Serverless Dataflows: A Decentralized Workflow Execution Engine with Predictive Planning

Diogo Jesus
Instituto Superior Tecnico (IST), INESC-ID Lisboa
Lisbon, Portugal
diogofjesus@inesc-id.pt

***Abstract*—Serverless computing offers operational simplicity and automatic scalability, but executing multi-task workflows in Function-as-a-Service (FaaS) environments remains inefficient due to stateless functions and heavy reliance on external storage.**

We present an adaptive decentralized DAG engine that uses historical metadata to guide task scheduling, combining metadata management, static workflow planning, and low-synchronization worker-level scheduling. Compared to WUKONG, a state-of-the-art decentralized serverless DAG engine, our most resource-efficient planner reduces execution time by 12.6% and resource consumption by 36%. For users prioritizing speed, our fastest planner achieves a 57.5% shorter makespan at the cost of a 114% increase in resource usage.

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

I. Introduction

Function-as-a-Service (FaaS) simplifies application deployment by abstracting infrastructure management and providing automatic, fine-grained scalability with pay-per-use pricing. These advantages have driven its adoption for event-driven systems and lightweight microservices on platforms such as AWS Lambda [1], Azure Functions [2], and Google Cloud Run Functions [3]. Recently, FaaS has also been used to execute complex scientific and data-processing workflows, typically represented as Directed Acyclic Graphs (DAGs). However, running such workflows efficiently on serverless platforms remains challenging.

Serverless environments introduce cold-start latency, lack direct inter-function communication, and often require external services for intermediate data exchange. Platform-specific workflow languages restrict portability, while the stateless and opaque execution model complicates resource optimization. Existing solutions, including stateful serverless services, extended FaaS runtimes, and workflow-level schedulers, partially address these issues but still suffer from two key limitations: (1) *one-step scheduling*, which considers only the current task rather than the full workflow context, and (2) *uniform resource allocation*, which is inefficient when tasks vary in computational or memory demands. Moreover, no prior work leverages historical execution metadata to inform scheduling decisions across an entire workflow.

This motivates our central research question: given knowledge of workflow structure and historical task behavior, can we make globally informed scheduling decisions that minimize makespan and maximize resource efficiency in a FaaS environment?

To answer this, we propose an adaptive decentralized serverless workflow execution engine that uses historical metadata to generate informed task allocation plans executed cooperatively by FaaS workers. This approach reduces reliance on external storage for intermediate data and avoids the inefficiencies of homogeneous worker configurations.

The key contributions of this work are:

- An analysis of current serverless workflow orchestration approaches and their limitations;
- A decentralized execution engine that overcomes one-step scheduling and uniform-resource constraints by using historical metadata to generate workflow-wide execution plans;
- Evaluation of our solution, demonstrating the benefits of using historical execution data to improve task placement, reduce external storage usage, and enhance overall workflow efficiency on FaaS platforms.

II. Architecture

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 *adaptive decentralized serverless workflow execution engine* that leverages historical metadata from previous workflow runs to make lightweight 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. Our solution was written in *Python*, a language known for its simplicity and popularity among data scientists.

A. Workflow Definition Language

We will now present our workflow language that transforms ordinary Python functions into parallelizable tasks, automatically managing dependencies and execution through an intuitive decorator-based API. It is inspired by WUKONG, Dask and Airflow: the user creates workflows by composing individual Python functions, as shown in Listing 1. In this example, we define two tasks, `task_a` and `task_b`, and then compose them into a workflow by passing their results as arguments to the next task.

Listing 1: DAG definition example

```
1  # 1) Task definition
2  @DAGTask
3  def task_a(a: int) -> int:
4      return a + 1
5
```

```
6  @DAGTask(forced_optimizations=[
       PreLoadOptimization()])
7  def task_b(*args: int) -> int:
8     return sum(args)
9
10 # 2) Task composition (DAG/Workflow)
11 a1 = task_a(10)
12 a2 = task_a(a1)
13 a3 = task_a(a1)
14 b1 = task_b(a2, a3)
15
16 # 3) Workflow execution
17 b1.compute(storage_configs, planner,
       planner_config)
```

While this example passes data directly, passing storage references (e.g., cloud object storage URLs) as function arguments is a common pattern in serverless workflows that is not supported by our solution. This limitation could be addressed through **code instrumentation**, wherein storage access APIs would be intercepted to record relevant metrics.

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. Workflow planning and execution occur only when `.compute()` is called on the final sink task (`b1`) (line 19), at which point the system backtracks through task dependencies to construct a DAG representation of the entire workflow.

One limitation of our DAG definition approach 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 [4], WUKONG [5], Unum [6], Oozie [7] 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 [1] supports this feature through an extension to their DAG language, allowing a variable number of tasks to be created at run-time depending on the number of results produced by a previous task. Implementing similar functionality is possible, but it would reduce the accuracy of predictions. This is because we would also need to predict the expected fan-out size, and any errors in that prediction could amplify inaccuracies in the predictions for the rest of the workflow.

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

B. Architecture Overview

Figure 1 shows the overall architecture and logical flow of our decentralized serverless workflow execution engine, which is organized into 3 high-level layers. The upper part of the figure represents the components that run on the user's machine, while the lower part represents the components that run outside the user's machine.

After the user writes its workflow in Python, as demonstrated in Section II-A, it can then specify a planning algorithm, which will run locally to generate a static workflow plan, defining a task-to-worker mapping and other task-level optimization hints for FaaS workers. Once the plan is created, 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, when it finally retrieves the result from storage.

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

1) **Metadata Management**: Responsible for collecting and storing task metadata from previous executions. It then 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 workflow plan/schedule;
3) **Decentralized 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 without the need for a central entity.

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;
- **Workers**: FaaS workers that execute workflow tasks and handle scheduling, task delegation, and launching new workers without a central scheduler. They also collect and upload relevant metadata;
- **Storage**: Consists of an *Intermediate Storage* for task outputs which may be needed for subsequent tasks and a *Metadata Storage* for storing metrics (used for predictions) and information crucial to workflow execution (e.g., notifications about task readiness and completion).

Next, we will go through the **three** layers that compose our solution: **Metadata Management**, **Static Workflow Planning**, and **Decentralized Scheduling**.

C. Metadata Management

The goal of the **Metadata Management** layer is to provide accurate task-wise predictions to help the planner algorithm chosen by the user make better decisions. To achieve this, while the workflow is running, we collect metrics (i.e., 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, which estimates upload/download times, worker startup times (whether it's cold or warm), and function output sizes and execution times. To improve accuracy, metrics are kept separate for each workflow meaning that, 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 have different characteristics and may have different execution patterns. Metrics are batched and

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

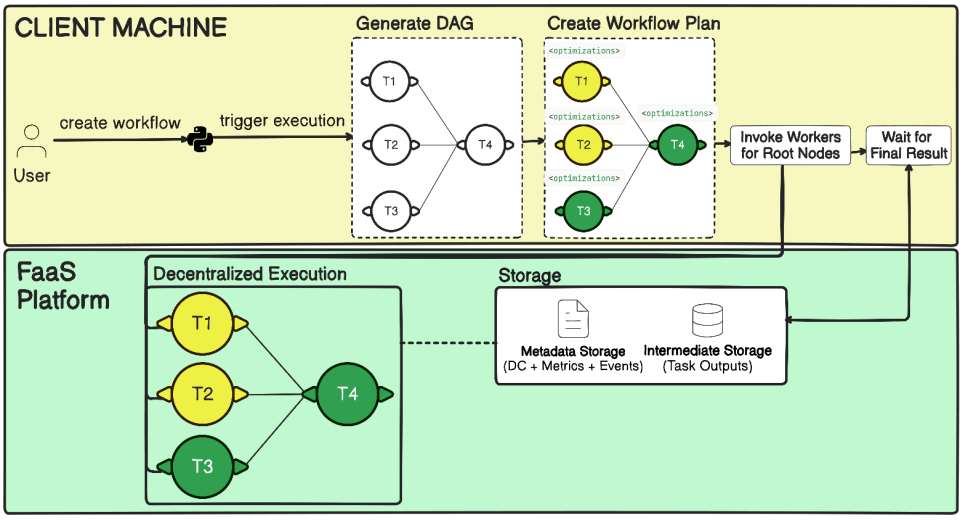

Fig. 1: Solution Architecture

uploaded when the worker shuts down, to reduce runtime overhead.

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=Percentile(50)` will use the median of the historical samples, whereas `SLA=Percentile(80)` will return a more conservative estimate. By allowing the user to control this parameter, the API can provide predictions that are tailored to different performance requirements.

In addition, metrics such as worker startup time, data transfer time, and task execution time are tied to the specific worker resource configuration. To account for this, our prediction method follows two paths. If we have enough historical samples for the same resource configuration, we use only those. Otherwise, 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.

D. Static Workflow Planning

This layer executes on the client, and it receives the workflow representation and a workflow planning algorithm chosen by the user. Its job is to run the planning algorithm, providing it access to the workflow DAG and predictions exposed by the Metadata Management layer (Section II-C).

Planners can run *workflow simulations* based on the predictions, allowing them to experiment with different resource configurations for different tasks and different task co-location strategies. Additionally, they can apply different user-selected optimizations. The accuracy of these simulations 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 [8], but in our case this assignment is mandatory rather than advisory, giving strict control over execution locality. Two tasks assigned the same `worker_id` will 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 [5] scheduling. We refer to these workers as *"flexible workers"*.

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

1) **WUKONG**: All tasks will use the same worker configuration (specified by the user) and won't be assigned a `worker_id`, meaning they will be executed by *"flexible workers"*. This is a dynamic scheduling approach where tasks aren't tied to specific workers, trying to reproduce WUKONG's scheduling behavior;
2) **Uniform**: Tasks use a common worker configuration specified by the user, with each task assigned a `worker_id`, allowing co-location of tasks;
3) **Non-Uniform**: Tasks can use different worker configurations (a 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** initially selects the most powerful configuration from the user-specified options for each task. After assigning initial resources, both planners employ the logic detailed in Algorithm 1 (in Appendix) for Worker ID assignment. This algorithm implements a **balanced clustering strategy** that

Algorithm 1 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

uses history to try maximizing data locality while avoiding resource contention. It achieves this by launching the minimal number of new workers necessary to maintain efficiency, while avoiding overloading any individual worker with excessive task assignments; and minimizing network data transfers by co-locating tasks whose outputs are expected to be larger. This clustering approach achieves a **balance between resource contention and data locality**, contrasting with **WUKONG**.

After Worker ID assignment, the **Non-Uniform** planner runs an additional algorithm 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 global effect of downgrading resources of workers *outside the critical path* with different configurations.

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

Aside from `worker_id` and resource assignments, planners can also apply different user-selected **optimizations** to further improve the workflow execution. We implemented and tested two optimizations, **pre-warm** and **pre-load**:

1) **pre-warm**(worker_config, delay_s): This optimization triggers a special invocation to the FaaS gateway to proactively launch a worker with the specified `worker_config`, masking cold-start latency. The planner selects a task whose execution window aligns with the ideal pre-warming interval, so the worker is neither initialized too early nor too late.
2) **pre-load**: This optimization allows a worker to proactively download a task's inputs as soon as its upstream dependencies complete, using notifications from the *Metadata Storage*. This optimization is assigned to tasks that depend on two or more upstream tasks executed on different workers.

Algorithm 2 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) 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 shorts tasks
  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

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 1, where the user explicitly requests that `task_b` applies the *pre-load* optimization.

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

To illustrate this execution model, Figure 2 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`.

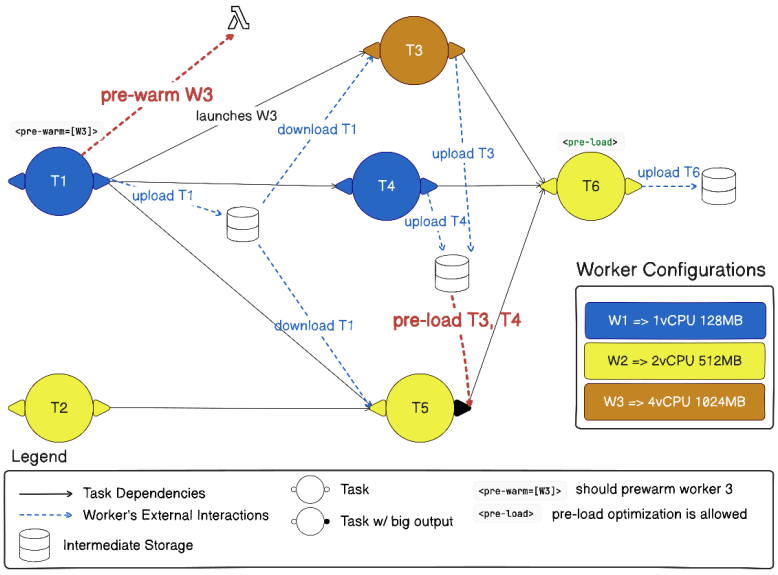

Fig. 2: Planned Workflow Execution Example

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 `Task 3`'s worker have a warm start. 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.

E. Decentralized Scheduling

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

When invoked, a worker receives the workflow structure with the embedded plan and the `task_ids` of the tasks it should execute first. Rather than immediately executing the initial tasks, the worker first subscribes to `TASK_READY` and `TASK_COMPLETED` events for specific tasks. These events are essential both for enabling certain optimizations and for ensuring the worker follows the workflow plan correctly.

After this preparation phase, the worker starts executing its initial tasks concurrently. The logic for executing tasks is the following:

1) **Gathering Dependencies**: Check which dependencies are missing (not downloaded yet) and download them from storage. Some dependencies might be present in the local representation of the DAG if `pre-load` was applied, if the same worker executed some of the upstream tasks, or if the worker already downloaded large hardcoded data because previous tasks needed it;
2) **Executing Task**: Execute task code, which is embedded within the workflow representation, similarly to WUKONG. Tasks' code execute on a separate thread, to avoid slowing down or even blocking (e.g., if the task code calls `time.sleep()`) the main thread, where the other coroutines are running;
3) **Handling Output**: Evaluate whether it's necessary to upload the task's output to storage and emit 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 last task of the workflow;
4) **Updating Dependency Counters**: For each downstream task, the perform an *atomic increment and get* operation on a *"Dependency Counter"* (inspired by WUKONG [5]) stored in *Metadata Storage*, which tracks how many dependencies of a task have been satisfied. If the counter value 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 identifies tasks that became *READY* and consults the execution plan. Tasks assigned to itself with no remaining dependencies are executed (one coroutine per task). For tasks assigned to another active worker, a `TASK_READY` event is emitted; for tasks assigned to an inactive worker, it launches that worker passing it the task IDs of the branches it should execute. Thanks to planner validation (Section II-D), a worker can always determine whether another worker is active by inspecting the workflow plan.

Both *Intermediate Storage* and *Metadata Storage* are implemented in Redis for deployment simplicity. For exchanging events among workers, we use Redis's Pub/Sub [2]. Such events include `TASK_READY` and `TASK_COMPLETED`, and are implemented as Pub/Sub channels.

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*.

In contrast to traditional FaaS-based workflow engines that rely on a centralized scheduler, our system's scheduling is driven by the workers themselves. This decentralized model eliminates continuous coordination with a central controller, reducing overhead and removing a single point of failure. By enabling workers to trigger subsequent tasks immediately after completing their own, this approach minimizes scheduling latency and improves scalability, which primarily depends on the horizontal scalability of the underlying FaaS and storage layers. The client launches the initial set of workers, after which execution proceeds autonomously based on metadata embedded within the workflow representation passed among the workers. A lightweight coordination mechanism (atomic dependency counter) ensures that all tasks are eventually executed according to the scheduling plan and applied optimizations.

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 comparing it against a state-of-the-art FaaS workflow engine, WUKONG [5].

III. Evaluation

A. FaaS Environment Emulation

To enable reproducible and controlled experiments, a lightweight Function-as-a-Service (FaaS) emulator was implemented in approximately 300 lines of Python. It reproduces the core behavior of serverless platforms while remaining simple, with greater observability, and allowing us to run lots of experiments inexpensively.

The system consists of two components: a **gateway service** and a **worker runtime**. The gateway, implemented as an HTTP server, receives invocation requests, manages container lifecycles, and enforces resource constraints using Docker's built-in CPU and memory limits[3]. Workers are packaged as Docker images containing the execution logic and a persistent background process to keep the container alive between invocations, until the gateway decides to shut it down (when idle for too long).

Function execution requests are issued to the gateway's `/job` endpoint, specifying task identifiers, resource configurations, and cached results. If no idle container with the required configuration is available and the maximum concurrency (32 containers) is reached, requests are queued until resources become free. To avoid resource contention, each container executes a single task at a time. This constraint is enforced through a file-based locking mechanism.

Idle containers are automatically removed after 7 seconds of inactivity, to simulate cold starts. The gateway also provides a `/warmup` endpoint that pre-allocates containers without executing worker logic, a simplification that makes it easier to perform *pre-warming* without adding extra logic to the workers code. To improve observability, worker logs (stdout and stderr) are streamed to the gateway and captured in real time for debugging.

B. Research Questions

With the evaluation of our 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?
- **RQ2:** Can our prediction-based scheduling approach in serverless workflows achieve lower *makespan* and reduced resource usage compared to reactive or static approaches, such as WUKONG [5]?
- **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?
- **RQ4:** How does a non-uniform worker resource allocation strategy impact performance, and what trade-off does it introduce between performance gains and additional resource consumption compared to a uniform allocation approach?

To address these questions, we collected a comprehensive set of metrics, including task execution time, dependency download time, I/O object sizes, workflow submission time, container statistics, and more. In the following section, we describe the experimental setup used to evaluate the proposed system.

C. Experimental Setup

To evaluate our proposed solution, we deployed the FaaS emulator described in Section III-A on an AMD EPYC 7282 16-Core 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 [4].

The workflows used to test our system were the following:

- **Matrix Multiplication**: A workflow with a straightforward structure that performs distributed matrix multiplication, comparable to a workflow used in WUKONG's evaluation.

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

[3]https://docs.docker.com/engine/containers/resource_constraints/

[4]https://www.cloudping.co/

- **Tree Reduction**: A workflow that performs a hierarchical reduction over a list of numeric values, identical in structure to a workflow from WUKONG's evaluation.
- **Text Analysis**: A workflow with a more complex structure, designed to simulate a multi-stage text processing pipeline on a large text file (750,000 lines).
- **Image Transformation**: A complex, highly parallel workflow composed of 130 tasks that applies multiple transformations to an input image, featuring large fan-outs and heterogeneous task execution times.

We ran our experiments with three different SLAs: 50th percentile (median), 75th percentile, and 90th percentile, and analyzed the fulfillment success rates of each SLA, as well as their prediction accuracy.

To assess our research questions, we implemented and evaluated three core planners: our proposed **Uniform** and **Non-Uniform** planners, and a planner replicating **WUKONG's** scheduling model. For a comprehensive comparison, we tested each planner with and without their respective optimizations enabled (*pre-load/pre-warm* for our planners, and *Task Clustering/Delayed I/O* for WUKONG). Planners using a uniform resource strategy (*Uniform* and *WUKONG* variations) assigned workers with 2GB of memory. The *Non-Uniform* planner could assign more powerful workers from a predefined list of configurations (*2GB, 4GB, and 8GB*). In total, we evaluated six planner variations.

The experimental process was automated with a script that iterated through every combination of planner variation, SLA, and workflow. Each unique configuration was executed 10 times to account for performance variability, resulting in 720 total experiments for analysis. Before each run, the environment was reset to ensure that the initial tasks of every workflow would trigger a cold start, ensuring consistency.

IV. Results

This section summarizes the main evaluation results of our proposed planners, our **WUKONG** planner implementation, and their optimized variants, focusing on prediction accuracy, workflow execution time (*makespan*), and resource efficiency. Note that the optimized versions of the **Uniform** and **Non-Uniform** planners incorporate our proposed *pre-loading* and *pre-warming* optimizations, while the optimized **WUKONG** variant uses its own native *Task Clustering* and *Delayed I/O* techniques.

A. Prediction Accuracy and SLA Fulfillment

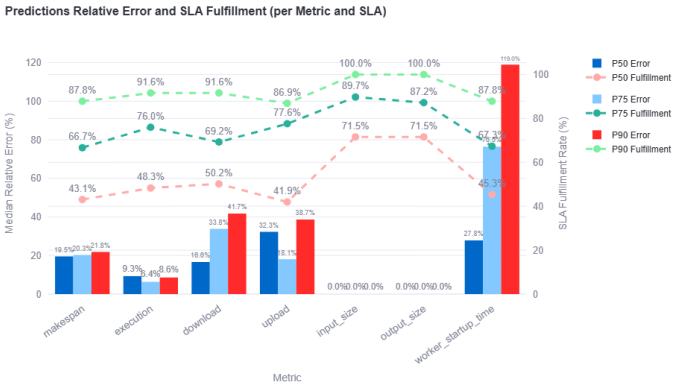

Fig. 3: Predictions accuracy across all planners and SLA fulfillment rates

Figure 3 shows the prediction accuracy and SLA fulfillment rates across all planners (excluding WUKONG, which does not employ predictions). Execution time predictions were highly accurate, with median relative errors below **9.3%**, confirming the reliability of our predictive models for workflow planning on a semi-predictable environment. Data transfer time predictions were less precise, with errors around **35%**, likely due to their higher variability and network-dependent characteristics. SLA fulfillment rates aligned with expectations: **41.9-71.5%** for P50, **66.7-89.7%** for P75, and **86.9-100%** for P90.

B. Execution Performance

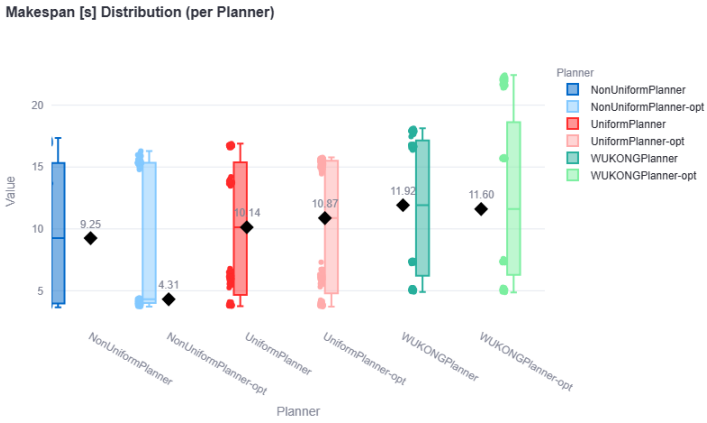

Fig. 4: *Makespan* distribution across all planners

As shown in Figure 4, all versions of our planners outperformed both **WUKONG** implementations in terms of makespan. The non-optimized **Uniform** planner was **12.6% faster** than the optimized version of **WUKONG**, while the optimized **Non-Uniform** planner achieved the best overall performance.

The observed gap between optimized and non-optimized versions of the **Non-Uniform** planner further confirms that

our proposed optimizations (*pre-loading* and *pre-warming*) contribute directly to faster task execution and reduced startup overhead. The same improvement was not observed between the baseline and optimized **Uniform** planners. We attribute this to **resource contention**: the **Uniform** planner allocates fewer resources per worker, and the *pre-loading* optimization increases concurrency within a worker, which can inadvertently slow overall execution.

C. Optimization Effects

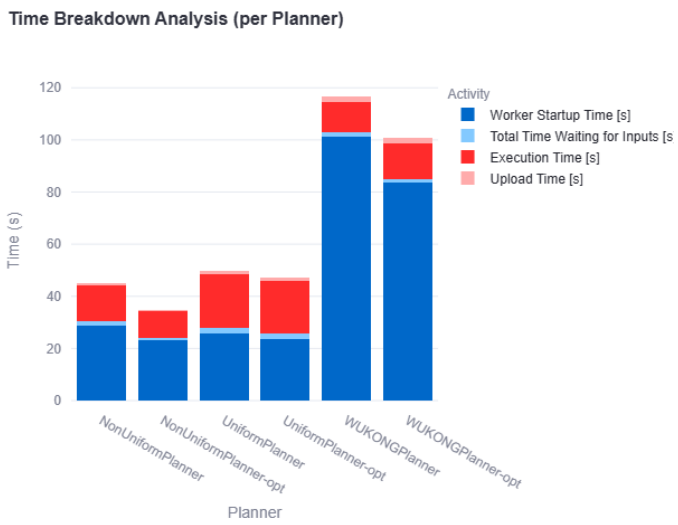

Fig. 5: Time breakdown analysis

Figure 5 highlights the effect of both our scheduling approach that tries to cluster more tasks per worker and our optimizations (*pre-loading* and *pre-warming*). Both optimized planners significantly reduced **worker startup** and **dependency waiting** times compared to their base versions and to **WUKONG**. The use of *pre-warming* minimized cold-start penalties, saving between **2-6 seconds** on average, while *pre-loading* allowed tasks to start executing earlier by proactively transferring dependencies.

The optimized **Non-Uniform** planner achieved the shortest total execution time, whereas the optimized **Uniform** planner maintained higher resource efficiency. This indicates that the system can prioritize either speed or efficiency depending on the selected planner.

D. Performance vs. Resource Efficiency

For context, **GB-seconds** is a commonly used billing metric in serverless platforms, including AWS Lambda [5], calculated as the product of allocated memory (GB) and execution duration (seconds). Figures 6 and 7 summarize the trade-off between execution speed and resource consumption across all tested planners. The optimized **Non-Uniform** planner achieved a **53% faster makespan** and consumed **45% less memory** than its non-optimized counterpart, making it the best performer overall. While the optimized Uniform planner is more resource-efficient, consuming **34.9%** fewer GB-seconds than its optimized Non-Uniform counterpart, it is also **152% slower**.

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

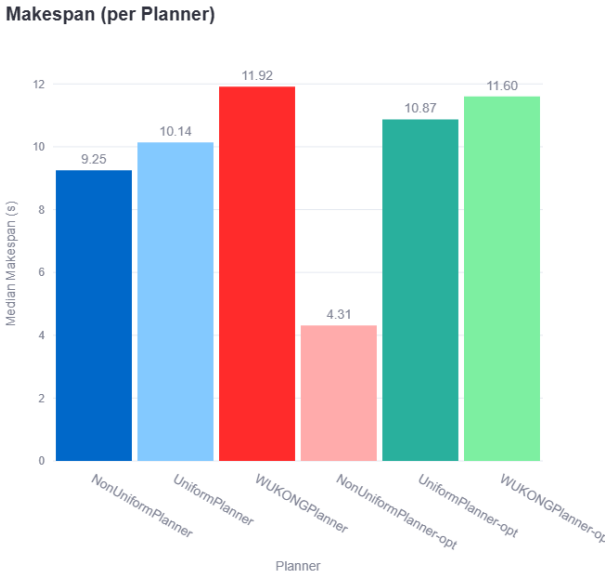

Fig. 6: *Makespan* comparison

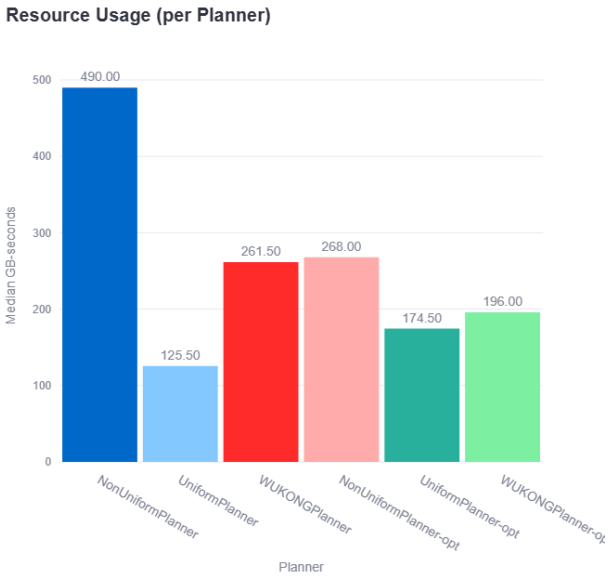

Fig. 7: Resource usage comparison

Comparing to **WUKONG**, both versions were less efficient, with the optimized version only **2.7% faster** and **25% less resource-intensive** than its baseline, but still **14.4% slower** and consuming **56.2%** more resources than our **Uniform** planner.

E. Summary

Overall, our results conclusively demonstrate that the proposed **Uniform** and **Non-Uniform** planners outperform the

WUKONG model in both execution speed and resource efficiency.

The **Uniform** planner establishes itself as the most cost-effective approach. It not only achieves the lowest median resource usage (in GB-seconds) among all tested configurations but also executes workflows faster than both the baseline and optimized versions of **WUKONG**. This highlights a key weakness in **WUKONG**'s scheduling strategy, where its handling of fan-outs leads to excessive worker startup and synchronization overhead, inflating both runtime and resource consumption in most scenarios.

For use cases prioritizing maximum speed, the **Non-Uniform** planner provides a distinct advantage over the **Uniform** planner, consistently delivering the shortest makespan. This performance gain is achieved by leveraging more powerful workers, presenting a clear trade-off between execution time and cost. The impact of our optimizations is most evident with this planner; its optimized version achieves a makespan over **50%** shorter while consuming **45%** fewer resources.

The **SLA** parameter, specified as a **percentile**, was also validated as a practical parameter for managing user expectations. Our lightweight, SLA-driven predictions provided reliable results without the computational overhead of complex models, often used in other solutions [9]–[12], enabling a path toward future dynamic scheduling.

In summary, our results validate three key findings. First, that predictive optimizations like *pre-loading* and *pre-warming* are critical for mitigating latency and inefficiency in serverless workflows. Second, it presents a practical choice for developers/data scientists: the **Uniform** planner for robust and resource-efficient execution, and the **Non-Uniform** planner for greater performance in latency-sensitive applications. Third, it shows the importance and positive impact of properly **balancing resource contention and data locality** in serverless workflows.

V. Related Work

This section reviews the evolution of workflow orchestration, tracing the shift from cluster-based frameworks to serverless execution engines and highlighting the innovations that motivated our approach.

A. Traditional and Cloud-Native Orchestration

Traditional distributed frameworks, such as Hadoop [13], Apache Spark [14], and Apache Flink [15], rely on the MapReduce [16] model or in-memory processing to manage large-scale data. While Dask [4] modernized this by enabling heterogeneous task graphs in Python [17], these cluster-based systems generally incur high operational overhead, slow scaling, and require complex resource provisioning.

To simplify deployment, cloud-native platforms (e.g., AWS Step Functions [18], Azure Durable Functions [19], Google Cloud Workflows [20]) and open-source orchestrators like Apache Airflow [21] provide managed orchestration. These "stateful functions" typically employ checkpointing to persist workflow state, enabling fault tolerance and retries. However, this abstraction often comes with separate billing costs, vendor lock-in, and centralized bottlenecks, without addressing the underlying FaaS inefficiencies like cold starts.

B. FaaS Runtime Extensions

Several works optimize the underlying FaaS runtime to address data locality and communication overhead without replacing the orchestration model:

Palette Load Balancing [8] improves locality by introducing *colors* as hints attached to function invocations. The system attempts to route invocations with the same color to the same worker instance, improving cache hit ratios without strictly enforcing the assignment if resources are unavailable.

Faa$T [22] enables transparent data sharing via an auto-scaling distributed cache. It assigns a dedicated in-memory *cachelet* to each application, using consistent hashing to intercept data requests and serve them from memory, effectively hiding the latency of remote storage.

Lambdata [23] focuses on data locality through explicit *data intents*. Developers annotate functions with `get_data` and `put_data` parameters, allowing the scheduler to co-locate computation with data storage. This avoids the complexity of distributed caching but requires manual developer annotations.

C. Serverless Workflow Execution Engines

Serverless-native engines aim to run workflows on unmodified FaaS infrastructure, offering distinct scheduling strategies:

DEWE v3 [24] utilizes a hybrid architecture to handle diverse resource requirements. It employs a queue-based distribution mechanism where short tasks are routed to FaaS workers and long-running tasks are sent to VMs. While efficient for scientific workflows like Montage [25], this hybrid approach introduces queuing latency and potential VM underutilization.

PyWren [26] pioneered pure serverless execution for "bag-of-tasks" workloads. It executes arbitrary Python functions on AWS Lambda, relying on S3 for all state exchange. While highly scalable, it lacks mechanisms to mitigate cold starts or manage complex dependencies efficiently.

Unum [6] decentralizes orchestration by embedding logic into a library wrapping the user's functions. It uses a consistent data store for coordination and translates workflows into an Intermediate Representation (IR). However, Unum supports only static control structures and lacks data locality optimizations.

D. WUKONG

WUKONG [5] is the most sophisticated decentralized engine related to our work. It employs a static scheduler to generate DAGs and a dynamic runtime where executors coordinate via atomic dependency counters in a Key-Value Store. To minimize data movement, WUKONG implements three specific heuristics:

- **Fan-Out Clustering:** Executes multiple downstream tasks on the same worker if the upstream task produces large outputs, avoiding fan-out overhead.
- **Fan-In Clustering:** Allows a worker to execute a fan-in task locally if dependencies are met during data upload, preventing unnecessary downloads.
- **Delayed I/O:** Defers writing large results to external storage, holding data in memory to potentially pass it directly to a downstream task.

While effective, WUKONG relies on **"one-step scheduling"**, making decisions based solely on the immediate workflow stage. It fails to leverage global workflow structure for

planning and assumes a homogeneous resource environment. Consequently, its heuristics enforce a rigid trade-off: maximizing locality often leads to high resource contention. Our work addresses this by using historical metadata to enable predictive, non-uniform scheduling that balances locality with resource efficiency.

VI. Conclusion

A. Achievements

This work explored a novel scheduling approach for serverless workflows, leveraging historical task metrics to **reduce makespan** and **improve resource efficiency**. We designed and evaluated a decentralized workflow engine integrating predictive planning with global knowledge of workflow structure.

Evaluation across multiple workflows showed that our planners consistently outperformed **WUKONG**. The optimized **Non-Uniform** planner achieved the shortest makespan (approximately **63% faster** than optimized WUKONG) while the **Uniform** planner proved to be the most resource-efficient, consuming about **36% fewer GB-seconds**. These improvements stem from effective task co-location strategies and the proposed **pre-warming** and **pre-loading** optimizations, which reduced both worker startup delays and data waiting times.

The solution's modular architecture, separating **prediction**, **planning**, and **execution** layers, provides a solid foundation for future research and is available at *https://github.com/diogojesusdev/octoflows*.

B. Limitations and Future Work

Limitations of our solution include unbounded historical data growth, potential conflicts in optimizations, and limited error handling. Future work could explore more aggressive strategies, such as *task duplication*, support for dynamic fan-outs, richer user control, and interactive dashboards. Another promising direction lies in adapting to potential platform evolutions, such as FaaS platforms supporting **decoupled memory and CPU configurations**. Allowing these resources to be configured independently would enhance efficiency and enable **finer-grained pricing models**. While this introduces complexity in **scheduling, resource allocation, and locality optimization** for the provider, our solution would **naturally extend** to such environments, as its design is not constrained by specific resource coupling. Nevertheless, the increased dimensionality of possible resource configurations would significantly expand the prediction search space, potentially leading to slower planning times.

Finally, broader evaluation (comparing prediction strategies, deploying on commercial FaaS platforms, or even on edge computing environments [27], [28], and testing complex real-world workflows) would further validate the approach and its practical applicability.

References

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

[5] 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.

[6] 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.

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

[8] 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/

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

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

[16] 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

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

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

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

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

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

[22] 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

[23] 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.

[24] 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.

[25] 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

[26] 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.

[27] 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

[28] 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