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, a state-of-the-art decentralized serverless DAG engine. Our evaluation shows that our most resource-efficient planner reduces workflow execution time by 12.6% and simultaneously lowers resource consumption by 36% compared to the optimized WUKONG scheduling approach. For users prioritizing speed over cost, our fastest planner (Non-Uniform optimized) achieves a 57.5% reduction in makespan compared to our most resource-efficient planner (Uniform), but this performance gain requires a 114% increase in resource consumption.

***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 [4] seismic hazard analysis or Montage [5], an astronomy image mosaicking workflow. These applications are structured as workflows, formally represented as Directed Acyclic Graphs (DAGs) of interdependent tasks. However, efficiently executing these complex workflows on serverless platforms remains a significant challenge.

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

Several solutions have emerged to address the limitations of serverless platforms. Stateful functions (e.g., AWS Step Functions [8], Azure Durable Functions [9], and Google Cloud Workflows [10]) expand the range of applications that can run on serverless platforms by maintaining state across multiple function invocations, coordinating complex workflows, and providing built-in fault tolerance. Other approaches tackle limitations at the runtime level, proposing extensions to FaaS platforms (e.g., Faa$T [11], Palette [12], Lambdata [13]) or entirely new serverless architectures (e.g., Apache OpenWhisk [14]). Finally, some workflow-focused solutions (e.g., WUKONG [15], Unum [16], DEWEv3 [17]) 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 use uniform resources for workers and rely on "one-step scheduling," making decisions based solely on the immediate workflow stage without considering the broader context or the downstream effects of their decisions. This combination of homogeneous worker configurations and limited scheduling foresight can lead to inefficient use of resources when tasks have diverse computational or memory requirements. Furthermore, the heuristic-based approaches used by other solutions can be inefficient in certain scenarios, as they lack mechanisms to adapt worker resource allocations to the specific needs of individual tasks. Moreover, we found no prior work that leverages metadata or historical metrics to inform scheduling decisions across an entire serverless workflow.

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 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, while also avoiding the inefficiencies of homogeneous worker resource allocations.

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” and uniform-resource limitations of existing workflow-focused solutions by leveraging historical metadata to generate informed execution plans;
- Demonstrate how incorporating historical execution data can be used to improve task allocation (balancing locality and resource contention), reducing reliance on external cloud storage services and enhancing overall workflow efficiency on FaaS platforms.

II. 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 do not allow 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. Workflow Definition Language

We will now present our DAG-based workflow engine 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’s way of expressing workflows: the user can create 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 DAG by passing their results as arguments to the next task. The resulting workflow structure is illustrated in Figure 1.

Listing 1: DAG definition example

```python
# 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)
```

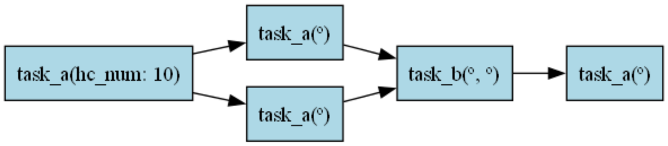

Fig. 1: Simple DAG example

It is important to note that 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, but it is not currently supported. In future iterations, this limitation could be addressed through **code instrumentation**, wherein storage access APIs would be intercepted to record metrics such as the volume of data transferred and the corresponding I/O latency.

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

Listing 2: Setting up and launching workflow execution

```python
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=UniformPlanner.Config(
            sla=sla,
            worker_resource_configuration=TaskWorkerResourceConfiguration(cpus=3, memory_mb=512),
            optimizations=[PreLoadOptimization, PreWarmOptimization]
        )
    )
)
```

One limitation of this DAG definition language is that it doesn't support "dynamic fan-outs" (e.g., creating a variable number of tasks depending on the result of another task) on a single workflow. This is a powerful and expressive feature, but that is seldom supported in other DAG definition languages (e.g., Dask [18], WUKONG [15], Unum [16], Oozie [19] 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

The overall architecture and logical flow of our decentralized serverless workflow execution engine is organized into 3 high-level layers. Figure 2 provides an overview of this architecture. The upper part represents the components that run on the user's machine, while the lower part represents the components that run outside the user's machine.

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

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

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

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

Storing these metrics enables us to provide a prediction API, shown in Listing 3. To improve accuracy, metrics are kept separate for each workflow. As a result, even if two workflows use the same function or task code, their metrics are stored independently. This design choice reflects our assumption that different workflows may follow different execution patterns. To avoid introducing runtime overhead, metrics are batched and uploaded when the worker shuts down.

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

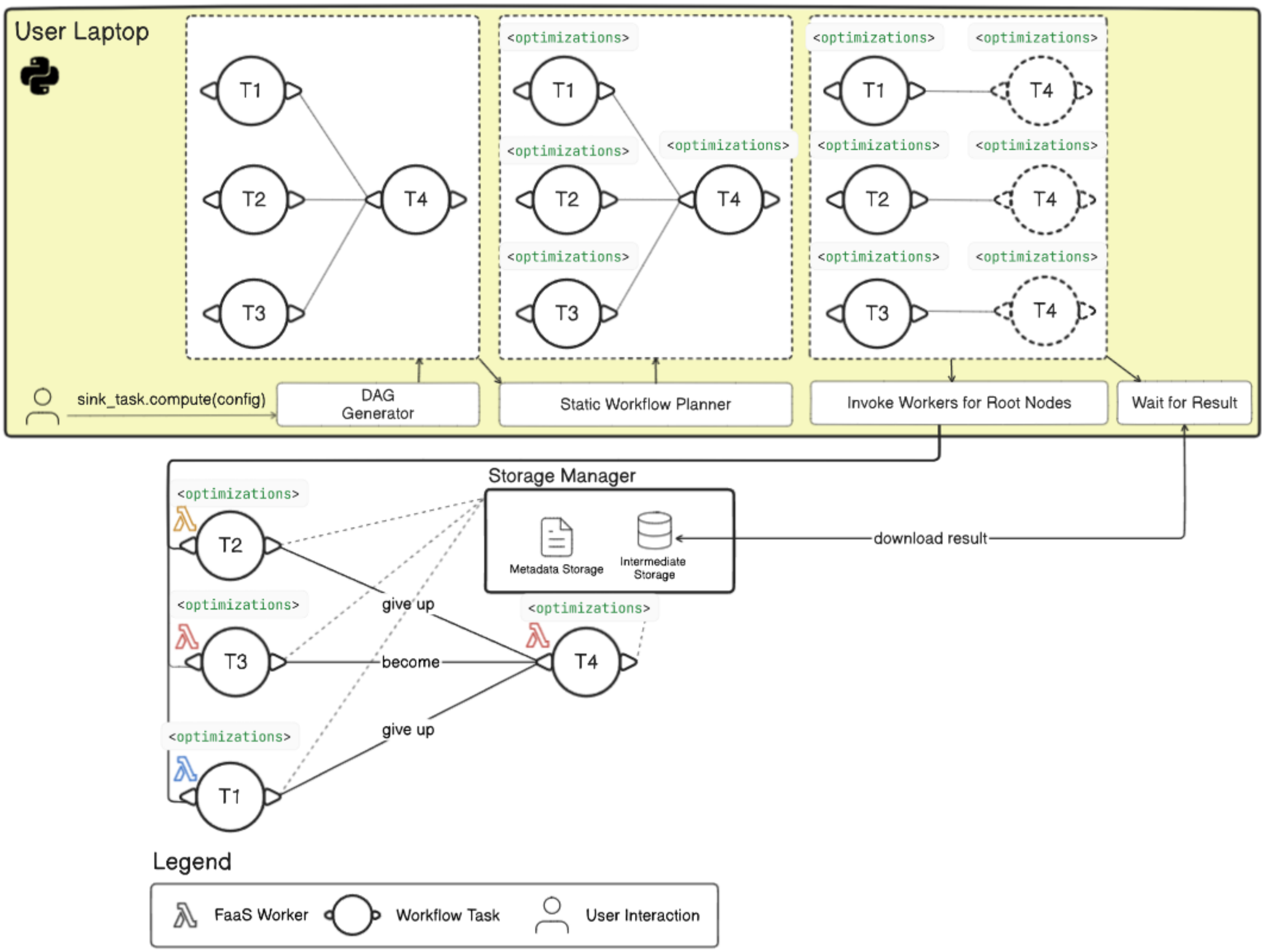

Fig. 2: Solution Architecture

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.

Listing 3: Task Predictions API

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

    def predict_worker_startup_time(state: ’cold’ | ’warm’, resource_config, sla) -> float

    def predict_data_transfer_time(data_size_bytes, resource_config, sla) -> float

    def predict_execution_time(task_name, input_size, resource_config, sla) -> float
```

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 filtering samples we use an algorithm that selects a limited number of the most relevant samples for each prediction. This algorithm works by gradually widening a tolerance window around the reference value until it finds enough nearby samples. Within each window, it balances samples that are smaller, larger, or exactly equal to the reference, giving preference to the closest ones. If there still aren’t enough candidates, it falls back to simply picking the nearest available samples overall. This way, the algorithm adapts to the data while keeping the selection both relevant and limited in size.

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 2). Its job is to execute the planning algorithm, providing it access to the 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 [12], but in our case this assignment is mandatory rather than advisory, giving strict control over execution locality. Two tasks assigned the same `worker_id` should be executed on the same worker instance. If `worker_id` is not specified, workers will, at run-time, have to decide whether to execute or delegate those tasks, similar to WUKONG's [15] 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 more dynamic scheduling approach where tasks aren't tied to specific workers that try to reproduce WUKONG's scheduling behavior;
2) **Uniform**: Tasks share a common worker configuration specified by the user, with each task assigned a `worker_id` to allow for co-location of tasks.
3) **Non-Uniform**: Tasks can use different worker configurations (list of available resources is specified by the user). Each task is assigned a `worker_id`. This algorithm starts by assigning the best available resources to all tasks. Then it runs a resource downgrading algorithm that attempts to downgrade resources of workers *outside the critical path* as much as possible without introducing a new critical path.

Both the **Uniform** and **Non-Uniform** planners follow a two-phase approach for task allocation: resource configuration assignment followed by worker ID assignment. The planners differ in their resource allocation strategies. The **Uniform planner** applies a single, user-specified CPU and memory configuration to all tasks, while the **Non-Uniform planner** selects the most powerful configuration from the user-specified options for each task. After resource configuration, both planners employ the logic detailed in Algorithm 1 (in Appendix) for worker ID assignment. This algorithm implements a balanced clustering strategy that 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**.

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
        $roots \leftarrow \{r \in nodes \mid r.upstream = \emptyset \land r \notin assigned\}$
        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
```

After this, the **Non-Uniform** planner runs an additional algorithm, shown in Algorithm 3, that attempts to downgrade resources of workers *outside the critical path* as much as possible without introducing a new critical path, by iteratively simulating the effect of downgrading resources of workers *outside the critical path* with different configurations.

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

Aside from their `worker_id` and resource assignments, planners can also apply different **optimizations** to further improve the workflow execution. The optimizations to be used are selected by the user, as shown in Listing 2. Similarly to planners, we provide two optimizations: **pre-warm** and **pre-load**, but it is also possible to create new optimizations and define how workers should react to them. Now, we will describe the two base optimizations and how they are assigned to tasks:

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($\{t \in tasks \mid exec\_t[t] \le median\}$)

                    ▷ 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
```

Algorithm 3 Resource Downgrading Algorithm (used by Non-Uniform planner)

```
Require: $dag$, $nodes$, $critical\_path\_ids$, $original\_cp\_time$, $configs$
$workers\_outside \leftarrow \emptyset$

                    ▷ 1) Identify workers outside the critical path
for all $n \in nodes$ do                    ▷ nodes are topologically sorted
    $wid \leftarrow n.worker\_id$
    if $n.id \notin critical\_path\_ids \wedge \forall cp \in dag.cp\_nodes : wid \neq cp.worker\_id$ then
        $workers\_outside \leftarrow workers\_outside \cup \{wid\}$
    end if
end for
$nodes\_outside\_cp \leftarrow \{n \in nodes \mid n.id \notin critical\_path\_ids\}$

    ▷ 2) Attempt downgrade for each worker outside critical path
for all $wid \in workers\_outside$ do
    $last\_acceptable\_rc \leftarrow \{n.id : n.config \mid n \in nodes\_outside\_cp \wedge n.worker\_id = wid\}$

                    ▷ Iterate through weaker configurations (skip best)
    for $i \leftarrow 1$ to $|configs| - 1$ do
        $trial \leftarrow configs[i].$Clone($wid$)

                    ▷ Apply trial configuration to all nodes of this worker
        for all $n \in nodes\_outside\_cp$ do
            if $n.worker\_id = wid$ then
                $n.config \leftarrow trial$
            end if
        end for

                    ▷ Recompute workflow timing with predictions
        $cp\_time \leftarrow$ SimulateCriticalPathTime($dag$)

        if $cp\_time = original\_cp\_time$ then
                                        ▷ Good Downgrade
            for all $n \in nodes\_outside\_cp$ do
                if $n.worker\_id = wid$ then
                    $last\_acceptable\_rc[n.id] \leftarrow n.config$
                end if
            end for
        else
                                        ▷ Bad Downgrade, revert
            for all $n \in nodes\_outside\_cp$ do
                if $n.worker\_id = wid$ then
                    $n.config \leftarrow last\_acceptable\_rc[n.id]$
                end if
            end for
            break                       ▷ move to next worker
        end if
    end for
end for
```

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

- *Interpretation*: Tasks/Nodes with this optimization should perform a special invocation to the FaaS gateway that forces it to launch a new worker with the specified resource configuration `worker_config`. This can be used to warm up workers ahead of time and mask cold start latencies.
- *Assignment Logic*: For workers that are predicted to experience a cold start, the algorithm identifies an optimal worker to perform the pre-warming action. It searches for a task whose execution window aligns with a calculated pre-warming interval. This process balances two objectives: ensuring the pre-warmed worker does not go cold before it is required, while also not being warmed too late to be

effective.

- *Integration with Task Handling Logic*: The optimization is attached to the identified optimal worker's task and includes a `delay_s` parameter. As soon as this optimal worker begins its own execution, it launches a concurrent Python coroutine. This coroutine waits for the specified delay before making a special "empty invocation" to the FaaS gateway, which in turn initializes the target worker.

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

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

Because planners may sometimes lack sufficient information to make optimal decisions about optimization assignments, it is important to not only allow the user to select optimizations at the workflow-level, but also allow them the flexibility to specify optimizations at the *task-level*. An example of this feature is shown in Listing 1, where the user requests that `task_b` attempt to use the *pre-load* optimization.

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

To illustrate this execution model, Figure 3 provides a visual trace of how a planned workflow would be executed. The diagram depicts the workflow with the optimizations and `worker_id` assignments for each task. The non-dashed arrows represent task dependencies, while the dashed arrows represent interactions with the *Intermediate Storage* to either upload or download task data. We can see that task outputs are only uploaded to storage when there is at least one downstream task that depends on it and is assigned to another worker.

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

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

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

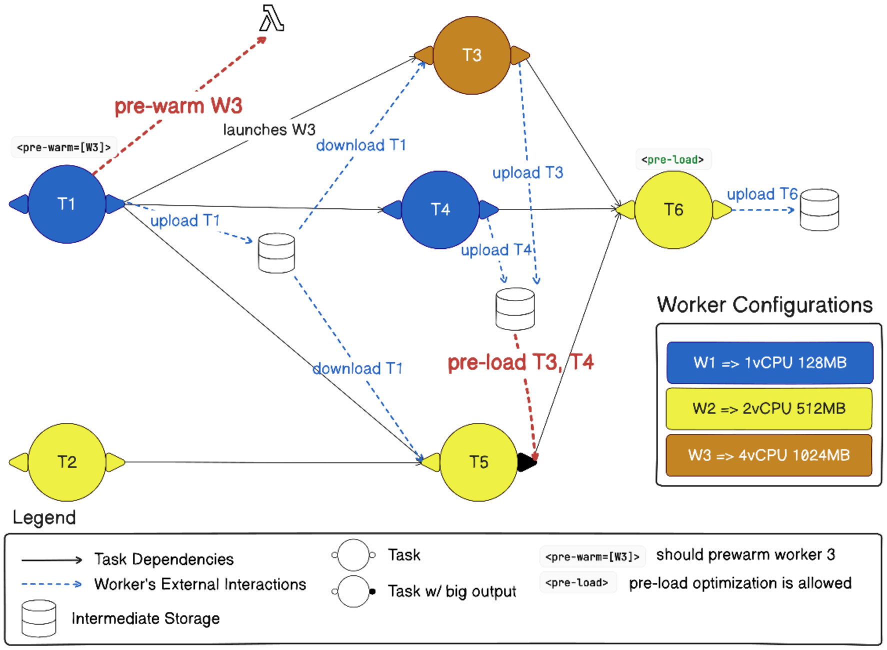

Fig. 3: Planned Workflow Execution Example

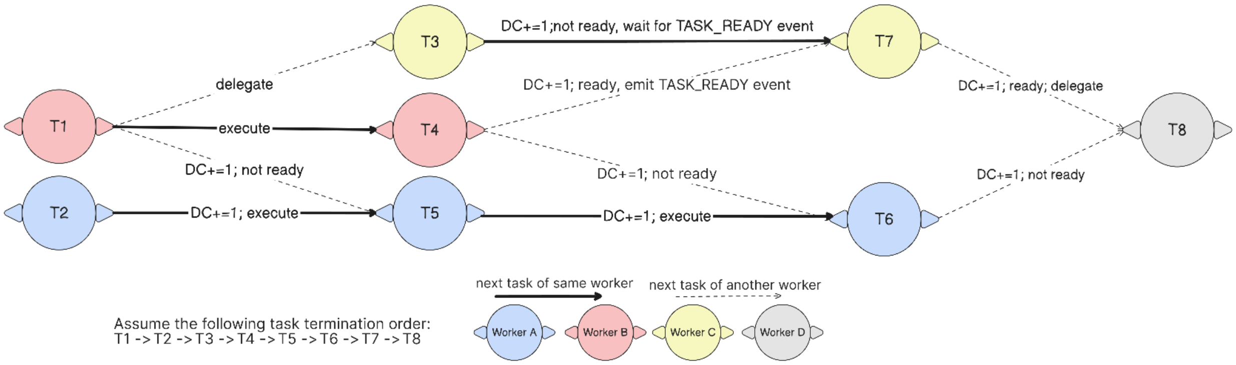
T1
T2
T3
T4
T5
T6
T7
T8
delegate
execute
DC+=1; not ready
DC+=1; execute
DC+=1;not ready, wait for TASK_READY event
DC+=1; ready, emit TASK_READY event
DC+=1; not ready
DC+=1; execute
DC+=1; ready; delegate
DC+=1; not ready
next task of same worker
next task of another worker
Worker A
Worker B
Worker C
Worker D
Assume the following task termination order:
T1 -> T2 -> T3 -> T4 -> T5 -> T6 -> T7 -> T8

Fig. 4: Choreographed Scheduling Example

the main thread, where the other coroutines are running: both task branch execution and other internal coroutines (i.e., Redis pub/sub listener, and delayed `pre-warm` requests);

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

4) **Updating Dependency Counters**: For each downstream task, the worker performs an *atomic increment and get* operation on a *"Dependency Counter"* (inspired by WUKONG [15]) stored in *Metadata Storage*, which tracks how many dependencies of a task have been satisfied. If the counter sees that the value returned is the same as the number of dependencies for a downstream task, the worker emits a `TASK_READY` event for that task, signaling other workers or workers that aren't active yet that the task became ready to execute;

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

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

We will now go over a practical example of how we use these events are used to coordinate decentralized scheduling. Figure 4 presents an example of a workflow with four workers (A, B, C, and D) and eight tasks (T1-T8).

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

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

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

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 delegates downstream task scheduling to 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, the approach minimizes scheduling latency and improves scalability, which primarily depends on the horizontal scalability of the underlying FaaS and storage layers. The client initiates 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.

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 and with greater observability.

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 help simulate cold starts more easily. 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 and performance analysis.

B. Research Questions

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

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

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

- **RQ1:** To what extent can historical metrics from previous workflow executions improve the accuracy of predicting serverless workflow behavior, specifically regarding execution time, container startup latency, data transfer performance, and function I/O characteristics?
- **RQ2:** What are the main advantages and limitations of applying prediction-based scheduling strategies in serverless workflow environments, compared to traditional reactive or static scheduling approaches? Can the proposed system achieve a lower *makespan* and reduced overall resource consumption compared to WUKONG [15], a decentralized serverless workflow engine that employs its own set of data-locality optimizations?
- **RQ3:** How effective are the proposed optimizations in practice? In particular, how much does `pre-load` contribute to hiding latency and enabling earlier task execution, and how beneficial is `pre-warming` in reducing cold-start delays and improving overall workflow performance?
- **RQ4:** How much performance improvement can be achieved by adopting a non-uniform worker resource allocation strategy, compared to a uniform scheduling approach that assigns identical resource configurations to all tasks? And what is the trade-off between the achieved performance gains and the additional resource consumption introduced by this adaptive allocation strategy?

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

- For each **task**:
  - Timestamp of when it started being handled (before executing its tasks);
  - Time to download all dependencies;
  - Size and time to download each of the dependencies;
  - Task execution time;
  - Size and time to upload output;
  - Optimization-specific metrics of all optimizations applied to a given task on a particular workflow instance.
- For each **workflow instance**:
  - Entire workflow plan, with worker resource and ID assignments, as well as optimizations;
  - Time at which user submitted the workflow;
  - Monitor Docker containers during the workflow, recording total run-time and memory allocation in GB-seconds, similar to AWS Lambda's cost calculation, which is based on memory (GB) * run-time (seconds).
- For each **worker**:
  - Resource Configuration (CPUs and Memory);
  - Time at which a worker invocation was made to the FaaS Gateway and time at which the worker script started executing (used to calculate worker startup latency);
  - Worker state, indicating whether the worker script was executed in a *cold* or *warm* environment. A cold start occurs when a new container must be created to run the worker, while a warm start reuses an existing, previously initialized container. The system determines this state using a file-based locking mechanism: during startup, the worker attempts an atomic *"create-if-not-exists"* operation on the file `/tmp/worker_startup.atomic`. If the file already exists, it implies that the container has been used before, and the current execution is therefore a warm start; otherwise, it is a cold start. When a container is *pre-warmed*, this file is proactively created during initialization, even though the worker script itself is not yet executed.

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

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.
- **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: median (50th percentile), 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

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

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

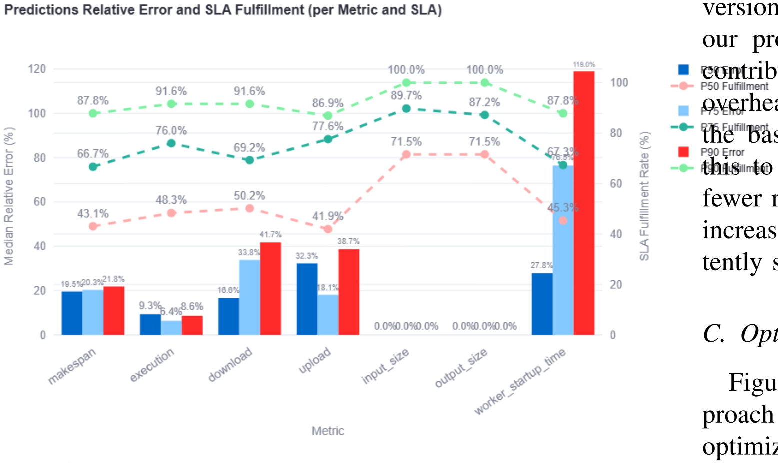

Fig. 5: Predictions accuracy across all planners and SLA fulfillment rates

Figure 5 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

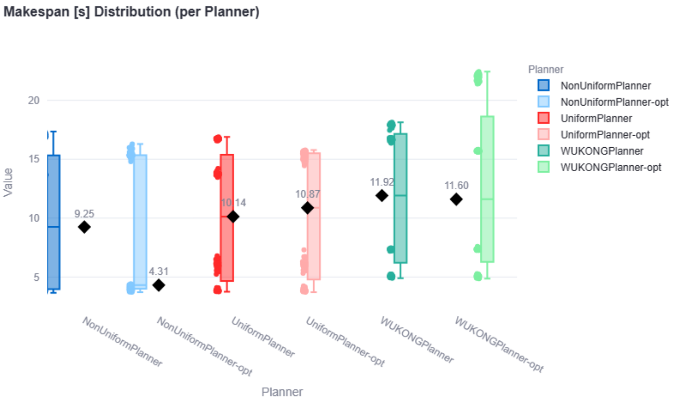

Fig. 6: *Makespan* distribution across all planners

As shown in Figure 6, 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

Figure 7 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.

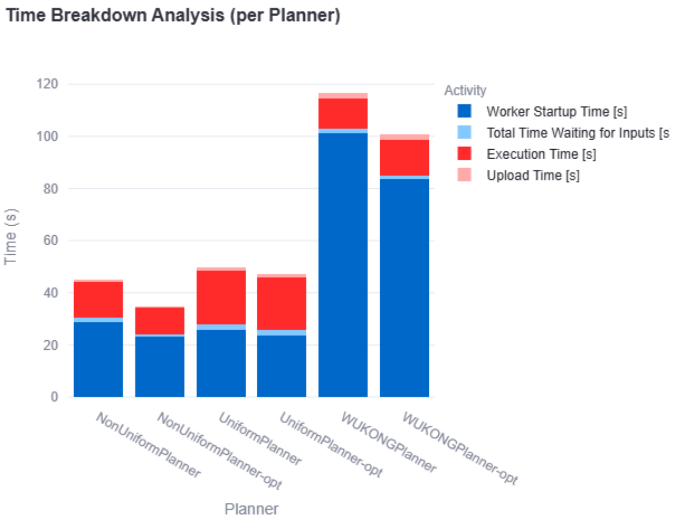

Fig. 7: Time breakdown analysis

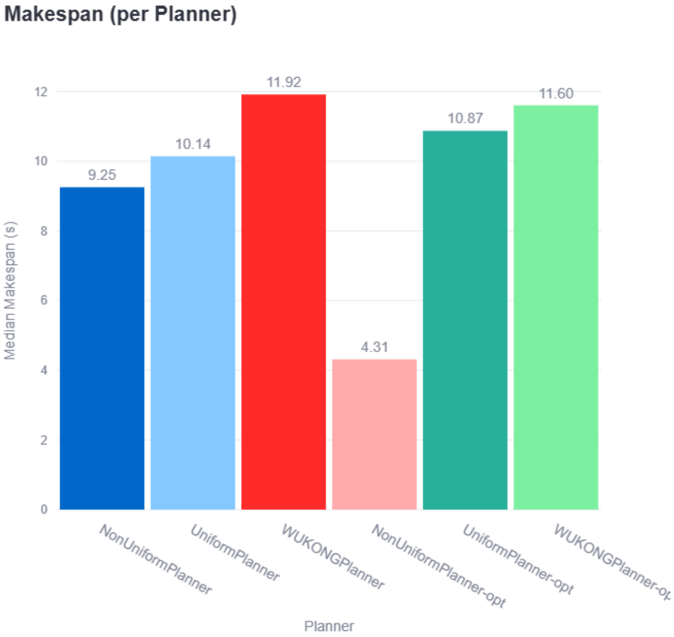

Fig. 8: *Makespan* comparison

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 8 and 9 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**.

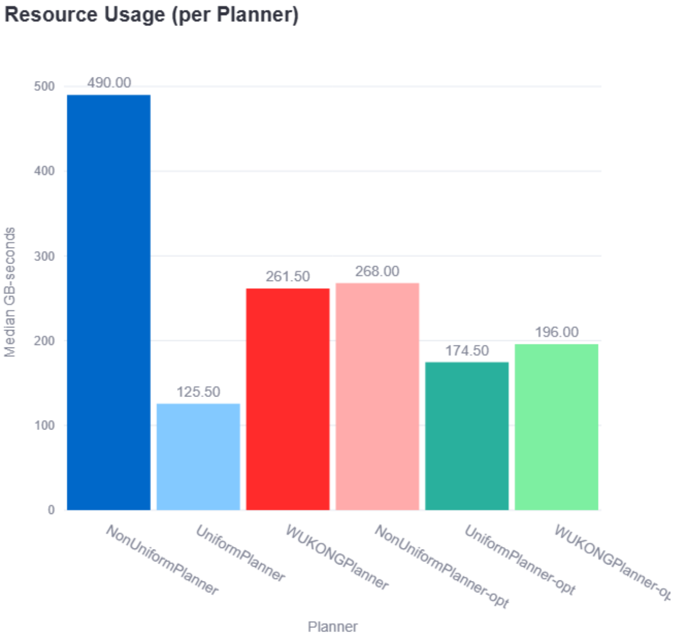

Fig. 9: Resource usage comparison

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 significantly 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

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

over **50%** shorter while consuming **45%** fewer resources than its unoptimized counterpart.

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 [20]–[23], enabling a path toward future dynamic scheduling.

In summary, this research validates 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 unparalleled performance in time-sensitive applications. Both strategies represent a significant advancement over existing scheduling models. Lastly, it shows the importance and positive impact of properly **balancing resource contention and data locality** in serverless workflows.

V. Related Work

This section highlights the most relevant related systems to our work, tracing the evolution from traditional cluster-based frameworks to modern cloud-native workflow orchestration platforms and serverless workflow engines. We highlight their key design principles, limitations, and the innovations that inspired our approach.

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 [24], Apache Spark [25], and Apache Flink [26] 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 [27], 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 [18] 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 [28].

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 [8], Azure Durable Functions [9], and Google Cloud Workflows [10]. 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 [29] is a widely used example, allowing workflows to easily be defined in Python code. Airflow has a rich ecosystem of pre-built integrations for databases, message queues, and major cloud services, simplifying the connection of workflows to diverse external services. It is mostly used for ETL [6] processes, machine learning workflows, and batch data processing.

C. FaaS Runtime Extensions

While commercial and open-source workflow platforms simplify orchestration and state management, they generally operate on top of standard FaaS runtimes and cannot fully address the inefficiencies inherent in serverless platforms, such as cold starts, limited inter-function communication, and lack of data locality. To address these limitations, a significant body of research has focused on modifying the underlying FaaS runtime or proposing extensions that improve data locality and scheduling efficiency. Palette [12], Faa$T [11], and Lambdata [13] are three prominent examples of such extensions.

1) Palette Load Balancing: Palette Load Balancing improves data locality by introducing the concept of *colors* as *locality hints*. These colors are parameters attached to function invocations, allowing users to express which invocations should be co-located on the same worker. Palette then attempts to route invocations with the same color to the same instance, enabling subsequent invocations to access locally cached data rather than fetching it from remote storage. Colors are treated as *hints*, so the system can ignore them if resource constraints require it. This approach improves cache hit ratios, reduces data transfer costs, gives enough flexibility to users or systems leveraging the runtime to define coloring policies with different strategies.

2) Faa$T: Faa$T provides a transparent, auto-scaling distributed cache for serverless functions. Each application has its own dedicated in-memory cache, called a *cachelet*, which stores data accessed during function executions and makes it available for subsequent calls. Faa$T's transparency is achieved by operating without requiring changes to application code, as it intercepts data requests and checks the cache before accessing remote storage. *Cachelets* collaborate to form a distributed cache using consistent hashing, dynamically scaling up or down based on application workload.

3) Lambdata: Lambdata focuses on optimizing data locality by using *user-provided data intents*. Developers explicitly annotate functions with *get_data* and *put_data* parameters, specifying which data objects the function will read or write. Lambdata then uses these annotations to make informed scheduling decisions such as co-locating functions that share data and leveraging local caches to minimize remote data transfers. This approach simplifies deployment and integration, as it does not require a distributed cache or complex runtime modifications, but relies on accurate intent declarations to achieve effective data reuse.

Together, these FaaS runtime extensions illustrate different strategies for improving data locality and execution efficiency in serverless environments.

D. Serverless Workflow Scheduling Execution Engines

Several frameworks have been designed to execute workflows efficiently on **unmodified** FaaS infrastructure, each offering distinct approaches to address the challenges of stateless, distributed computing. These solutions represent significant innovations in workflow orchestration, with varying trade-offs in terms of scalability, data locality, architectural complexity and ease-of-use and inspired our work. Here we will mention the most relevant workflow execution engines built to run on top of FaaS.

1) DEWE v3: DEWE v3 [17] introduces an innovative hybrid approach to serverless workflow orchestration that combines the best aspects of both serverless and serverful computing models. This hybrid workflow execution engine intelligently distributes tasks based on their characteristics: *short tasks* are run on FaaS workers while *longer tasks* run on virtual machines. The system employs a queue-based job distribution mechanism where jobs expected to complete within FaaS limits are published to a common job queue for serverless execution, while long-running jobs are directed to a separate queue for local, serverful execution on dedicated servers. Jobs that fail to execute on FaaS workers, for being longer than expected and exceeding execution limits imposed by the platform, are redirected to the serverful queue. This dual-execution model enables DEWE to accommodate workflows with diverse resource consumption patterns. This system proves particularly effective for scientific workflows, such as Montage [5], where task durations and resource requirements vary significantly. However, this hybrid approach introduces specific trade-offs. Latency-sensitive workflows may be slowed by job queuing overhead. In addition, hybrid deployments often lead to resource underutilization, as serverful workers may sit idle when most tasks are executed on FaaS. Finally, the centralized workflow manager can become a scalability bottleneck when handling many short tasks.

2) PyWren: PyWren [30], representing one of the pioneering pure serverless approaches, demonstrates the potential and limitations of leveraging unmodified serverless infrastructure for distributed computation. Built atop AWS Lambda, PyWren focuses on executing arbitrary Python functions as stateless serverless functions with minimal user management overhead, automatically handling function execution, dependencies, S3 bucket storage for serialized code and intermediate data. The system is ideal for embarrassingly parallel workloads, also known as *"bag-of-tasks"* scenarios, with many independent, parallel tasks such as simple data transformations, scientific simulations, parallel model training, and large-scale media processing. While PyWren's serverless orchestration model provides excellent scalability and removes the burden of infrastructure management, its simplicity limits its applicability. It is not well-suited for workflows with complex dependency structures or those that require sharing large intermediate results through object storage. Moreover, latency-sensitive applications are disadvantaged by function cold starts, since PyWren does not include mechanisms to mitigate their impact.

3) Unum: Unum [16] takes a radically different approach from the two previous solutions by decentralizing orchestration logic entirely, eliminating the need for a standalone orchestrator service. This application-level serverless workflow

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

orchestration system embeds orchestration logic directly into a library that wraps user-defined FaaS functions, leveraging an external scalable consistent data store for coordination during fan-ins and execution correctness. Unum introduces an Intermediate Representation (IR) to capture information about workflow progression (nearby tasks) and relies only on minimal, common serverless APIs (function invocation and basic data store operations) available across cloud platforms. This design choice provides exceptional portability and cost-effectiveness, as it can run on unmodified serverless infrastructure. Unum also can and compile workflows defined in languages from providers like AWS Step Functions and Google Cloud Workflows into its IR format. However, Unum's generic approach comes with trade-offs: it currently supports only statically defined control structures and cannot express workflows where the next step is determined dynamically at runtime, and it lacks data locality optimizations since it cannot force related tasks to execute on the same worker, with each function instance executing only its specific task before triggering the next function.

4) WUKONG: WUKONG [15] represents the most sophisticated approach among these solutions, designed as a decentralized locality-enhanced serverless workflow engine. WUKONG addresses the limitations of traditional serverful models like Dask Distributed while maximizing the advantages of serverless computing, focusing on improving scale-out speed and enhancing data locality to minimize large object movement. The system's architecture is divided into static components (operating before workflow execution) and dynamic components (operating during execution). The static scheduler includes a **DAG Generator** that converts Python code into DAGs (using Dask), a **Schedule Generator** that creates `n` static schedules for `n` root/leaf nodes (each containing every reachable task in a depth-first search starting at that node), **Initial Task Executor Invokers** that launches the first Lambda instances for each root task, and a **Process** on the client that waits for and downloads final results.

After receiving the initial schedules, FaaS workers (referenced to as *AWS Lambda Executors*) drive workflow execution. Workers execute tasks until they encounter a *fan-out*, at which point they transfer data to intermediate storage, execute 1 of the `N` fan-out tasks and invoke *N - 1 new* executors for the other tasks. Then, when they find a *fan-in*, the group of executors that reach the common fan-in node cooperate using a *dynamic scheduling* model to select only one executor to proceed. This coordination is managed through a shared **dependency counter** in a Key-Value Store (KVS). Each involved executor *atomically* updates this counter; the one whose update satisfies the final input dependency for the fan-in task will execute that task and continue along its static schedule. The other executors transfer their intermediate data to storage, and then stop their execution, decreasing the workflow parallelism.

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

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

While WUKONG represents a significant advance in serverless workflow orchestration through its decentralization, its scheduling and optimizations remain limited. The system bases decisions only on the next stage of the workflow (i.e., one-to-one, fan-in, or fan-out transitions). We refer to this as *one-step scheduling*, since it relies solely on information about the next step. Crucially, WUKONG does not exploit the global knowledge of the workflow structure, even though the entire structure is known before execution begins. Consequently, its optimizations rely on fixed, heuristic-based strategies that enforce an extreme trade-off between data locality and resource contention. Optimizations like *Task Clustering* and *Delayed I/O* achieve **maximum data locality** by having executors hold large objects and prolong execution, but this comes at the cost of **high resource contention**. Conversely, disabling these optimizations results in **no resource contention**, leading to **no data locality**. This highlights the absence of adaptive mechanisms to achieve a *dynamic balance between locality and resource utilization*, resulting in suboptimal performance when workload behavior deviates from expected patterns. With our solution we demonstrated that this dynamic balance is critical to improving scheduling efficiency.

E. Discussion

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

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

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/48302-DiogoJesus/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 [31], [32], 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] R. Graves, T. H. Jordan, S. Callaghan, E. Deelman, E. Field, G. Juve, C. Kesselman, P. Maechling, G. Mehta, K. Milner, D. Okaya, P. Small, and K. Vahi, "Cybershake: A physics-based seismic hazard model for southern california," *Pure and Applied Geophysics*, vol. 168, no. 3, pp. 367–381, 2011. [Online]. Available: https://doi.org/10.1007/s00024-010-0161-6

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

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

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

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

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

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

[11] F. Romero, G. I. Chaudhry, I. n. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "Faa$t: A transparent auto-scaling cache for serverless applications," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 122–137. [Online]. Available: https://doi.org/10.1145/3472883.3486974

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

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

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

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

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

[27] 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

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

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

[30] 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.

[31] 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

[32] 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

APPENDIX

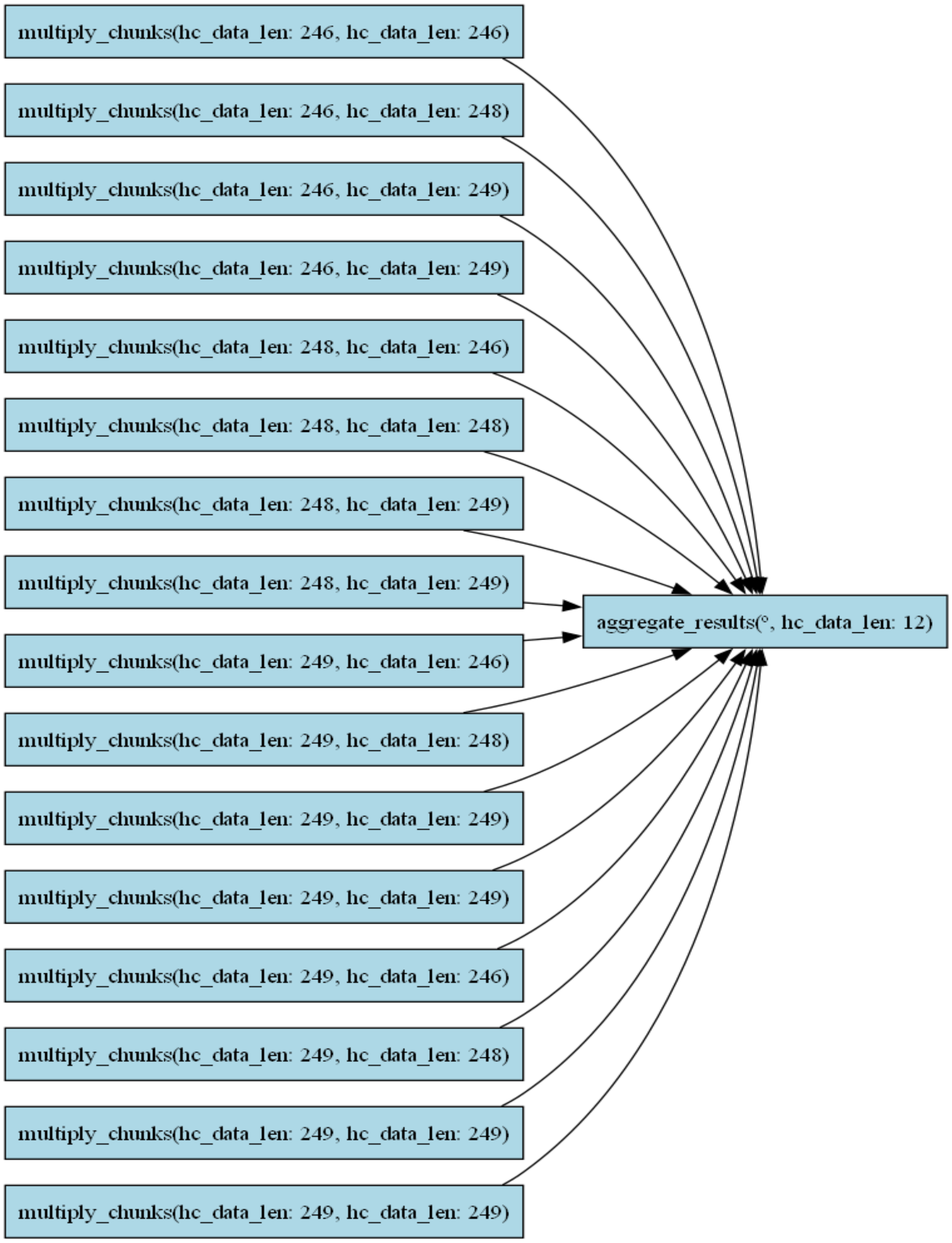
multiply_chunks(hc_data_len: 246, hc_data_len: 246)
multiply_chunks(hc_data_len: 246, hc_data_len: 248)
multiply_chunks(hc_data_len: 246, hc_data_len: 249)
multiply_chunks(hc_data_len: 246, hc_data_len: 249)
multiply_chunks(hc_data_len: 248, hc_data_len: 246)
multiply_chunks(hc_data_len: 248, hc_data_len: 248)
multiply_chunks(hc_data_len: 248, hc_data_len: 249)
multiply_chunks(hc_data_len: 248, hc_data_len: 249)
aggregate_results(°, hc_data_len: 12)
multiply_chunks(hc_data_len: 249, hc_data_len: 246)
multiply_chunks(hc_data_len: 249, hc_data_len: 248)
multiply_chunks(hc_data_len: 249, hc_data_len: 249)
multiply_chunks(hc_data_len: 249, hc_data_len: 249)
multiply_chunks(hc_data_len: 249, hc_data_len: 246)
multiply_chunks(hc_data_len: 249, hc_data_len: 248)
multiply_chunks(hc_data_len: 249, hc_data_len: 249)
multiply_chunks(hc_data_len: 249, hc_data_len: 249)

Fig. 10: Matrix Multiplication Workflow

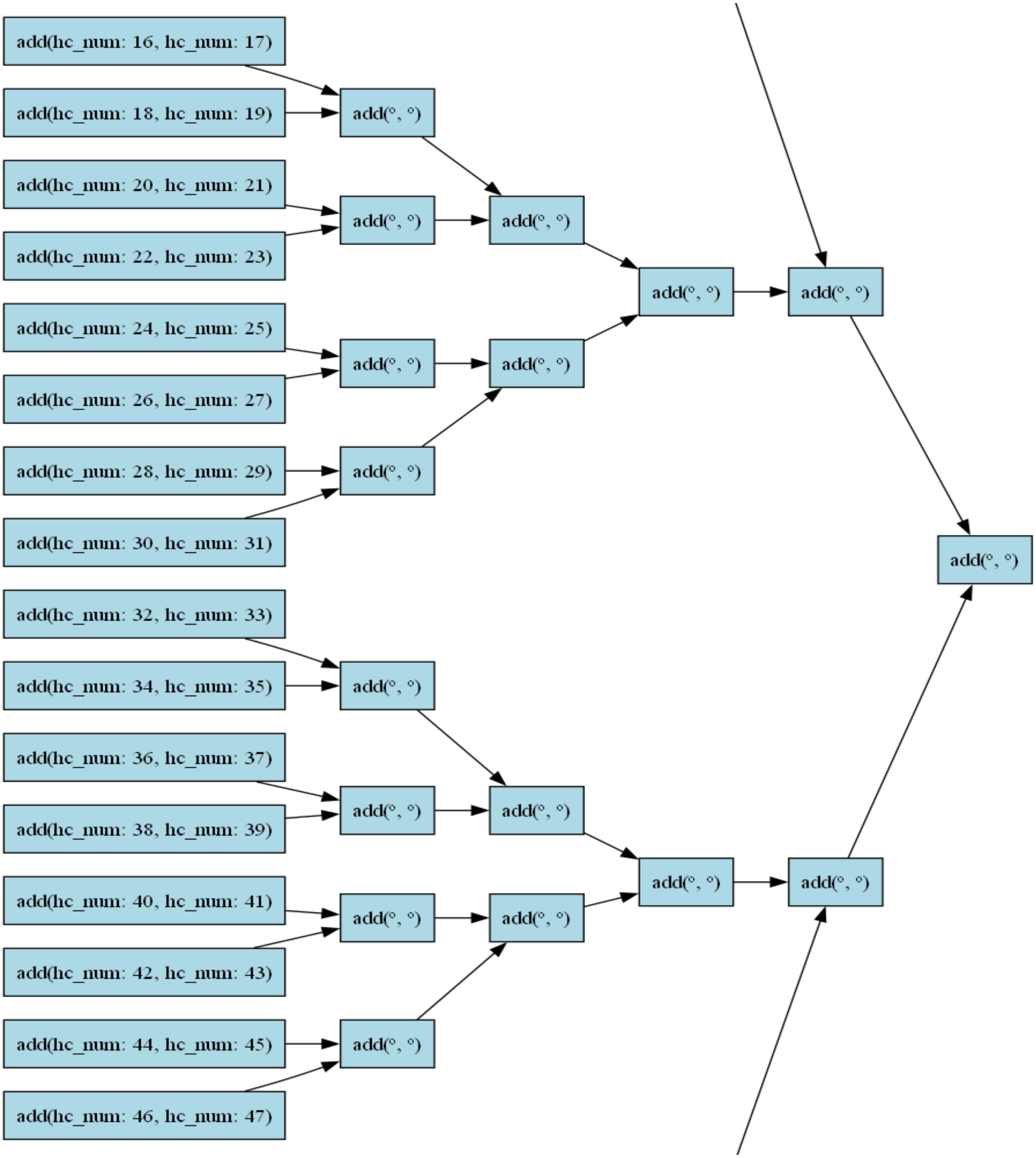
add(hc_num: 16, hc_num: 17)
add(hc_num: 18, hc_num: 19)
add(hc_num: 20, hc_num: 21)
add(hc_num: 22, hc_num: 23)
add(hc_num: 24, hc_num: 25)
add(hc_num: 26, hc_num: 27)
add(hc_num: 28, hc_num: 29)
add(hc_num: 30, hc_num: 31)
add(hc_num: 32, hc_num: 33)
add(hc_num: 34, hc_num: 35)
add(hc_num: 36, hc_num: 37)
add(hc_num: 38, hc_num: 39)
add(hc_num: 40, hc_num: 41)
add(hc_num: 42, hc_num: 43)
add(hc_num: 44, hc_num: 45)
add(hc_num: 46, hc_num: 47)
add(°, °)

Fig. 11: Tree Reduction Workflow (Partial)

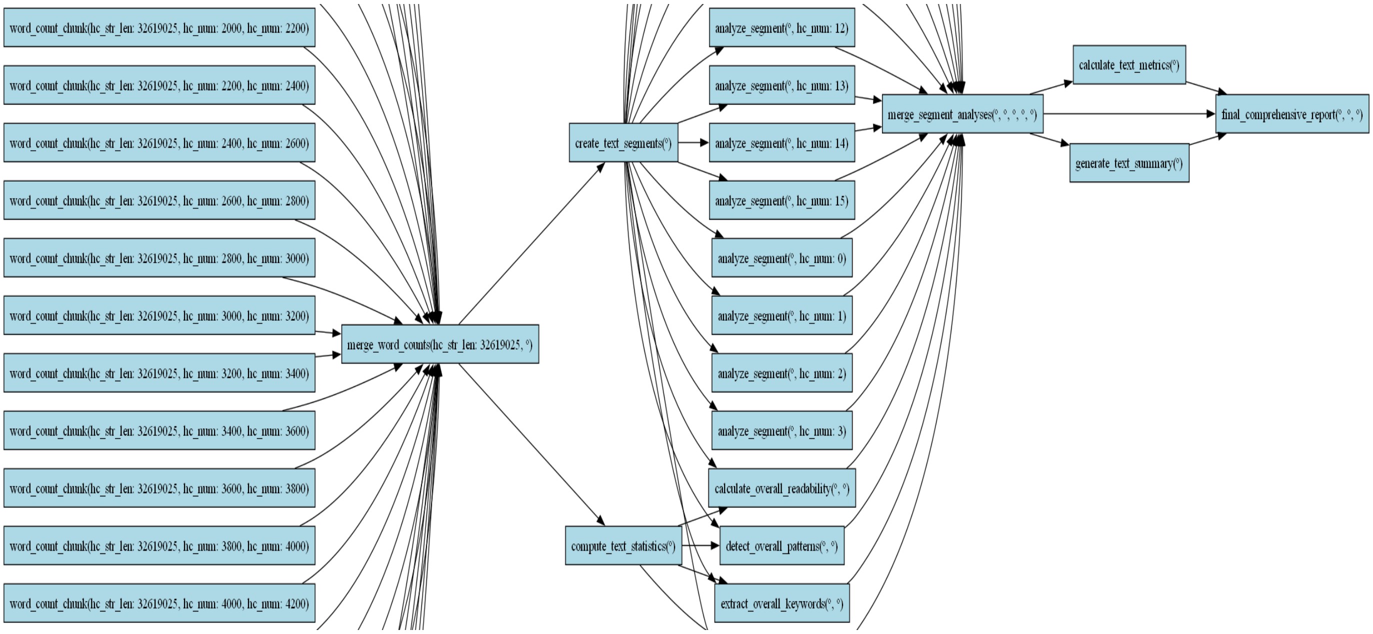
word_count_chunk(hc_str_len: 32619025, hc_num: 2000, hc_num: 2200)
word_count_chunk(hc_str_len: 32619025, hc_num: 2200, hc_num: 2400)
word_count_chunk(hc_str_len: 32619025, hc_num: 2400, hc_num: 2600)
word_count_chunk(hc_str_len: 32619025, hc_num: 2600, hc_num: 2800)
word_count_chunk(hc_str_len: 32619025, hc_num: 2800, hc_num: 3000)
word_count_chunk(hc_str_len: 32619025, hc_num: 3000, hc_num: 3200)
word_count_chunk(hc_str_len: 32619025, hc_num: 3200, hc_num: 3400)
word_count_chunk(hc_str_len: 32619025, hc_num: 3400, hc_num: 3600)
word_count_chunk(hc_str_len: 32619025, hc_num: 3600, hc_num: 3800)
word_count_chunk(hc_str_len: 32619025, hc_num: 3800, hc_num: 4000)
word_count_chunk(hc_str_len: 32619025, hc_num: 4000, hc_num: 4200)
merge_word_counts(hc_str_len: 32619025, °)
create_text_segments(°)
compute_text_statistics(°)
analyze_segment(°, hc_num: 12)
analyze_segment(°, hc_num: 13)
analyze_segment(°, hc_num: 14)
analyze_segment(°, hc_num: 15)
analyze_segment(°, hc_num: 0)
analyze_segment(°, hc_num: 1)
analyze_segment(°, hc_num: 2)
analyze_segment(°, hc_num: 3)
calculate_overall_readability(°, °)
detect_overall_patterns(°, °)
extract_overall_keywords(°, °)
merge_segment_analyses(°, °, °, °, °)
calculate_text_metrics(°)
generate_text_summary(°)
final_comprehensive_report(°, °, °)

Fig. 12: Text Analysis Workflow (Partial)

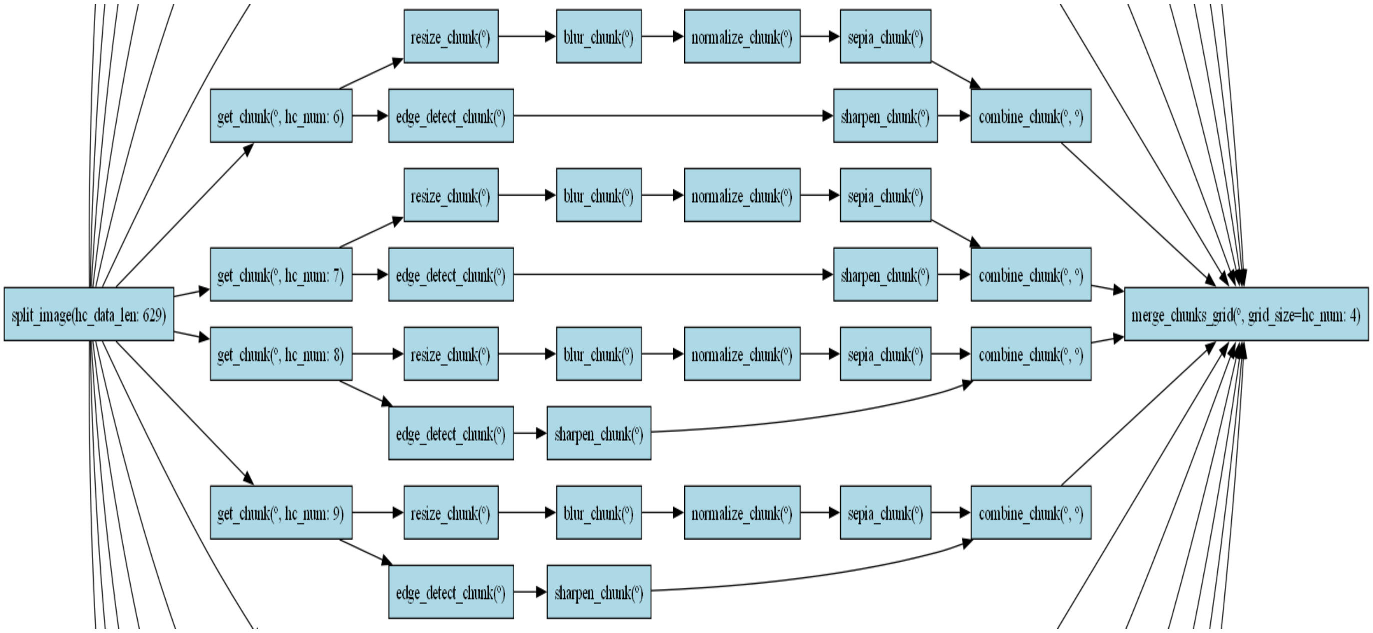

Fig. 13: Image Transformation Workflow (Partial)