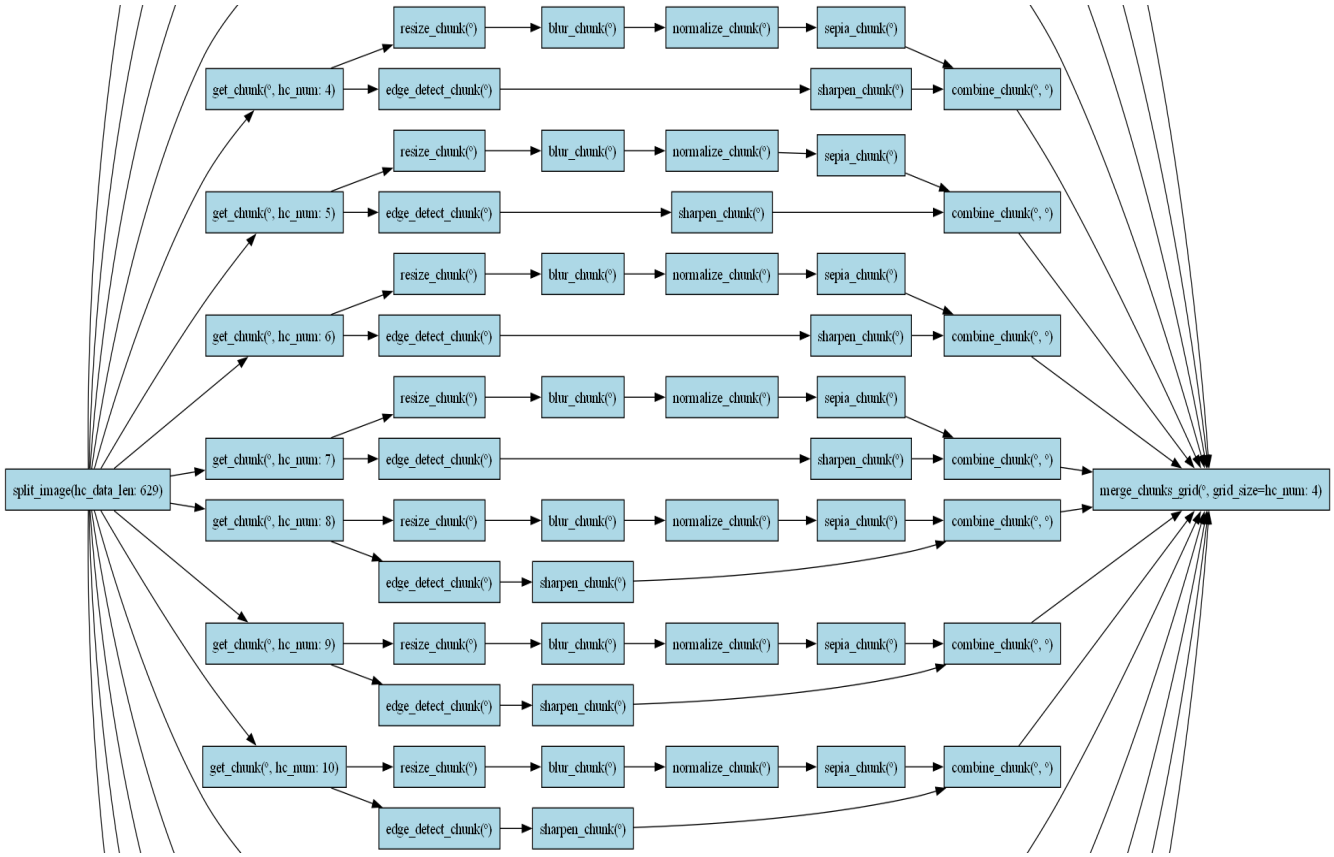

Figure A.4: Image Transformation Workflow (Partial)