FlowFusion: Optimizing Cloud Workflows Through Fusion and Parallelization

Nithin Tatikonda

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-80
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-80.html

May 16, 2025

Copyright © 2025, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

FlowFusion: Optimizing Cloud Workflows Through Fusion and Parallelization

by Nithin Tatikonda

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

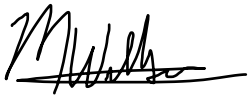
Committee:

Professor Alvin Cheung
Research Advisor

May 15, 2025

(Date)

* * * * * * *



Professor Max Willsey
Second Reader

May 12, 2025

(Date)

FlowFusion: Optimizing Cloud Workflows Through Fusion and Parallelization

Nithin Tatikonda
nithintatikonda@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Abstract

Currently, workflow services such as Google Cloud Workflows, AWS Step Functions, Azure Durable Orchestrations, and Airflow workflows execute literally as dictated by the user. This is not ideal as we want users to program for readability and programmability without worrying about impacts on performance. We introduce FlowFusion, a tool for programmatically combining or rearranging the separate tasks (task fusion) of a workflow into an optimized workflow with fewer workflow tasks, fewer database operations, and/or increased parallelism. FlowFusion works through a three-step process: profiling, task fusion, and task parallelization. Profiling involves executing the original workflow and determining the durations of tasks and read/write operations. Task fusion involves determining which tasks to combine into a single task to reduce the cost of scheduling new tasks and transferring data from task to task. Task parallelization involves determining if and how a data parallel task should be parallelized in order to minimize execution time. In implementing these three phases of our optimization tool, the "quirks" of cloud functions, cloud workflows, and database operations are considered. Our tool considers task fusion versus parallelism. Fusion inherently reduces parallelism, which could increase execution time. On the other hand, for some workflows, the task invocation overhead and task spin-up time could make fusion the optimal choice. Our tool also considers failure rate and retries for tasks. Some workflow tasks will have multiple retries enabled, meaning tasks will sometimes need to be executed until success is achieved or the number of retries is exceeded, and this behavior is also taken into consideration by the optimizer.

Overall, our evaluation shows that FlowFusion achieves significantly lower execution times for most workflows, achieving up to a 4× improvement.

1 Introduction

Cloud workflows are used for a wide variety of applications such as data engineering, machine learning, media processing, monitoring and alerting, e-commerce, and DevOps [3]. These workflows are often run numerous times as they can be triggered by user actions, sensors (e.g., environmental indicators [1]), or schedules (e.g., daily scheduled reports [32]) Therefore, optimizing workflows is extremely important as a slow workflow execution time can negatively impact user experience. For example, Bloomberg, which uses Airflow workflows [2], has clients who rely on up-to-date data to enable them to "assess the health of mortgage-backed securities," which requires efficient ETL pipelines [39]. Quicker workflow execution time can allow for timely insights and quicker iteration. In workflows where there are many tasks, small slowdowns can accumulate, leading to larger slowdowns. Therefore, optimizing workflow execution time is important to allow systems to handle more work and deliver faster results.

Workflows are a set of tasks or processes that are configured to execute in a certain order to achieve a specific goal. The success (completion without error) of the entire workflow is determined by the success of each of the individual tasks. Workflows are handled by the workflow engine or orchestrator. These engines or orchestrators are responsible for taking in a workflow specification in order to instantiate the workflow. They are then responsible for managing the execution of the workflows. The orchestrator is responsible for queuing up tasks that are ready to be executed. This involves handling task dependencies (queuing the next tasks after their prerequisite tasks or conditions are fulfilled) and handling the requeuing of tasks when failures occur. The queued-up tasks are started either by having a worker pick up the task to execute it or by making an API call to a service like AWS Lambda [7] to execute the task logic. Google Cloud Workflows [12], Azure Durable Orchestrations [8], and Airflow Workflows [2] are all workflow orchestration services that people can use to author their own workflows.

Workflows are created in a variety of different ways depending on the platform being used. Google Cloud Workflows, for example, are specified using a JSON or YAML file that specifies the order in which tasks are executed. Individual tasks can perform a variety of actions, such as making API calls to Google's services or other third-party services. Often, Google Cloud workflows chain together Google Cloud Functions [11]. On the other hand, Airflow requires the user to define their workflow in Python. This involves creating a DAG (Directed Acyclic Graph) object which represents the overall workflow, creating Operator objects which represent the individual tasks, and specifying task dependencies which define the prerequisites for certain tasks to start. Other workflow services have their own way of defining workflows.

The main problem with workflows and workflow services, as they currently exist, is that they have significant overhead when it comes to scheduling tasks, starting tasks, and transferring data between tasks. In addition, they do not always make full use of available resources. One reason these issues occur is that workflow services tend to execute exactly as dictated by the user without the use of more aggressive optimization techniques. Some prior work attempts to make workflows more efficient by making communication between tasks faster or by trying to speed up the allocation of resources and the startup of each task [34, 47]. This has shown to be effective but is insufficient as it does not attempt to address the idea of possible savings within a task. Other prior work related to function fusion tends to look more at the system aspect of combining or inlining cloud functions (fusion) to reduce the startup delay for a task [33, 44, 45]. However, they fail to consider the optimization of

an entire workflow end-to-end. Furthermore, optimizer-driven parallelization is novel for workflows. Most workflow services require users to manually create multiple tasks that can occur in parallel rather than specifying one task that can be parallelized.

The challenge in optimizing cloud workflows is taking into account the complex dependencies between tasks. This involves tasks that can only be executed after another task finishes executing and tasks that depend on data or some output from another task. Different scheduling/startup costs, input reading costs, task durations, and workflow structures can drastically change the best way to optimize a workflow, whether it be through task fusion, determining what tasks to combine or rearrange, or parallelization, determining how much to parallelize a task if at all.

In this paper, we introduce FlowFusion. FlowFusion addresses the problems facing workflow execution with two core ideas: task fusion and parallelization. Task fusion, when possible, allows for the elimination of any costs incurred by the scheduler between the execution of two consecutive tasks as well as costs involved in transferring data between tasks. Parallelization, when possible, allows for better utilization of available resources. Our tool ensures that an optimal number of branches is created in order to allow for speedup while not exceeding the number of workers, which can cause additional scheduling/startup costs. Our tool also accounts for failure rates and retries to more accurately model the execution time of the optimized workflow, allowing for more accurate optimization. In sum, we make the following contributions:

- We propose the idea of workflow task fusion, which allows tasks of a workflow to be combined, reducing task scheduling/startup costs, and eliminating data transfer costs.
- We propose an integer programming formulation that, when solved, determines which tasks should be fused and which tasks should remain unfused, shortening the critical path of a workflow.
- We propose a parallelization scheme that determines when and how data parallel tasks should be parallelized.
- We propose FlowFusion, which incorporates the previous three ideas to transform an Airflow workflow into an optimized workflow with lower execution time.
- We evaluate the time taken for optimization as well as the performance of the optimized and unoptimized workflows, demonstrating a significant reduction in execution time.

2 Related Work

2.1 Workflow Execution

Netherite [34] is an architecture that aims to optimize workflow execution. In workflows, the state of a task must be written to persistent storage before any successor tasks can proceed to execute. Netherite attempts to eliminate the costs associated with this by allowing the successor task to start before the persistence of a previous task. Essentially, the successor task can start while the predecessor task is still finishing up. Our solution is similar in that task fusion allows for the content of a successor task to execute immediately after the content of the predecessor task as the tasks have been combined. However, our solution differs in that task fusion allows the two tasks to be combined into one, meaning the successor task implicitly has access to the output of the predecessor task without having to read the output from persistent storage. This additional feature of task fusion allows for a potential reduction in execution time when compared to Netherite as extra database reads are eliminated.

One major way existing work has tried to optimize workflows and cloud functions is by addressing the problem of "cold start" [38, 41, 42, 46, 47]. The problem of "cold start" refers to the excessive time that is taken to allocate resources and actually start the execution of a function. Addressing this problem is similar to our work in that we work to eliminate the unnecessary costs incurred when starting up a task. However, our solution addresses this problem by simply taking over the environment of a preceding task when possible, resulting in a nearly nonexistent starting process. Our approach, when compared to works looking into "cold start," is advantageous as it does not require the extra cost incurred by having to restore snapshots. In addition, our solution allows for further optimization such as eliminating the need to read inputs from a previous task, which allows for further speedup.

2.2 Fusion

Our idea of task fusion is similar to the idea of task fusion in multi-user clusters [37]. In this case, separate programs, sometimes from separate users, are combined into a single task to save input reading costs when tasks have the same input data. However, this only works if the tasks are independent of one another. Our solution, which focuses on workflows, is intended to work on tasks that may depend on the completion and/or the outputs of a previous task.

Our idea of task fusion is also similar to the idea of function fusion demonstrated in the FUSIONIZE framework [45]. FUSIONIZE focuses on the Function-as-a-Service (FaaS) execution model, where developers deploy small, reusable chunks of code to be executed on a cloud service. FUSIONIZE particularly focuses on the case where these functions call each other either synchronously or asynchronously. FUSIONIZE focuses more on the system aspect of actually combining these functions. However, the optimization strategy consists of inlining functions when the call is synchronous and not inlining when the function call is asynchronous, which may not always be optimal as asynchronous function calls can still cause a greater execution time than executing the inlined function. Our approach, in contrast, uses a clear optimization strategy to determine when fusion would lower execution time. Our approach explicitly profiles the costs of reading and writing data from other tasks as well as the scheduling/startup costs of a task to determine whether tasks should be combined, regardless of whether the tasks execute synchronously or asynchronously.

In general, previous work has looked at function fusion for the FaaS model [33, 44], but these works do not consider the end-to-end performance of an entire workflow. They do not take into consideration the complex task dependencies, and they do not closely examine the exact costs of communicating data between tasks. Our implementation of fusion looks at the workflow application as a whole, with all of its complex dependencies, and reorganizes it based on profiled costs for scheduling/startup and input reading.

2.3 Parallelism

Although optimizer-driven task parallelization is novel in the context of cloud workflows, parallelization in the context of distributed computing is not. Dask [9], for example, optimizes parallel applications through work stealing, where busy workers get their tasks stolen from them by idle workers. However, stealing tasks has additional overhead, which we would like to avoid. Similar to Dask, other distributed computing frameworks, like Hadoop and Spark, encourage the creation of many smaller tasks [4, 5] as it is beneficial for load balancing. As discussed before, this creates a higher overhead as many tasks need to incur a scheduling/startup cost. Furthermore, spawning too many tasks could cause other tasks in the workflow to have to wait for a worker, which is not ideal. Therefore, it is beneficial in our case to only parallelize up to the number of available workers.

Frameworks like OpenMP [35] provide parallelism at the computation level. OpenMP can either determine how to assign work to threads statically equally distributing work to each thread, or dynamically schedule work, deciding how to assign work to threads at runtime. The problem with both of these approaches, when it comes to workflows, is that scheduling/startup overhead is much higher when it comes to workflows. As such, we need to determine whether or not parallelization is a worthwhile optimization before deciding to do it, which is why we use statistics on task durations when deciding to parallelize.

3 Overview

At a high level, FlowFusion is a tool that takes in an Airflow workflow and outputs and a workflow optimized through task fusion and task parallelization. Although this tool works only on Airflow workflows, the core idea can be applied to any workflow orchestration service. Moreover, the tool focuses on static workflows, meaning that the original workflow's structure does not change between runs. This is not a serious limitation, as cloud workflows often have a fixed and predictable structure that remains constant across executions. In fact, some workflow services, such as Flexera [10], only support static workflows.

As shown in Figure 1, FlowFusion works by performing the following three steps: profiling, task fusion, and task parallelization. Each of these three steps are explained in more detail in the following sections. Profiling runs the workflows several times and then gathers information about task durations, read costs, write costs, task scheduling/startup costs, and read/write locations (where certain tasks are reading and writing to). Task fusion involves strategically combining tasks to eliminate the costs associated with queuing up and then starting up a task (by making it so that these costs are only incurred once for the group of fused tasks). How a task is fused is determined by an integer programming formulation as described in the section on task fusion. Finally, task parallelization involves taking tasks that are data parallel and determining if and how they should be parallelized considering the extra scheduling costs required for parallelization. FlowFusion computes the information, such as which tasks to fuse and how much to parallelize a certain task, during the optimization phase itself. This means that FlowFusion does not do any dynamic task creation, task fusion, or task parallelization at the time the workflow is running, which saves time as dynamically creating and deploying new tasks can further increase execution time.

Here, we have included an example workflow that we will refer to throughout this paper. The code snippet can be seen in Listing 1. The corresponding unoptimized workflow can be seen in Figure 2. In the figure, T represents the time it takes for the entire task to run. R represents the time it takes for the task to read its input from the previous task (also incorporated as a part of T). C represents the scheduling/startup time.

```
def data_ingestion():
    data = ingest_data()
    write('xcom', 'data_ingestion', data)

def data_summary():
    data = read('xcom', 'data_ingestion', data)
    summary = summarize_data(data)
    write('xcom', 'data_summary', summary)

def sa_read():
    return read('xcom', 'data_ingestion')

def sa_shard(sharding_num, df):
    return np.array_split(df, sharding_num)

def sa_compute(df):
    return compute(df)

def sa_merge(df_list):
    return pd.concat(df_list)

def sa_write(df):
    write('xcom', 'sentiment_analysis', df)

def email_summary():
    summary = read('xcom', 'data_summary')
    email_summary_to_subscribers(summary)

default_args = {
    'owner': 'airflow',
    'start_date': datetime(2023, 7, 1),
}

with DAG(dag_id='tweet_analysis_dag',
         default_args=default_args) as dag:

    # Declare Tasks
    t1 = PythonOperator(task_id='data_ingestion',
     python_callable=data_ingestion)
    t2 = PythonOperator(task_id='data_summary',
     python_callable=data_summary)
    t3 = ParallelFusedPythonOperator(
        task_id='sentiment_analysis',
        data_collection_function=sa_read,
        sharding_function=sa_shard,
        compute_function=sa_compute,
        merge_function=sa_merge,
        write_function=sa_write
    )
    t4 = PythonOperator(task_id='email_summary',
     python_callable=email_summary)

    # Task Dependencies
    t1 >> [t2, t3]
    t2 >> t4
    t3 >> t4
```

Listing 1: Tweet Analysis Workflow

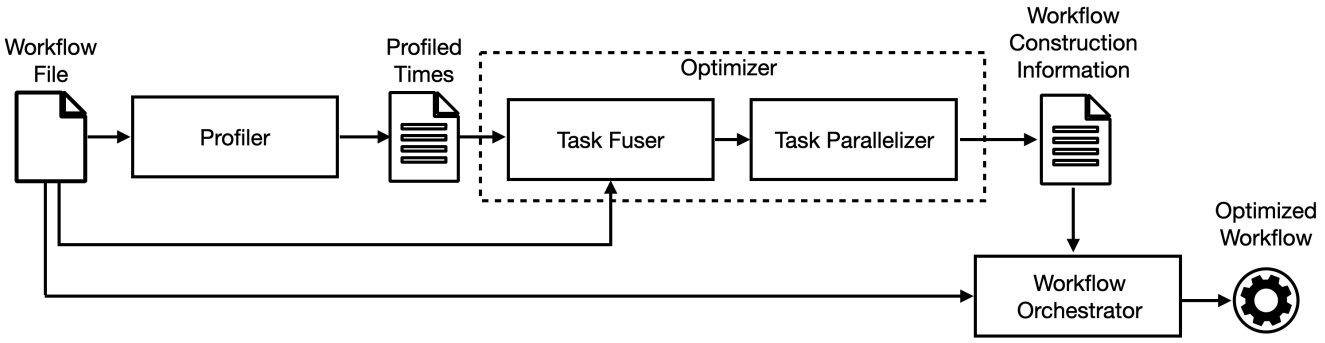


Figure 1: Execution Overview

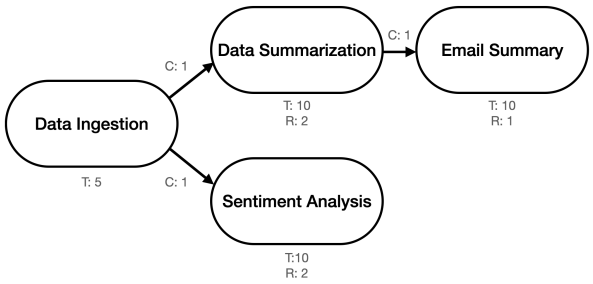


Figure 2: Tweet Analysis Original Workflow

3.1 Preparing the Input

The initial workflow file that is fed into the profiler and the optimizer must meet some requirements. The Airflow workflow must be defined with traditional syntax, not using the TaskFlow API. Next, reads and writes to persistent store should be done using the optimization library's new read and write functions. These new read and write functions allow the profiler to recognize and profile these persistent store accesses. They will also allow the optimizer to recognize read costs that can be eliminated through task fusion. Finally, for data parallel tasks, the user should use the ParallelFusedOperator. Instantiating this operator with functions to read the data, shard the data, perform computation, merge the data, and write the output will allow the task to be parallelized by the task parallelizer. The sharding function will allow the task to be parallelized to any number of branches determined by the optimizer. The merge function will gather all the outputs of the parallelized computation to combine the results. Finally, the user can also optionally provide a failure rate for tasks that have retries enabled.

4 Profiler

The purpose of the profiler is to gather information about the task durations for tasks in the original workflow as well as information about the reads and writes that facilitate data transfer between tasks. The profiler takes as input the workflow that needs to be profiled. The profiler then executes the workflow N times where N is a value specified by the user. Airflow already logs the start times and end times for each task, which allows us to determine the task durations and the task scheduling/startup costs. The read and write functions discussed in the previous section also record the time required for their completion and the identity of the task they are currently executing within, which provides access to read costs, write costs, and read/write locations. The profiler then outputs the average task durations, average read costs, average write costs, task scheduling/startup costs, and information about which tasks are reading or writing to which locations. The information collected by the profiler is passed on to the optimizer, which will use the information to produce an optimized workflow. As discussed previously, workflows are often run numerous times. Therefore, profiling the original workflow ahead of time in order to perform optimization becomes a worthwhile effort.

5 Optimizer

Before we can fully dive into task fusion and task parallelization, we need to consider failure rates and retries. Failures and retries should also be considered when performing optimization as this can affect the overall time it takes for tasks to complete. Task failures are handled by the scheduler/orchestrator for most workflow services. When a task fails, the scheduler is informed of the failure, which will then reschedule the task for the retry. This adds a scheduling cost that can be eliminated. In order to remove this cost, retries are moved from above the task level to below the task level. Essentially, a task that fails will not have to communicate back to the orchestrator to be scheduled for a retry. Instead, retries will occur on the worker that has been assigned to execute the task. This will also be the case for tasks that have been fused together. Assuming task 1 and task 2 are fused, if task 1 has 3 retries and task 2 has 4 retries, then the combined task will fail if the content of task 1 fails 4 times or the content of task 2 fails 5 times. To account for this handling of task retries, the user can provide failure rates for tasks in the original workflow (the failure rate will be treated as 0 if not provided). The user can obtain these failure rates by finding SLAs for the services used in each task. The tool uses these failure rates to compute a more accurate estimate of task durations. Assuming t_i is the average successful task duration of task i and

f_i is the failure rate of task i, the task duration used for the task fusion and parallelization optimizations will be:

$$\frac{t_i}{1 - f_i} \tag{1}$$

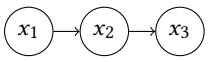
This is a simple expected value calculation of the time for the task to complete given that retries are allowed. Going forward, the time for a task to execute will be given by this modified task duration.

5.1 Task Fusion

As discussed above, task fusion provides two major benefits. The first benefit is that it eliminates any task scheduling/startup costs. Any time between the completion of the predecessor task and the start of the execution of the successor task would be completely eliminated by task fusion. The second benefit is the ability to eliminate unnecessary reads in the successor task. With task fusion, the successor task will be able to obtain the inputs without having to read from persistent store (assuming that the predecessor task has read or has written the corresponding input values).

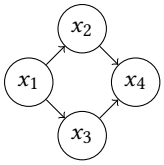
5.1.1 Execution Time Modeling. Workflows can be represented as directed acyclic graphs. The vertices represent tasks while the edges represent task dependencies. An edge from x_1 to x_2 indicates that task x_1 must be completed before task x_2 can begin its execution. In this directed graph, both tasks and edges have associated costs. The cost of a task is its task duration. The cost of an edge is the task scheduling/startup cost. For the workflow optimization that is being performed, the execution time of a workflow can be modeled as the critical path through the graph. This will end up being the time between the first task starting its execution and the last task ending its execution. Below are two examples of how execution time is modeled for workflows. t_i represents the execution time for task i, c represents the scheduling/startup cost, and C represents the total execution time.

Case 1: Linear



$$C_{original1} = (c + t_1) + (c + t_2) + (c + t_3)$$

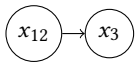
Case 2: Nonlinear (Assume $t_2 > t_3$)



$$C_{original2} = (c + t_1) + (c + max(t_2, t_3)) + (c + t_4)$$
$$C_{original2} = (c + t_1) + (c + t_2) + (c + t_4)$$

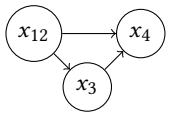
Now, that we have equations representing the execution time of these two workflows, we can see the benefits of task fusion in both cases. In the following examples, we say that r_{ij} is the read cost that is no longer needed in task i when combined with task j. If we assume that only tasks 1 and 2 are fusable in both cases, we get the following:

Case 1: Linear



$$C_{fused1} = (c + t_1 + t_2 - r_{21}) + (c + t_3)$$
$$r_{21} \leq t_2, c > 0$$
$$C_{original1} - C_{fused1} = c + r_{21}$$
$$C_{original1} > C_{fused1}$$

Case 2: Nonlinear (Assume $t_2 > t_3$)



$$C_{fused2} = (c + t_{12}) + (c + t_3) + (c + t_4)$$
$$t_{12} = t_1 + t_2 - r_{21}$$
$$C_{original2} - C_{fused2} = r_{21} - t_3$$
$$C_{original2} > C_{fused2} \quad iff\ r_{21} > t_3$$

In the linear case, the savings are obvious. Even if r_{21} is 0, there are still savings because of the reduced scheduling cost. This particular nonlinear case is less obvious. In this case, r_{21} needs to exceed a certain threshold for fusion to be beneficial. In the nonlinear case it can also be seen that parallelism is eliminated when choosing to fuse x_1 and x_2. In the original workflow, x_2 and x_3 would have executed at the same time. However, in the fused workflow, x_{12} executes, which consists of the content of x_1 and x_2 executing consecutively. Then x_3 executes. Finally, x_4 executes.

Furthermore, in the nonlinear case, if all tasks are fusable with one another, there is the case where all tasks can be fused into one, also eliminating all parallelism. Fusion helps with shortening critical paths and in reducing the overall execution time. Therefore, it is not readily apparent when and how fusion can and should be performed. Figure 3 shows how fusion is applied to the example workflow in Figure 2. Calculating the critical path of the original and fused workflows, as we have just done, gives us a cost of 27 and 24 respectively, showing the benefits of fusing.

As you can see, using the critical path as a model for execution time implies perfect parallelism. Although this model may seem unusual, it is important to realize that fusion inherently reduces parallelism as discussed previously. Therefore, by using this model of parallelism, fusion can only occur when the execution time for the fused graph is less than the original graph assuming perfect

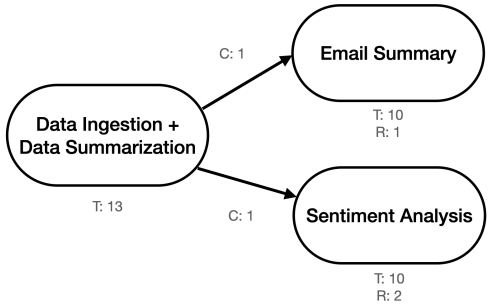


Figure 3: Tweet Analysis Fused Workflow

parallelism, meaning that the fused graph will likely always have a lower execution time even when perfect parallelism is not the case.

5.1.2 Integer Programming Formulation. As discussed previously, it is not always clear when task fusion should be applied. Therefore, we rely on integer programming to filter through the possibilities. In the following, we have specified an integer programming formulation based on the execution time model discussed above. This is inspired by the critical path integer programming formulation [36]. However, in our case, we are constructing the graph on the fly and trying to find the minimum critical path from the starting task to the ending task. We construct a graph by taking into account the predecessor and successor relationships of the original workflow. We can construct operator edges, which means that the two tasks are unfused but there is still a task dependency. We can also create fused edges, meaning that tasks are fused. When a fused edge is present, it indicates that there will be no scheduling/task startup cost. It also indicates that the cost of reading data from another task could be eliminated. However, if a fused edge is present, then it is the only possible incoming edge or the only possible outgoing edge, meaning that parallelism is prevented. These rules are implemented in the following integer programming formulation.

Constants.

- s : (constant) source task
- v : (constant) end task
- M : (constant) large constant for utility.
- c : (constant) scheduling cost
- p_{ij} : (constant) binary constant that is 1 if task i is a predecessor of task j in the original workflow.
- t_i : (constant) amount of time for task i to execute.
- r_{ij} : (constant) is the read cost in task j that is written or read in task i. This is the eliminatable read cost if i and j are fused.

Variables.

- d_i : nonnegative continuous variable that is the longest path length from i to v.
- e_{ij} : binary variable indicating an operator edge between i and j. j cannot be a predecessor of i.
- f_{ij} : binary variable indicating a fused edge between i and j. j cannot be a predecessor of i. Tasks i and j must also be fusable.
- x_{ij} : binary variable indicating if i is a predecessor of j in the same fused component
- y_{ij} : binary variable indicating if i is a predecessor of j.
- a_{ijk} : binary variable that is one if and only if $(f_{ik} + e_{ik})$ is one and y_{kj} is one.
- b_{ijk} : binary variable that is one if and only if f_{ik} is one and x_{kj} is one.
- g_{ij} : nonnegative continuous variable that is the longest path length from task i to task v with task j as the next node connected with an operator edge.
- h_{ij} : nonnegative continuous variable that is the longest path length from task i to task v with task j as the next node connected with a fusion edge.

Objective and Constraints. The objective is to minimize execution time by minimizing the critical path from the source to the end.

$$\text{minimize } d_s$$

1. Enforce that d_i is the longest path for the case of operator edges.

$$d_i \geq g_{ij} \; \forall i \neq j$$

2. Set g_{ij} to enforce the previous constraint.

$$g_{ij} \leq M * e_{ij} \; \forall i \neq j$$
$$g_{ij} - (c + t_j + d_j) \leq M * (1 - e_{ij}) \; \forall i \neq j$$
$$(c + t_j + d_j) - g_{ij} \leq M * (1 - e_{ij}) \; \forall i \neq j$$

3. Enforce that d_i is the longest path for the case of fusion edges.

$$d_i \geq h_{ij} \; \forall i \neq j$$

4. Set h_{ij} to enforce the previous constraint.

$$h_{ij} \leq M * f_{ij} \; \forall i \neq j$$
$$h_{ij} - (d_j + t_j - \sum_k r_{kj} * x_{kj}) \leq M * (1 - f_{ij}) \; \forall i \neq j$$
$$(d_j + t_j - \sum_k r_{kj} * x_{kj}) - h_{ij} \leq M * (1 - f_{ij}) \; \forall i \neq j$$

5. If there is an outgoing fused edge, then it is the only outgoing edge. If there is an incoming fused edge, then it is the only incoming edge.

$$\sum_j e_{ij} \leq M * (1 - f_{ik}) \; \forall i \neq k$$
$$\sum_i e_{ij} \leq M * (1 - f_{kj}) \; \forall j \neq k$$
$$\sum_j f_{ij} \leq 1 \; \forall i \neq j$$
$$\sum_i f_{ij} \leq 1 \; \forall i \neq j$$

6. Constraint that sets x_{ij}, which checks for the predecessor relationship within a fused component.

$$x_{ij} \geq f_{ij} \; \forall i \neq j$$
$$x_{ij} \geq b_{ikj} \; \forall i \neq j \neq k$$
$$x_{ij} \leq f_{ij} + \sum_k b_{ikj} \; \forall i \neq j$$
$$x_{ij} + x_{ji} \leq 1 \; \forall i < j$$

7. Constraint setting b_{ikj} to enforce previous constraint

$$b_{ikj} \leq f_{ik} \; \forall i \neq j \neq k$$
$$b_{ikj} \leq x_{kj} \; \forall i \neq j \neq k$$
$$b_{ikj} \geq f_{ik} + x_{kj} - 1 \; \forall i \neq j \neq k$$

8. Enforce the predecessor constraints from the original workflow. There must be a path from i to j if i is a predecessor of j in the original workflow.

$$y_{ij} \geq f_{ij} + e_{ij} \; \forall i \neq j$$
$$y_{ij} \geq a_{ikj} \; \forall i \neq j \neq k$$
$$y_{ij} \leq f_{ij} + e_{ij} + \sum_k a_{ikj} \; \forall i \neq j$$
$$y_{ij} \geq p_{ij} \; \forall i \neq j$$
$$y_{ij} + y_{ji} \leq 1 \; \forall i < j$$

9. Constraint setting a_{ikj} to enforce previous constraint

$$a_{ikj} \leq f_{ik} + e_{ik} \; \forall i \neq j \neq k$$
$$a_{ikj} \leq y_{kj} \; \forall i \neq j \neq k$$
$$a_{ikj} \geq f_{ik} + e_{ik} + y_{kj} - 1 \; \forall i \neq j \neq k$$

10. Enforce that all tasks have a path to the end.

$$y_{iv} = 1 \; \forall i \neq v$$

11. Enforce that all tasks have a path from the start.

$$y_{si} = 1 \; \forall i \neq v$$

12. Need to set constraints that make certain pairs of tasks unfusable. This can be done by setting f_{ij} to 0 for unfusable tasks.

$$f_{ij} = 0 \; \forall notFusable(i, j)$$

13. Enforce binary constraints and ensure all continuous variables are nonnegative.

$$e_{ij}, f_{ij}, x_{ij}, y_{ij}, a_{ijk}, b_{ijk} \in 0, 1 \quad \forall i, j, k$$

$$d_i, g_{ij}, h_{ij} \geq 0 \quad \forall i, j$$

Integer programming can be very unpredictable in terms of the time it takes to get the provably optimal solution. Solvers such as Gurobi [13], the solver used in this project, also tend to find the optimal objective value quickly but can then proceed to take a long time to prove the optimality. Since any feasible solution to our integer programming formulation will also produce a valid workflow, for the purpose of this paper, a time limit of 5 minutes has been set for Gurobi to find the best possible solution. We can use the best possible solution that Gurobi has achieved in those 5 minutes to create the optimized workflow. This will be very helpful for workflows with a larger number of tasks, which can blow up the number of variables in the problem.

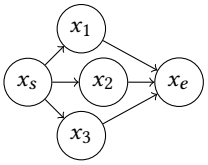
5.2 Task Parallelization

Parallelism is a clear direction to take when considering optimizing workflows as it allows workflows to make use of available resources such as idle workers, which in turn allows tasks to be completed more quickly. However, even when data parallel tasks are available, attempting to parallelize excessively can cause its own problems. For example, if there are 8 available workers for a parallelizable task, attempting to parallelize excessively, for example, by attempting to schedule 16 tasks, will cause slowdowns. This happens because the first 8 tasks have to incur a scheduling/startup cost when being assigned on to a worker. Then, once those tasks are complete, a second round of scheduling/startup costs will have to be incurred. Correctly branching out to 8 tasks, the number of available workers will avoid this unnecessary cost. Statically setting the branch-out factor to the the total number of workers is also not an option as other tasks in the workflow could be using the workers, which is why we use the number of currently available workers. Dynamic creation of tasks has additional costs when compared to statically determining the number of tasks, which is why we choose to determine the amount of parallelism during optimization time prior to running the workflow by estimating the number of available workers at any given time.

Increasing parallelism will work by simulating a workflow execution using the statistics gathered. We use a minimum priority queue ordered by task end time to keep track of tasks that are currently executing. We will call this priority queue the execution heap. At each round, we pop a task off the heap and add its successors on to a queue. We will call this queue the task queue. Then, we take tasks from the task queue and add them to the execution heap until the task queue is empty or the execution heap reaches a size of M, where M is the number of workers allowed for the workflow. When adding tasks to the execution heap, we take a special look at parallelizable tasks. We calculate the number of available workers by taking M and subtracting the length of the execution heap. We will call the number of workers available N. N tells us the maximum amount we can try to parallelize the task. Assuming N is 3, the following image shows how parallelization looks in a workflow.

Original:

Parallelized:



Here, x_s is the task that reads the necessary data and does any work that cannot be parallelized. x_s also calls the user-defined sharding function to split up the data among the subsequent tasks.

This sharding function should take in a number b and data d. It will then split d into b different parts (in this case b will be set to N). x_1, x_2, and x_3 represent the parallelized computation task on the data that has been split up from the first initial task. Finally, x_e is the final task that combines the results of the branched tasks and does any additional work that could not be parallelized, including writing the output to some persistent store.

This optimization is only done if the parallelized version is determined to have a shorter execution time based on the number of available workers. This is done by checking the following condition where N is the number of available workers, t_x is the execution time of task x, and c is the scheduling cost:

$$2c + \frac{t_x}{N} < t_x$$

This formula checks if parallelizing the task, which creates two extra tasks along the critical path and incurs extra scheduling/startup costs, will have a lower execution time when compared to the unparallelized task.

Figure 4 shows how the optimizer implements parallelism in the fused workflow from Figure 3 when the number of workers is 4. In this case, the "Sentiment Analysis" is the only parallelizable task.

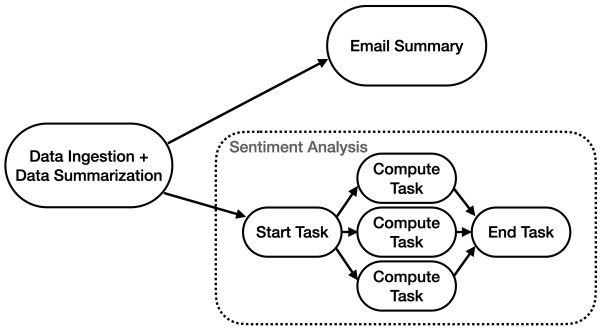


Figure 4: Tweet Analysis Fused + Parallelized Workflow

6 Evaluation

6.1 Code, Setup, and Benchmarks

To test the tool on a more standardized setup, we chose to run the optimization tool and the benchmark workflows on an AWS EC2 instance with 8 Airflow workers. These are the specifications of the EC2 instance used:

- Instance Type: t2.2xlarge
- vCPUs: 8
- Memory (GiB): 32
- Instance Storage: EBS-Only
- Network Performance: Moderate

Most of the benchmarks were sourced from Astronomer, the company that provides managed services related to Airflow. Astronomer's website provides a registry of numerous example workflows from the Airflow community. These workflows are verified and certified by Astronomer. In addition to Astronomer's registry, workflows were also sourced from Serverless Land, the site that brings together resources related to AWS Serverless. Serverless Land contains a registry of AWS Step Function workflows. We converted some of these workflows to Airflow for use as benchmarks. For these workflows, benchmarks were chosen that did not make excessive use of different AWS specific services. Finally, we also looked into Airflow workflows on GitHub. Any data required to run these benchmarks were provided by the GitHub repositories themselves. This included repositories that contained csv files within the repository, provided API calls to public data sources, and used public BigQuery datasets. For all of these benchmark sources, we tried to choose workflows that were not just showcasing features of workflow, but were actually trying to accomplish some task.

In addition, for all of the workflows, when possible, accesses to any cloud storage were turned into accesses AWS S3 buckets. Any calls to AI services were turned into calls to OpenAI services to avoid creating an excessive number of accounts. Moreover, when data parallel tasks appeared, the tasks were changed to use the new ParallelFusedOperator that would take in information like a user-defined sharding function. Finally, the uses of the TaskFlow API were changed to Airflow traditional syntax.

All associated code and benchmarks can be found at https://github.com/nithintatikonda1/AirflowOptimization.

6.2 Performance of Optimized Workflows

In order to gather the data for Figure 5, three variations of each benchmark were run: the original workflow, the workflow with fusion optimization, and the workflow with fusion and parallelization optimization. For nearly every case, the optimized (both fused and fused + parallel) workflows achieve an execution time less than the original workflow. For the case where this has not happened, such as in the case for the "great_expectations" benchmark, it is because the optimizer found that the original workflow was already in an optimal configuration. In these cases, the optimized workflows have nearly the exact same average execution time as the original. The very small difference in execution time can be attributed to the variance in execution time for the workflows as the workflows make calls to third-party services. A similar phenomenon occurs when comparing the workflow with only the fusion optimization applied and the workflow with both optimizations applied. The cases where the fused and the fused+parallel execution times are nearly identical occur when the optimizer decides not to parallelize or because the original workflow did not have any data parallel tasks to parallelize.

Overall, FlowFusion achieves an average speedup of 2.25× on all benchmarks when turning on both the fusion optimization and parallelization optimization. FlowFusion achieves an average speedup of 2.09× with just fusion optimization. These results are expected. As long as the execution time model holds true and as long as the task duration and read cost distributions do not vary too widely from the workflow executions that have been profiled, then there is no reason why our tool should not produce an optimized workflow with an approximately equal or lower execution time.

Figure 6 shows how scaling the amount of input data affects the execution time of the "stock_tweets" benchmark. The optimizer

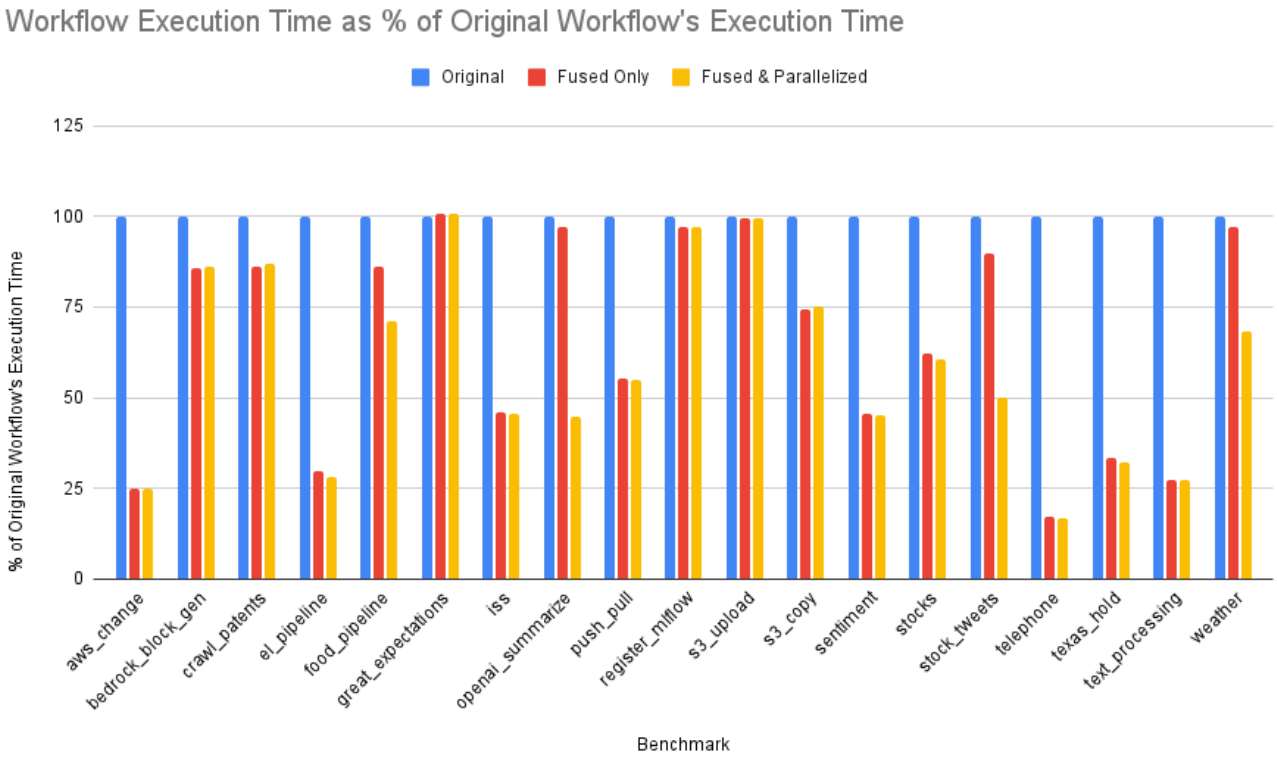


Figure 5: Execution time as a percentage of the original workflow's execution time. For each benchmark, we take the average over 10 executions for the original, the fusion optimization turned on, and both the fusion and parallelization optimization on.

Table 1: Benchmark Descriptions

Benchmark	Description
aws_change [22]	Calculates coin types to give when giving change.
bedrock_blog_gen [23]	Builds a blog by AI generating an image and an article.
crawl_patents [43]	Crawls patents related to phones and software.
el_pipeline [24]	Uploads data to S3 and performs validation checks.
food_pipeline [26]	Processes raw data about fruits and vegetables and then uploads it.
great_expectations [28]	Runs Great Expectations validation suite on a pandas dataframe.
iss [14]	Pulls the location of the ISS and record it.
openai_summarize [15]	Summarizes and searches financial documents with AI.
push_pull [16]	Sends and retrieves data from S3 buckets
register_mlflow [27]	Trains and registers a model to MLFlow
s3_upload [17]	Uploads multiple files to s3
s3_copy [18]	Uploads multiple files to s3 and copies files.
sentiment [19]	Retrieves jokes from API and performs sentiment analysis
stocks [29]	Determines whether to buy or sell a stock.
stock_tweets [31]	Analyzes tweets for stock prediction.
telephone [20]	Plays the telephone game with some text.
texas_hold [21]	Simulates a game of Texas Hold Em
text_processing [30]	Performs text processing on some text.
weather [25]	Trains a weather model and uses it for predictions

was rerun for each number of data points tested. This benchmark utilizes both the fusion and parallelization optimizations. At around 8000 data points, the execution time of both the original and the optimized workflows becomes very similar. This also happens to be the

point where the optimizer switches from not parallelizing the data parallel task to parallelizing it. This makes sense as parallelizing when the number of data points is low would cause extra overhead for little payoff. Parallelization would be more beneficial for a larger input due to larger savings in execution time. The graph also makes sense, as fusion is the most beneficial when the individual task durations are small, which means eliminating scheduling/startup costs has a greater impact. Moreover, as the amount of data grows, the greater the variance of the execution time, which could mask savings from fusion. The graph reflects this, as the optimized workflow performs at its best (compared to the original workflow) when the input data is either very small or very large. A similar phenomenon also occurs in Figure 7 for the "food_pipeline" benchmark. At around 400 input files, the optimizer switches from not parallelizing the data parallel tasks to parallelizing the data parallel tasks.

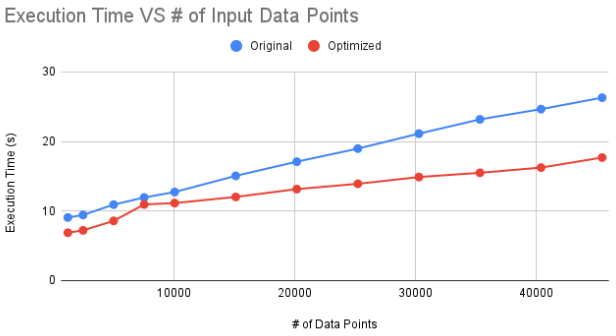


Figure 6: Varying the input data points for the stock_tweets benchmark. Each point is an average over 5 executions.

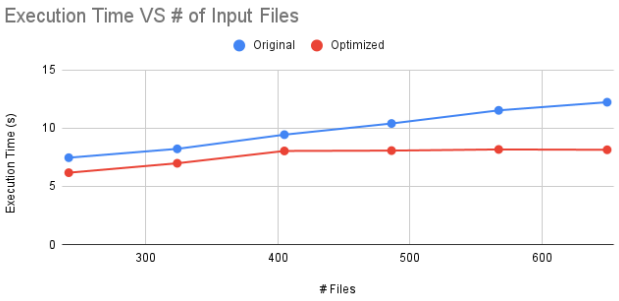


Figure 7: Varying the input data files for the food_pipeline benchmark. Each point is an average over 5 executions.

Figure 8 shows how the failure rate affects the execution time of the "push_pull" benchmark. The failure rate was manipulated by randomly failing a task according to the specified failure rate. The graph shows that the optimized workflow handles the increasing failure rate much better than the original workflow. While the execution time of the optimized workflow increases very gradually, the execution time of the original workflow increases at a more rapid rate, likely because of the increased costs associated with rescheduling the task after each retry.

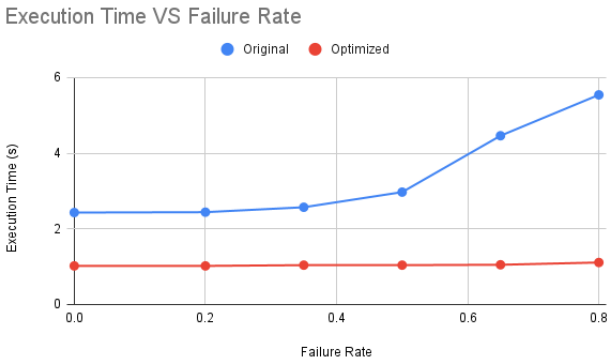


Figure 8: Varying the failure rate for one task in the push_pull benchmark. Each point is an average over 30 executions.

6.3 Optimization Time

Table 2: Optimization Time For Benchmarks

Benchmark	Number of Tasks	Optimization Time
aws_change	4	2.58
bedrock_blog_gen	3	1.54
crawl_patents	3	1.52
el_pipeline	4	2.47
food_pipeline	4	2.81
great_expectations	1	0.27
iss	3	2.10
openai_summarize	5	3.99
push_pull	2	0.81
register_mlflow	4	2.37
s3_upload	11	34.69
s3_copy	21	404.24
sentiment	4	2.54
stocks	6	6.01
stock_tweets	4	2.61
telephone	6	6.10
texas_hold	7	9.21
text_processing	4	2.53
weather	4	2.09

In order to the gather the data for optimization time, the optimization code for all benchmarks was run five times each. The average of those optimization times is shown in Table 2. As the table shows, optimization time for the benchmarks generally increases as the number of tasks in the original workflow. The times quickly ramp up from about a second for some of the shorter workflows to past our cut-off mark of 5 minutes for the largest workflows. The largest workflow, with 21 tasks, ends up with 844 continuous variables and 16293 binary variables to solve for. Taking a closer look at the solver's logs, the solver actually finds a solution that achieves the optimal objective value in around 3 minutes, showing that cutting off the solver and using the best solution found is a viable solution.

7 Conclusion

Our work shows that the execution time of workflows can be substantially reduced by an intelligent application of task fusion and parallelization, which effectively considers task durations, read-/write costs, and failure rates. Our optimization of cloud workflows will allow workflow authors to concentrate on the core applications of workflows without being concerned about the inefficiencies associated with existing orchestration services.

References

[1] 2024. *A Guide to Airflow Sensor Applications.* https://flussoltd.com/2024/11/07/a-guide-to-airflow-sensor-applications/

[2] 2025. *Apache Airflow.* https://airflow.apache.org/

[3] 2025. *Apache Airflow® Use Cases: A Comprehensive Guide with Real-World Examples.* https://www.astronomer.io/airflow/use-cases/?utm_term=airflow+use+cases&utm_campaign=airflow-guides-na&utm_source=adwords&utm_medium=ppc&hsa_acc=4274135664&hsa_cam=21865965775&hsa_grp=175142912878&hsa_ad=731398628570&hsa_src=g&hsa_tgt=kwd-1592919964288&hsa_kw=airflow+use+cases&hsa_mt=p&hsa_net=adwords&hsa_ver=3&gad_source=1&gad_campaignid=21865965775&_gl=1*qwgjod*_up*MQ..*_gs*MQ..&gclid=CjwKCAjwiezABhBZEiwAEbTPGHZl3WghnZOKCHRvhaYvRKt37KmwnlH-PtP3dHjjiWiFg_K2V_QLMhoCBsEQAvD_BwE&gbraid=0AAAADP7Y9iW7vdoteJyCBnrBWIx00WVe

[4] 2025. *Apache Hadoop.* https://hadoop.apache.org/

[5] 2025. *Apache Spark.* https://spark.apache.org/

[6] 2025. *AWS Lambda.* https://aws.amazon.com/lambda/

[7] 2025. *AWS Step Functions.* https://aws.amazon.com/step-functions/

[8] 2025. *Azure Durable Orchestrations.* https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-orchestrations?tabs=csharp-inproc

[9] 2025. *Dask Work Stealing.* https://distributed.dask.org/en/stable/work-stealing.html

[10] 2025. *Flexera Cloud Workflow Language.* https://docs.flexera.com/flexera/EN/Automation/CWL.htm?utm_source=chatgpt.com

[11] 2025. *Google Cloud Run Functions.* https://cloud.google.com/functions?hl=en

[12] 2025. *Google Workflows.* https://cloud.google.com/workflows?hl=en

[13] 2025. *Gurobi Optimization.* https://www.gurobi.com/

[14] [n. d.]. https://github.com/astronomer/get-started-with-airflow-2-tutorial/tree/1.3.1.

[15] [n. d.]. https://github.com/astronomer/airflow-llm-providers-demo/tree/1.0.2.

[16] [n. d.]. https://github.com/astronomer/pass-data-between-tasks-webinar/tree/1.0.0.

[17] [n. d.]. https://github.com/astronomer/airflow-example-dags/blob/v1.0.0/dags/s3_upload_dag.py.

[18] [n. d.]. https://github.com/astronomer/airflow-example-dags/blob/v1.0.0/dags/s3_upload_copy_dag.py.

[19] [n. d.]. https://github.com/astronomer/sdk-sentiment-analysis-demo/blob/1.0.0/dags/manatee_sentiment.py.

[20] [n. d.]. https://github.com/astronomer/webinar-task-groups/blob/1.0.1/dags/telephone_game.py.

[21] [n. d.]. https://github.com/astronomer/2-7-example-dags/blob/1.0.0/dags/setup_teardown/setup_teardown_cleanup_xcom.py.

[22] [n. d.]. AWS Step Functions Workflows Collection. https://github.com/aws-samples/step-functions-workflows-collection/tree/main.

[23] [n. d.]. Bedrock Blog Generator. https://github.com/aws-samples/step-functions-workflows-collection/tree/main/bedrock-blog-generator/.

[24] [n. d.]. Data Quality Demo. https://github.com/astronomer/airflow-data-quality-demo/tree/v1.4.0.

[25] [n. d.]. End-to-End Temperature Forecasting with Apache Airflow Automation. https://github.com/gersongerardcruz/temperature_forecasting_airflow_automation.

[26] [n. d.]. ETL Pipeline com Apache Airflow e Python. https://github.com/DaviRic/ETL_Pipeline_Airflow/blob/604e3c0933b16897acecdae08aac20e24fad5e90/airflow_dags/etl_food_pipeline.py.

[27] [n. d.]. Example DAGs for Data Science and Machine Learning Use Cases. https://github.com/astronomer/mlflow-example/tree/v1.2.1.

[28] [n. d.]. Great Expectations Tutorial. https://github.com/astronomer/gx-tutorial/tree/1.0.0.

[29] [n. d.]. Lambda Orchestration. https://github.com/aws-samples/step-functions-workflows-collection/tree/main/lambda-orchestration-cdk/.

[30] [n. d.]. Process High-Volume Messages from Amazon SQS. https://github.com/aws-samples/step-functions-workflows-collection/tree/main/text-processing-sqs-express.

[31] [n. d.]. Stock Prediction Pipeline. https://github.com/Zhenyubbx/Stock-Prediction-Pipeline/tree/a4a5ec6c01e435145554daffc46a530de4321849.

[32] Hossein Arsham. 2020. *Automated Scheduled Reporting using Airflow.* https://home.ubalt.edu/ntsbarsh/opre640a/partIII.htm

[33] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. The serverless trilemma: function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Vancouver, BC, Canada) *(Onward! 2017)*. Association for Computing Machinery, New York, NY, USA, 89–103. doi:10.1145/3133850.3133855

[34] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. 2022. Netherite: efficient execution of serverless workflows. *Proc. VLDB Endow.* 15, 8 (April 2022), 1591–1604. doi:10.14778/3529337.3529344

[35] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* 5, 1 (Jan. 1998), 46–55. doi:10.1109/99.660313

[36] Ayush Dosajh. 2012. *Integer Optimization and the Network Models.* https://medium.com/pasarpolis-product-tech/automated-reporting-system-using-airflow-a62f2ce12e80

[37] Robert Dyer. 2013. Task fusion: improving utilization of multi-user clusters. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity* (Indianapolis, Indiana, USA) *(SPLASH '13)*. Association for Computing Machinery, New York, NY, USA, 117–118. doi:10.1145/2508075.2514878

[38] Muhammed Golec, Guneet Kaur Walia, Mohit Kumar, Felix Cuadrado, Sukhpal Singh Gill, and Steve Uhlig. 2024. Cold Start Latency in Serverless Computing: A Systematic Review, Taxonomy, and Future Directions. *ACM Comput. Surv.* 57, 3, Article 65 (Nov. 2024), 36 pages. doi:10.1145/3700875

[39] Matthew Keep. 2024. *Airflow in Action: ETL Insights from Bloomberg.* https://www.astronomer.io/blog/airflow-in-action-bloomberg/

[40] Georgia Kougka and Anastasios Gounaris. 2012. On optimizing workflows using query processing techniques. In *Proceedings of the 24th International Conference on Scientific and Statistical Database Management* (Chania, Crete, Greece) *(SSDBM'12)*. Springer-Verlag, Berlin, Heidelberg, 601–606. doi:10.1007/978-3-642-31235-9_43

[41] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. 2023. FaaSLight: General Application-level Cold-start Latency Optimization for Function-as-a-Service in Serverless Computing. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 119 (July 2023), 29 pages. doi:10.1145/3585007

[42] Manish Pandey and Young-Woo Kwon. 2024. FuncMem: Reducing Cold Start Latency in Serverless Computing Through Memory Prediction and Adaptive Task Execution. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing* (Avila, Spain) *(SAC '24)*. Association for Computing Machinery, New York, NY, USA, 131–138. doi:10.1145/3605098.3636033

[43] Juan Roldan. [n. d.]. https://gist.github.com/juanroldanbrz/468eac144bc4cb0532b53fb6a9d2bfec.

[44] Joel Scheuner and Philipp Leitner. 2019. Transpiling Applications into Optimized Serverless Orchestrations. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. 72–73. doi:10.1109/FAS-W.2019.00031

[45] Trever Schirmer, Joel Scheuner, Tobias Pfandzelter, and David Bermbach. 2024. FUSIONIZE++: Improving Serverless Application Performance Using Dynamic Task Inlining and Infrastructure Optimization. *IEEE Transactions on Cloud Computing* 12, 4 (Oct. 2024), 1172–1185. doi:10.1109/tcc.2024.3451108

[46] Biswajeet Sethi, Sourav Kanti Addya, and Soumya K. Ghosh. 2023. Alleviating Total Cold Start Latency in Serverless Applications with LRU Warm Container Approach. In *Proceedings of the 24th International Conference on Distributed Computing and Networking* (Kharagpur, India) *(ICDCN '23)*. Association for Computing Machinery, New York, NY, USA, 197–206. doi:10.1145/3571306.3571404

[47] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking Functions to Warm the Serverless Cold Start. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3423211.3425682

[48] Kent Wilken, Jack Liu, and Mark Heffernan. 2000. Optimal instruction scheduling using integer programming. *SIGPLAN Not.* 35, 5 (May 2000), 121–133. doi:10.1145/358438.349318