TÉCNICO
LISBOA

Serverless Dataflows: ...

Diogo Alexandre Ferreira de Jesus

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Luis Manuel Antunes Veiga

Examination Committee

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

October 2025

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

Acknowledgments

Abstract

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

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

Keywords

Maecenas tempus dictum libero; Donec non tortor in arcu mollis feugiat;Cras rutrum pulvinar tellus.

Resumo

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

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

Palavras Chave

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

Contents

List of Figures

List of Tables

List of Algorithms

Listings

1

Introduction

Contents

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

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

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

1.1 Problem/Motivation

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

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

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

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

1.2 Gaps in prior work

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

1.3 Proposed Solution

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

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

1.4 Document Organization

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

2

Related Work

Contents

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

2.1 Serverless Computing

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

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

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

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

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

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

appeal of serverless architectures, which can be attributed to the following key benefits:

- **Operational Simplicity** means that developers are abstracted away from the underlying infrastructure management, without worrying about server maintenance, scaling, or provisioning. This enables faster development and deployment cycles;
- **Scalability** means the FaaS runtime handles increasing workloads by automatically provisioning additional computational capacity as demand grows, ensuring that applications remain responsive and performant. This makes the FaaS model ideal for applications with *highly variable* or *unpredictable* usage patterns, where we don't know *how many* or *when* requests will arrive;
- **Pay-per-use**: FaaS provides a pricing model where users are only charged for the resources used during the actual execution time over the memory used by their functions, rather than for pre-allocated resources (as in Infrastructure-as-a-Service).

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

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

2.1.2 Limitations

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

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

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

- **Scheduling Policies**: Traditional cloud computing policies were not designed to operate in dynamic and ephemeral environments, such as FaaS's;

- **Resource Management**: Particularly storage and networking, needs to be optimized (by service providers) to handle the low latency and scalability requirements of serverless computing. The lack of direct inter-function networking is an example of a limitation that narrows down the variety of applications that can currently run on FaaS, as some may not support the overhead of using intermediaries (external storage) for data exchange;

- **Fault-Tolerance**: Cloud platforms impose restrictions on developers by encouraging the development of *idempotent functions*. This makes it easier for providers to implement fault-tolerance by retrying failed function executions.

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

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

2.1.3 Research Efforts

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

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

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

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

Palette [12] is a FaaS runtime extension that improves data locality by introducing the concept of **"colors"** as **locality hints**. These colors are parameters attached to function invocations, enabling the invoker to express the desired affinity between invocations without directly managing instances. Palette then uses these hints to route invocations with the same color to the same instance *if possible*, allowing for data produced by one invocation to be readily available to subsequent invocations, reducing the need for expensive data transfers, as it would be required in a typical FaaS runtime.

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

- **Improved Cache Hit Ratios:** Palette enables effective local caching by routing related requests to the same instance, leading to *higher hit ratio* of local caches. For example, in a social networking application, requests related to the same post or user can be assigned the same color, leading to a higher cache hit rate for the post data;

- **Reduced Data Transfer Costs:** For data-intensive applications, Palette can minimize expensive data transfers between function instances;

- **Flexibility in Coloring Policies:** Palette allows users to implement different coloring policies based on the application's needs. This opens the possibility for *serverless workflow schedulers to achieve greater results* by influencing co-location of function invocations. For instance, schedulers can choose to color tasks within a workflow based on their data dependencies, or they can use a more dynamic approach that takes into account factors such as data size and task execution time to *implement custom policies*.

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

A – Faa$T Caching Faa$T (Function-as-a-Service Transparent Auto-scaling Cache) (Romero et al. [11]) is a serverless **caching layer** where each application has its own dedicated in-memory cache, called a *"cachelet"*. This *cachelet* stores data accessed during function executions, making it readily available for subsequent related calls, and thus, minimizing the need for expensive data transfers from remote storage.

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

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

Faa$T core mechanisms are the following:

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

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

B – Lambdata Similarly to Palette, Lambdata (Tang et al. [13]) focuses on optimizing data locality by leveraging user-provided information. In Lambdata this information is called "data intents", where users specify *exactly* what data objects a function will *read and/or write*.

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

- **Data Intent Declarations:** Function invokers annotate their functions with *“get_data“* and *“put_data“* parameters, explicitly listing the data objects required for input and output, respectively. These annotations provide the *Lambdata Controller* with insights into the data access patterns of each function. For instance, a thumbnail-generating function would declare its intent to read a specific image file (“pic/1.jpg“) and write the resulting thumbnail to another location (“thumb/1.jpg“);

- **Local Caching:** Each *Lambdata Invoker*, responsible for executing functions, maintains a local object cache. When a function is invoked, Lambdata checks the local cache for the required data objects specified in the “get_data“ intent. If the data is present locally, the function directly reads it from the cache, avoiding the latency of retrieving it from remote storage;

- **Data-Aware Scheduling:** The *Lambdata Controller* utilizes the declared data intents to make *informed scheduling decisions*. When multiple functions require access to the same data, Lambdata tries to schedule them on the same *Invoker*. This colocation allows the functions to *reuse the cached data*, minimizing data transfer overheads.

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

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

2.2 Workflows

2.3 Relevant Related Systems

2.3.1 Serverless Workflow Scheduling

2.4 Discussion/Analysis

3

Architecture

Contents

3.1 Workflow Definition Language

3.2 Overview

3.3 Metadata Management

3.4 Static Workflow Planning

3.4.1 Simulation Layer

3.4.2 Planners

3.4.3 Optimizations

3.5 Decentralized Scheduling

Algorithm 1 Worker Assignment Algorithm

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

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

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

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

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

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

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

Algorithm 2 Resource Downgrading Algorithm

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

$workers\_outside \leftarrow \emptyset$

▷ 1) Identify workers outside the critical path

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

▷ 2) Attempt downgrade for each worker outside critical path

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

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

    for $i \leftarrow 1$ **to** $|configs| - 1$ **do**
        $trial \leftarrow configs[i].\textsc{Clone}(wid)$

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

4

Evaluation

Contents

4.1 Maecenas vitae nulla consequat

5

Conclusion

Contents

5.1 Conclusions

5.2 System Limitations and Future Work

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[15] M. Wawrzoniak, R. Bruno, A. Klimovic, and G. Alonso, "Boxer: Faast ephemeral elasticity for off-the-shelf cloud applications," 2024. [Online]. Available: https://arxiv.org/abs/2407.00832

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

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

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

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

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

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

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

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

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

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

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

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

A

Code of Project

B

A Large Table