Serverless Dataflows

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

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 [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.

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* [3], particularly for short-lived functions, affecting overall workflow performance. The lack of *direct inter-function communication* [4] 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[4], Azure Durable Functions[5], and Google Cloud Workflows[6]) 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 [5], Palette [6], Lambdata [7]) or entirely new serverless architectures (e.g., Apache OpenWhisk [8]). Finally, some workflow-focused solutions (e.g., WUKONG [9], Unum [10], DEWEv3 [11]) 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.

These workflow-focused approaches, however, often rely on "one-step scheduling," making decisions based solely on the immediate workflow stage without considering the broader context or the downstream effects on their decisions. This limitation motivates the central research question of this work: if we have knowledge of the computation steps, 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

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

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

[3] https://cloud.google.com/functions

[4] https://aws.amazon.com/pt/step-functions/

[5] https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=in-process%2Cnodejs-v3%2Cv1-model&pivots=csharp

[6] https://cloud.google.com/workflows

minimize makespan 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, thereby improving efficiency and reducing overhead.

The main contributions of this work are as follows:

- Analysis of the serverless workflow orchestration research landscape;
- Propose a decentralized serverless workflow execution engine that overcomes the ”one-step scheduling” limitation of existing workflow-focused solutions by leveraging historical metadata to generate informed execution plans;
- Demonstrate how incorporating historical execution data can improve task allocation, reduce reliance on external cloud storage services, and enhance overall workflow efficiency on FaaS platforms.

II. Related Work

A. *Traditional Workflow Scheduling*

Traditionally, executing computation workflows has relied on distributed data processing frameworks designed to manage computation and data across clusters of machines. Frameworks such as Hadoop [12], Apache Spark [13], and Apache Flink [14] provide abstractions for parallel and distributed execution, enabling efficient coordination, scheduling, and data movement across clusters. A key programming model that influenced these systems is MapReduce [15], which allows developers to process massive datasets by implementing simple `map` and `reduce` operations, with canonical example jobs like WordCount. Spark and Flink extend this model with in-memory processing and stream-oriented computation. More recently, Dask [16] has emerged as a flexible Python-based framework that enables parallel and distributed execution of complex task graphs. Unlike traditional MapReduce-style frameworks, Dask supports a more generic programming model that goes beyond simple map and reduce operations, allowing execution of heterogeneous tasks across clusters while maintaining ease of integration with Python libraries. Dask is particularly well-suited for data science applications due to its seamless integration with NumPy, Pandas, and other libraries, and its ability to handle large-scale mathematical computations efficiently and in a distributed manner, through Dask Distributed [17].

Collectively, these frameworks rely on clusters of machines to distribute computation and manage data, providing scalability and parallelism for large-scale workflows. They offer several advantages, including control over the underlying system, easier to achieve better data locality, efficient use of dedicated resources, and the flexibility to fine-tune scheduling and execution policies. These features make them well-suited for complex, resource-intensive workloads where predictable performance is critical. However, the cluster-based approach also introduces overheads and has inherent limitations related to ease-of-use, deployment, resource provisioning, and slow resource scaling, which can limit performance and efficiency, particularly for dynamic and embarrassingly parallel workflows.

B. *Cloud-Native Workflow Scheduling*

Traditional cluster-based frameworks often require significant expertise to configure, deploy, and maintain. Cloud-native workflow solutions provide a higher-level, managed approach, enabling developers to orchestrate workflows without worrying as much about underlying infrastructure. These platforms simplify deployment, and automatically handle resource allocation, fault tolerance, and state management.

Prominent commercial serverless workflow platforms, also referred to as **stateful functions**, include AWS Step Functions [4], Azure Durable Functions [5], and Google Cloud Workflows [6]. These solutions allow the user to create workflows by composing *stateless* functions together. The platform is then responsible for the orchestration part, managing workflow state, intermediate results, and fault tolerance without requiring developers to implement state management themselves. These platforms typically employ *checkpointing* techniques to persist workflow state, allowing the orchestrator or stateful function to pause while waiting for stages of the workflow to complete and then resume execution to trigger subsequent stages. Checkpointing also ensures that workflow execution can recover correctly after failures.

This orchestration and management capability is typically billed separately, reflecting both the convenience and reliability it provides and the additional resources required to maintain workflow state and fault tolerance. The main differences among these platforms lie in their execution model, workflow definition, and target use cases. Both AWS Step Functions and Google Cloud Workflows use JSON to represent workflow state machines, while Azure Durable Functions provides a more flexible programming model, where workflows are defined in code. One advantage of using such commercial services is that integration with other cloud services is easier, as they are tightly integrated with the AWS, Azure, and Google Cloud ecosystems.

In addition, there are also open-source workflow orchestration projects that run on general-purpose cloud or on-premises infrastructure, combatting vendor lock-in by providing better interoperability and flexibility. Apache Airflow [18] is a widely used example, allowing workflows to easily be defined in Python code. It provides fine-grained scheduling, monitoring, and retry policies, being mostly used for ETL [7] processes, machine learning workflows, and batch data processing. Another open-source alternative is Luigi [8], also

[7] https://pt.wikipedia.org/wiki/Extract,_transform,_load

[8] https://github.com/spotify/luigi

a Python-based workflow framework, developed and used at Spotify for managing complex batch data pipelines.

C. 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** [6] introduces locality "hints" to co-locate related function invocations on the same worker. **Faa$T** [5] provides a transparent, auto-scaling distributed cache for serverless functions. **Lambdata** [7] 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.

D. Serverless Workflow Scheduling

Several frameworks have been designed to execute workflows efficiently on unmodified serverless infrastructure. **PyWren** [19] is an early orchestrator for embarrassingly parallel computations, but its simple design can be inefficient for more complex workflows. **Unum** [10] 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** [9], a decentralized DAG engine that uses static scheduling and runtime optimizations to minimize data movement. WUKONG's 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.

E. 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 WUKONG's 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.

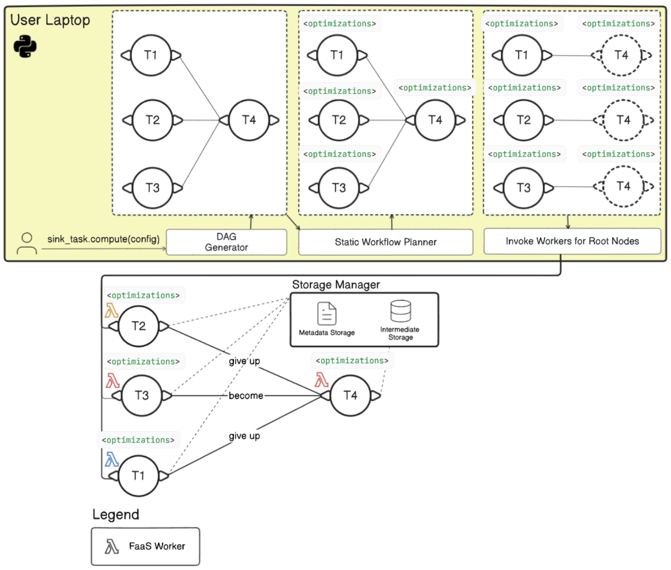

Fig. 1: Solution Architecture

1) **Metadata Management**: Responsible for collecting and storing task metadata from previous executions. It also uses this metadata to provide predictions regarding task execution times, data transfer times, task output sizes, and worker startup times;
2) **Static Workflow Planning**: Receives the entire workflow, represented as a Directed Acyclic Graph (DAG), and a ”planner” (an algorithm chosen by the user). This planner will use the predictions provided by Metadata Management to create a static plan/schedule to be followed by the workers;
3) **Scheduling**: This component is integrated into the workers, and it is responsible for executing the plan generated by the Static Workflow Planning layer, applying optimizations and delegating tasks as needed.

There are 3 distinct computational entities involved in this system:

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

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.

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

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

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

Fig. 2: DAG definition example

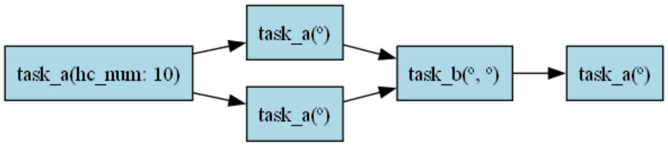

Fig. 3: Simple DAG example

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

One limitation of this DAG definition language is that it doesn't support ”dynamic fan-outs” (e.g., creating a variable number of tasks depending on the result of another task).

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

```
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

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 Planning

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 [6], but in our case this assignment is mandatory rather than advisory, giving strict control over execution locality. 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 [9] scheduling. We refer to these workers as *"flexible workers"*.

Users can select from a 3 provided algorithms or implement their own planner by implementing an interface. All planners have access to the predictions API as well as the workflow simulation. The basic planners the user can choose from are the following:

1) **Simple**: All tasks will use the same worker configuration (specified by the user) and be executed by *"Flexible workers"*. This is a more dynamic scheduling approach where tasks aren't tied to specific workers;
2) **Uniform**: All tasks will use the same worker configuration (specified by the user). Fixed worker IDs are assigned to tasks. Applies `task-dup` optimization to suitable tasks and simulates the potential benefits of using `pre-load` optimizations on tasks that are on the critical path;
3) **Non-Uniform**: All tasks will use different worker configurations (list of available resources is specified by the user). Fixed worker IDs are assigned to tasks. 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. Then, similarly to the Uniform planner, it applies `task-dup` optimization to suitable tasks and simulates the potential benefits of using `pre-load` optimizations on tasks that are on the critical path. It also applies `pre-warm` optimizations to suitable tasks.

Planners can use different optimizations to achieve their goals, whether it's extracting maximum performance or minimizing resource utilization. With the information they have access to, these algorithms 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, and then executing the task, against the alternative of simply executing the task on the current, less powerful worker.

Similarly to planners, users can also create new optimizations and define how workers should react to them. The provided optimizations are **pre-warm**, **pre-load** and **task-dup**.

pre-warm(worker_config):

- *Interpretation*: Tasks/Nodes with this optimization should perform a special invocation to the FaaS gateway that forces it to launch a new worker with the specified resource configuration `worker_config`. This can be used to warm up workers ahead of time and mask cold start latencies;
- *Assignment Logic*: For the nodes whose workers are expected to have cold starts, find the "optimal" node to perform the pre-warming by searching for nodes/tasks whose activity timing falls within a window (goal: avoid the pre-warmed worker from going cold before needed, while also not being warm too late). The optimization will be added to the "optimal" node, which will be responsible for doing the special "empty invocation" to the FaaS gateway.

pre-load:

- *Interpretation*: Workers assigned to tasks or nodes with this optimization should begin downloading the task's dependencies as early as possible. This prevents scenarios where a worker must fetch all dependencies at once. The optimization is effective only if the worker is active before executing the task, allowing it to download dependencies in parallel with other ongoing tasks. This is implemented by having the worker receive completion notifications from the *Metadata Storage* for all tasks upstream of the optimized task;
- *Assignment Logic*: This is an iterative process that optimizes the workflow along its critical path. First, the algorithm identifies the critical path and assigns the optimization to eligible nodes on it. The critical path is then recalculated: if the total execution time increases, the optimization is removed; if the execution time decreases but the critical path changes, the algorithm restarts with the new path. This process repeats until no further improvements are possible, or the algorithm hits a fixed iteration limit.

task-dup [Task Duplication]:

- *Interpretation*: Tasks or nodes with this optimization can be executed by other workers if doing so helps unlock dependent tasks more quickly. The task could be "duplicated" by workers that depend on its output. It is a trade-off between performance and resource utilization, allowing potentially faster execution at the cost of using additional compute resources;
- *Assignment Logic*: Assigned to all nodes whose execution time and input size do not exceed predefined thresholds. Whether duplication actually occurs is decided at runtime. The optimization targets fast tasks with small inputs, as these are inexpensive to duplicate in terms of both downloading dependencies and execution. This way, even if duplication turns out to be unnecessary, the impact on performance and resource usage remains minimal.

For simplicity, all planners currently use the same optimization assignment logic, so both *Uniform* and *Non-Uniform* planners apply identical criteria when assigning optimizations to tasks. However, this is not a strict requirement—different planners could adopt their own assignment strategies.

Because planners may sometimes lack sufficient information to make optimal decisions about optimization assignments, it is important to allow users to specify optimizations for specific tasks manually. An example of this feature is shown in Listing 2, where the user requests that `task_b` attempt to *pre-load* its dependencies.

Once these optimizations are assigned, workflow planning is complete, and workers can begin execution. Because planning occurs on the user's machine, 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.

TODO: show example of planned workflow: IMAGE and describe it

E. 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_id` and the `task_ids` of the tasks it should execute first. Using this information, it retrieves the DAG structure and execution plan from *Metadata Storage*. 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 that, the worker starts executing the 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;
2) **Executing Task**: Execute the task. Tasks' code is stored in a serialized/pickled format (using cloudpickle[9]) and deserialized and executed by the workers. This enables the worker to remain generic, capable of receiving and executing arbitrary task code;
3) **Handling Output**: This phase is responsible for evaluating whether it's necessary to upload the task's output to storage and emitting a `TASK_COMPLETED` event;
4) **Delegating Downstream Tasks**:
5) **Delegating Downstream Tasks**: For each downstream task, the worker performs an `atomic_increment_and_get()` operation on a "dependency counter" (inspired by WUKONG [9]) stored in *Metadata Storage*, which tracks how many dependencies of a task have been satisfied. When the counter indicates that all dependencies for a downstream task have been satisfied, the worker emits a `TASK_READY` event for that task. The worker then consults the execution plan to determine how to proceed for each downstream task unlocked: if the unlocked task is assigned to another worker, a `TASK_READY` event is emitted; if the unlocked task is assigned to the same worker and has no remaining dependencies, the worker immediately continues the cycle by executing it.

To illustrate how workers handle downstream tasks, Figure 6 presents an example of choreographed scheduling with three workers (A, B, and C) and seven tasks (T1-T7). In this example, once Task T1 completes on Worker B, the worker inspects the dependency counters for tasks T3, T4, and T5. It determines that T3 and T4 are ready to run, while T5 is still pending because Task T2 has not yet completed. Worker B then launches a new worker (C) to execute T3 and proceeds to execute T4 itself. Later, when T2 finishes on Worker A, all dependencies of T5 are satisfied, prompting Worker A to execute it directly.

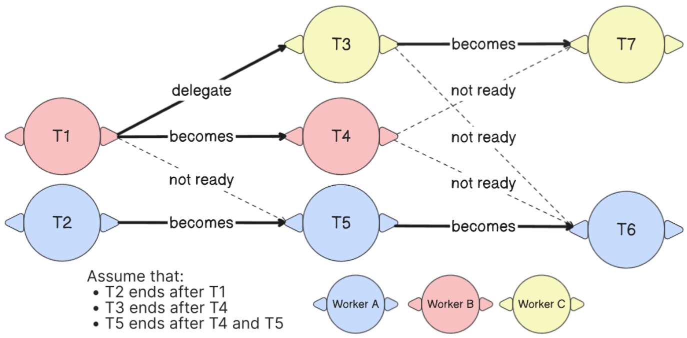

Fig. 6: Choreographed Scheduling Example

A workflow is considered complete once the output of the final (sink) task is available in storage. The worker that uploads this final result is also responsible for cleaning up all intermediate results before shutting down. Meanwhile, after submitting the workflow, the user's machine subscribes to the `TASK_COMPLETED` event for the sink task; upon receiving this notification, it retrieves the final result from *Intermediate Storage*.

By delegating downstream tasks to workers, our approach eliminates the need for a central scheduler, a common component in many existing FaaS-based workflow engines. This decentralization increases flexibility and scalability, as workers can dynamically invoke additional workers as needed, following a choreographed execution model.

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.

IV. Evaluation and Analysis

A. Testing Environment

TODO: explain Docker simulating faas environment

B. Testing Configurations

TODO: worker resources, workflows, planners, SLAs

C. Results

D. Analysis

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] 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.

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

[5] 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

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

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

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

[15] 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

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

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

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

[19] 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.