Serverless Dataflows

1st Diogo Jesus
dept. name of organisation (of Aff.)
Instituto Superior Tecnico (IST), INESC-ID Lisboa
Lisbon, Portugal
diogofjesus@inesc-id.pt

2nd Luís Veiga
dept. name of organisation (of Aff.)
Instituto Superior Tecnico (IST), INESC-ID Lisboa
Lisbon, Portugal
luis.veiga@inesc-id.pt

***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–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 system against WUKONG, another decentralized serverless DAG engine, and Dask Distributed, a more traditional cluster-based DAG engine. Our evaluation demonstrates that utilizing historical information significantly improves performance and reduces resource utilization for workflows running on serverless platforms.

***Index Terms*—Cloud Computing, Serverless, FaaS, Serverless Workflows, DAG, Metadata, Workflow Prediction**

I. Introduction

Serverless computing, with its Function-as-a-Service (FaaS) model, offers operational simplicity, automatic scalability, and a fine-grained pay-per-use pricing model by abstracting infrastructure management. 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].

This paradigm is also increasingly used to execute complex scientific and data processing workflows, such as the Cybershake [1] seismic hazard analysis or Montage [2], 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. The stateless nature of serverless functions hinders direct communication, forcing data exchange between these tasks through external storage and leading to performance degradation and increased latency for data-intensive workflows.

Existing solutions, including commercial stateful functions and research prototypes like WUKONG [3], address these issues with improved orchestration and decentralized scheduling. Yet, these approaches often rely on one-step scheduling, making decisions based solely on the immediate workflow stage without considering global implications.

In this paper, we argue that leveraging historical metadata from previous workflow runs can provide critical insights into task behavior. We propose a decentralized serverless workflow execution engine that uses this information to make smarter scheduling decisions, aiming to reduce overall workflow makespan and increase resource efficiency on serverless platforms.

[1]https://aws.amazon.com/pt/lambda/
[2]https://azure.microsoft.com/en-us/products/functions
[3]https://cloud.google.com/functions

II. Related Work

Our work builds upon and differs from existing research in serverless workflow orchestration, which can be broadly categorized into **cloud-native stateful services**, **FaaS runtime extensions**, and **decentralized serverless schedulers**.

A. Cloud-Native Stateful Orchestration

Commercial platforms like AWS Step Functions [4], Azure Durable Functions [5], and Google Cloud Workflows [6] provide a managed solution for composing serverless functions into workflows. They handle state persistence and fault tolerance, abstracting orchestration complexity from the developer. However, they are often tightly coupled with their provider's ecosystem, can introduce vendor lock-in, and are generally designed for reliability and ease of use rather than performance, optimal resource usage or data locality.

B. FaaS Runtime Extensions for Data Locality

A body of work seeks to address the fundamental data exchange inefficiency in serverless platforms through runtime modifications. **Palette** [7] introduces locality "hints" to co-locate related function invocations on the same worker. **Faa$T** [8] provides a transparent, auto-scaling distributed cache for serverless functions. **Lambdata** [9] requires developers to declare data intents (input/output objects) to enable data-aware scheduling. These solutions demonstrate the significant performance gains possible by improving data locality, but often require modifications to the application code or the underlying FaaS platform itself, limiting their portability and adoption.

C. Decentralized Serverless Schedulers

Several frameworks have been designed to execute workflows efficiently on unmodified serverless infrastructure. **PyWren** [10] is an early orchestrator for embarrassingly parallel computations, but its simple design can be inefficient for more complex workflows. **Unum** [11] decentralizes orchestration logic by embedding it within application code, allowing portability across different cloud providers, but offering limited data locality optimizations.

The most directly comparable work to ours is **WUKONG** [3], a decentralized DAG engine that uses static scheduling and runtime optimizations to minimize data movement. WUKONGs' key innovations include:

- **Decentralized Scheduling:** Eliminating the central scheduler bottleneck;
- **Task Clustering:** Forcing the execution of sequences of tasks on the same worker to reuse intermediate results and avoid using external storage;
- **Delayed I/O:** Heuristically postponing data writes to external storage in the hope that dependent tasks can be executed locally.

While WUKONG represents a significant advance, its scheduling and optimizations are based solely on the structure of the *current* workflow DAG. It employs a **one-step scheduling** policy, making decisions using immediate runtime information without leveraging knowledge it has about the entire workflow. Besides that, WUKONG also uses optimizations based on heuristics, which can lead to suboptimal performance when workflow behavior deviates from expected patterns.

D. Discussion

The Function-as-a-Service (FaaS) model offers a transformative approach to cloud computing by abstracting infrastructure management and enabling developers to focus on business logic. Despite current platform limitations, Serverless architectures have been widely adopted for event-driven applications, microservices, IoT processing, and web services.

While researching, we explored some modern cloud-native solutions that seamlessly integrate with cloud environments. We also found innovative extensions to the FaaS runtime that aim to improve **data locality**, a technique that can enhance performance and resource efficiency of serverless workflows by reducing data transfer overheads and removing the need to use external services for synchronization.

Finally, we compared serverless workflow execution orchestrators and schedulers, from which we found WUKONG to be the most interesting, innovative and promising approach for exploiting the most out of the serverless computing paradigm. WUKONG achieves **fast scale-out times** by delegating part of the worker instancing to an external component, **scalability** with its distributed scheduling approach, and **data locality** with its optimizations that try to run related tasks on the same worker, minimizing data transfers. Despite its advantages, we also found that WUKONGs' heuristic optimizations could become inefficient in some scenarios. We believe that WUKONG scheduling decisions are optimized for workflows with short and uniform tasks.

By analyzing related works, we didn't find solutions that tried to make scheduling decisions based on the entire workflow nor that used historical information to make scheduling decisions. We saw this as a research opportunity and decided to explore it with the goal of reducing makespan of workflows running on top of FaaS while also using fewer resources, thereby improving cost efficiency and enabling a wider variety of workflows to run on top of FaaS.

III. Architecture

While current serverless platforms excel at embarrassingly parallel jobs with short-duration tasks, they present challenges for workflows involving significant data exchange between tasks due to their architectural limitations, which deny inter-function communication and control over where each function is executed. In the future, however, serverless platforms are expected to improve, eventually overcoming these limitations and becoming a viable, user-friendly and cost-effective alternative to IaaS for a wide range of workflows.

As we have stated, most existing serverless schedulers employ an approach where decisions are made based solely on the immediate workflow stage without considering the global implications. We hereby propose a novel *decentralized serverless workflow execution engine* that leverages historical metadata from previous workflow runs to make fast predictions and create workflow plans before they execute. Such plans include information about where to execute each task (locality), the worker resource configuration to use (how much vCPUs and Memory) and optimizations. At run-time, the workers will execute the plan and apply the specified optimizations. It was written in *Python*, a language known for its simplicity and popularity among data scientists.

A. Architecture Overview

The overall architecture of our decentralized serverless workflow execution engine is organized into 3 high-level layers. Figure 1 provides an overview of this architecture, while the following sections should provide a deeper understanding of each layer as well as how the user interacts with the system.

1) **Metadata Management**: Responsible for collecting and storing task metadata from previous executions. It also uses this metadata to provide predictions regarding task execution times, data transfer times, task output sizes, and worker startup times;
2) **Static Workflow Planner**: 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 Planner, applying optimizations and delegating tasks as needed.

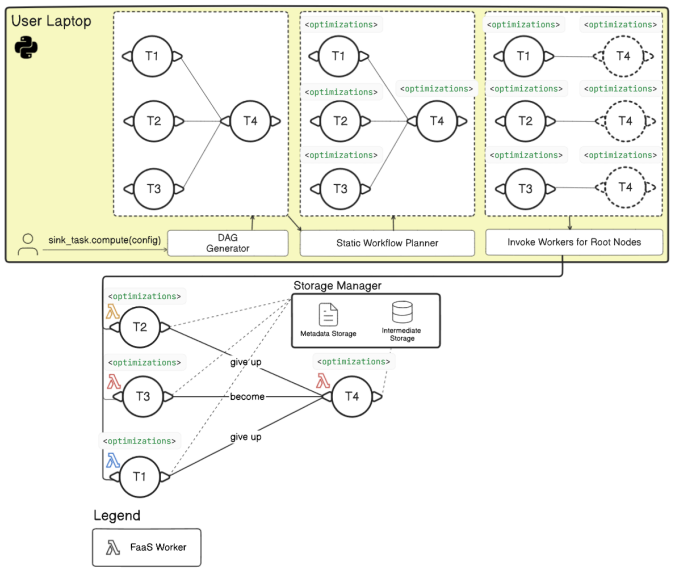

Fig. 1: Solution Architecture

There are 3 distinct computational entities involved in this system:

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

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

B. Workflow Definition Language

The user can create workflows by composing individual Python functions, as shown in listing 2. Here, we define two tasks, `task_a` and `task_b`, and then compose them into a DAG/Workflow by passing their results as arguments to the next task. The resulting workflow can be visualized in figure 3.

When `task_a(10)` is invoked, it doesn't actually run the user code. It instead creates a representation of the task, which can be passed as argument to other tasks. The workflow planning and execution only happens once `.compute()` is called on the last/sink task (`a4`), as shown in listing 4.

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

@DAGTask
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)
```

Fig. 2: DAG definition example

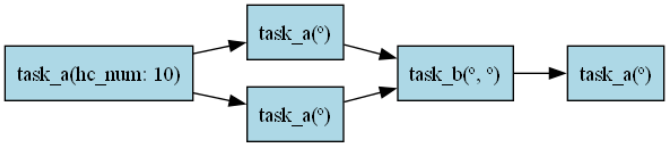

Fig. 3: Simple DAG example

When `compute()` is called, we can create a representation of the entire workflow structure by backtracking the task dependencies.

```
result = a4.compute(
    dag_name="simpledag",
    config=Worker.Config(
        faas_gateway_address=...,
        intermediate_storage_config=(ip, port,
            password),
        metrics_storage_config=(ip, port, password),
        planner_config=SimplePlannerAlgorithm.Config(
            sla=sla,
            worker_resource_configuration=
                TaskWorkerResourceConfiguration(cpus=3,
                memory_mb=512),
        )
    )
)
```

Fig. 4: Triggering workflow execution

It's important to note that this DAG definition language doesn't support "dynamic fan-outs" (e.g., creating a variable number of tasks depending on the result of another task).

We will now go into more detail about each of the 3 layers, unfolding its components and importance to the overall system.

C. Metadata Management

The goal of the **Metadata Management** layer is to provide the most accurate task-wise predictions to help the planner algorithm chosen by the user to make better decisions. To achieve this, while the workflow is running we collect metrics about each task's execution. These metrics are stored in

Metadata Storage: task execution time, data transfer size and time, task input and output sizes, and worker startup time.

Storing these metrics enables us to provide a prediction API, shown in Listing 5. To improve accuracy, metrics are kept separate for each workflow. As a result, even if two workflows use the same function or task code, their metrics are stored independently. This design choice reflects our assumption that different workflows may follow different execution patterns. To avoid introducing runtime overhead, metrics are batched and uploaded when the worker shuts down.

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

```
class PredictionsProvider:
  def predict_output_size(function_name, input_size,
       sla) -> int
  def predict_worker_startup_time(resource_config,
      state: 'cold' | 'warm', sla) -> float
  def predict_data_transfer_time(
      type: 'upload' | 'download',
      data_size_bytes,
      resource_config,
      sla,
      scaling_exponent
    ) -> float
  def predict_execution_time(
      task_name,
      input_size,
      resource_config,
      sla: SLA,
      size_scaling_factor
    ) -> float
```

Fig. 5: Task Predictions API

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

D. Static Workflow Planner

This layer executes on the user side, and it receives the workflow representation and a workflow planning algorithm chosen by the user (as shown in listing 4). Its job is to execute the planning algorithm, providing it access to the predictions exposed by the Metadata Management layer (section III-C).

Planners can run *workflow simulations* based on the predictions, allowing them to experiment with different resource configurations for different tasks, different task co-location strategies, and different optimizations. The accuracy of this simulation depends on the accuracy of the predictions exposed by the *Predictions API*.

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 [7], but in our case this assignment is mandatory rather than advisory, giving strict control over execution locality.

Users can select from a range of algorithms or implement their own by adhering to the `PlanningAlgorithm` interface [22]: The algorithm must also output a

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

Fig. 6: PlanningAlgorithm interface

`WorkerConfigurationMapping`, which maps `Worker IDs` to their respective `Worker Configurations` (vCPU and RAM) [23]: Two algorithms are proposed for

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

Fig. 7: WorkerConfigurationMapping structure

implementation [23, 24]:

1) A **uniform Lambda worker algorithm**: This algorithm uses the `MetadataAccess` API to predict and optimize the longest workflow path (critical path) by simulating the `pre-load` optimization. It iteratively optimizes the critical path until no further improvements can be made, providing a benchmark against WUKONG 2.0 to assess the value of historical metadata [23, 24].
2) A **non-uniform worker algorithm**: This algorithm first identifies the critical path assuming all tasks run on the most performant worker configuration. It then strategically downgrades resources on other paths to maximize resource efficiency without creating new critical paths. After tasks are attributed to workers at plan-time, it simulates additional optimizations to further enhance resource efficiency and reduce makespan [24]. This algorithm aims for better performance and resource efficiency by masking data transfer latency and minimizing large data transfers [24].

Future enhancements could include defining "re-planning points" within the workflow, allowing the initial plan to

be dynamically adjusted during runtime based on updated metadata for greater precision, though this introduces potential overhead [25].

E. Scheduling

The **Scheduling** layer is integrated into the workers and is responsible for executing the plan generated by the Static Workflow Planner [25]. By having foreknowledge of task placements, the Scheduler can apply various optimizations defined by the algorithm [25]. These optimizations include [21]:

- **Pre-warm**: Sending empty invocations to FaaS workers to reduce cold start latencies and achieve a shorter workflow makespan [21].
- **Pre-load**: Allowing a worker, designated to execute a downstream task, to begin downloading its data dependencies from remote storage concurrently while it executes other tasks. This parallelizes data download and can reduce overall makespan [21].
- **Task-dup** (Task duplication): If a worker is waiting for data from an upstream task (currently executing or pending on another worker), it can execute that upstream task itself. This can reduce makespan and save data download time, as the waiting worker no longer needs to wait for the data to be uploaded to external storage and then download it [21].

To illustrate the impact of these scheduling approaches, consider a workflow with Task 1 outputting large data, and Task 2 also feeding into Task 5, which also depends on Task 1.

In **WUKONGs' approach**, if Task 1's output is large, its worker attempts to execute all downstream tasks (e.g., Task 3, Task 4, and Task 5) locally to avoid large data transfers [26]. Assuming Task 2 finishes after Task 1, Task 1's worker would execute Tasks 3 and 4, then check if Task 5 is ready. If Task 2 hasn't finished, Task 1's worker begins uploading its large output to external storage. Task 5 would only run after this upload, and potentially a download by another worker [26]. This can be inefficient if Task 2 then finishes, but Task 5's worker still has to wait for Task 1's output, which was unnecessarily delayed [26].

With a **plan-based approach**, as soon as Task 1's worker finishes, it immediately uploads Task 1's output to external storage, knowing it will not execute Task 5 [27]. This means when Task 5 is ready to execute (i.e., Task 2 is complete), it only needs to download Task 1's output, as the upload has already occurred, potentially in parallel with other computations [27]. This is effective because the worker knows its exact role from the plan [27].

By adding **optimizations like pre-warming and pre-loading**, the plan-based approach can be further enhanced [28]. For example, Task 1's worker can pre-warm the worker designated for Task 3, reducing cold start latency [28]. Simultaneously, if Worker 2 needs Task 1's output for Task 5, it can start downloading Task 1's output while it executes Task 2 [28]. This pre-loading of data dependencies in parallel significantly helps in reducing the overall workflow makespan [28].

IV. Evaluation and Analysis

A. Testing Environment

B. Testing Configurations

V. Conclusion

References

[1] 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

[2] 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

[3] 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.

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

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

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

[7] 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/

[8] 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

[9] 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.

[10] 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.

[11] 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.