A Generic Efficient Scientific Workflow Engine for the Optimizations of Run-time Execution

Changxin Bai*, Junwen Liu*, Anik Tahabilder *, M M Imran*, Shiyong Lu*
Department of Computer Science
*Wayne State University**
Detroit, Michigan, USA
*Email: {changxin, junwen, tahabilderanik, mmimran, shiyong}@wayne.edu**

Dunren Che†
Department of Computer Science
Southern Illinois University†
Carbondale, Illinois, USA
Email: dche@cs.siu.edu†

2023 IEEE International Conference on Software Services Engineering (SSE) | 979-8-3503-4075-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/SSE60056.2023.00023

***Abstract*—Workflow has proven to be a highly effective computing model for a variety of scientific applications, offering flexible data types and unstructured parallelism that surpasses simple parallel execution models such as MapReduce. However, current workflow management systems in cloud computing environments experience unnecessary delays in task execution due to the separation of task execution and data transfer processes, which causes a child task to wait until all its predecessor tasks complete, rather than waiting only for necessary input data becoming ready. The goal of this paper is to eliminate the unnecessary delay of child tasks in a workflow, which is achieved through a new workflow engine architecture that separates workflow planner from workflow executor in the general framework of the DATAVIEW scientific workflow management system. This new engine architecture can be generalized and applied to other workflow systems. Our design integrates a new task release mechanism based on a data dependency model with the workflow executor of DATAVIEW. This approach enables prompt task launching once input data becomes available, instead of waiting for all predecessor tasks to finish. The architecture employs distributed algorithms for implementing the workflow executor and the task executors, performing various optimization on data movement, task movement, and communication among different subsystems. The experiments show that our new architecture based on the new task release model can significantly reduce overall execution time of a workflow in DATAVIEW.**

***Keywords*-Workflow Engine; Workflow Execution; Parallel Processing; Task Releasing; Optimization;**

1. Introduction

Recently, more and more complex analysis problems are addressed by using scientific workflow, which has been widely recognized to be a non-trivial computing paradigm [1] in numerous domains such as astronomy [2], biology [3], earthquake science [4], physics [5], patient diagnosis [6], and others [7]. Workflows offer a convenient, modular approach to the computational challenge of complex scientific projects of different domains, allowing scientists to focus on their respective domain problems rather than on the computational challenge [8].

Workflow scheduling and workflow execution are two fundamental research problems in the field of scientific workflow management systems [9]. While extensive research has been conducted on workflow scheduling to enhance the efficiency of workflow execution under various constraints [10], a key question remains unanswered: is it feasible to develop a generic workflow executor that can efficiently execute schedules generated by arbitrary workflow schedulers (or planners)? The answer to this question is paramount because 1) separation between workflow planners and workflow executors would enable independent and parallel optimizations, and 2) it could lead to the development of a generic and efficient workflow executor that can be coupled with any workflow planner to achieve efficient execution of workflow schedules. The primary objective of this research is to design a new workflow engine architecture, with a focus on developing a generic and efficient workflow executor that optimizes the run-time execution of generated workflow schedules through a new task release model.

Directed acyclic graphs (DAGs) are typically used to represent workflows, with nodes indicating tasks and edges indicating data flows [11]. However, executing a workflow in the cloud is a complex task, requiring consideration of dynamic policies for virtual machine (VM) provisioning, parallel execution of tasks on different VM instances, and parallel movement of data between VM instances. The generic nature of the proposed workflow executor means that it can accept workflow schedules generated by arbitrary workflow planners and execute them in a parallel and distributed manner, applying various optimizations and managing all low-level execution details such as task movement, data movement, synchronization, and communication among the workflow executor and the various task executors residing at different worker nodes. Therefore, a second objective of this research is to design a parallel and distributed algorithm that can coordinate the execution of a workflow schedule by various task executors under the orchestration of the workflow executor. This algorithm must be both correct and efficient, ensuring that tasks are executed in the correct order and that data is moved and synchronized in a timely and efficient manner.

Scientific workflow management systems (SWMSs) [12] [13] typically enforce the release of successor tasks only

979-8-3503-4075-4/23/$31.00 ©2023 IEEE
DOI 10.1109/SSE60056.2023.00023

Authorized licensed use limited to: Wayne State University. Downloaded on October 29,2026 at 18:26:02 UTC from IEEE Xplore. Restrictions apply.

after the completion of their predecessor tasks, even if the input datasets are available. To tackle the unnecessary delay in launching a successor task, we introduce a novel task release model that permits the workflow engine to commence task execution as soon as the input data becomes available. Additionally, we apply these principles in the development and deployment of our proposed workflow executor and task executors. This paper presents our results and makes the following contributions:

1) We propose a novel task release mode based on the identification of input data requirements, rather than the execution status of their predecessor tasks.
2) We propose a new algorithm for the workflow executor, denoted as $Workflow Executor_Beta$, which handles the submission of parallel tasks, and a new algorithms for the corresponding task executors, named $TaskExecutor_Beta$, which execute each task.
3) We conducted two use case studies in the DATAVIEW system to validate the effectiveness of our proposed architecture and task release model.

This paper is organized as follows. Section 2 presents both the traditional and the proposed task release models. Section 3 describes the new architecture of the workflow engine designed for the DATAVIEW system. Section 4 explains the implementation and algorithms of workflow executor $Workflow Executor_Beta$ and its associated task executors $TaskExecutor_Beta$. The experimental results are presented and discussed in Section 5. Other centralized workflow systems and serverless workflow systems are reviewed as related works in Section 6. Finally, Section 7 concludes this paper.

2. Task Release Model

When executing a workflow through a workflow engine, it is essential to consider the dependencies between tasks. Figure 1 depicts a simple workflow where task B and task C require only a portion of the output from task A. Task A generates the output data set $output_1, output_2$, with task B utilizing $output_1$ and task C using $output_2$. In the conventional task release strategy, task B and task C can only be dispatched to the assigned computing resources after task A has completely finished executing. This task release arrangement is known as the task dependency model, which hinders the ability to release a task as soon as its data dependencies have been resolved, which are represented by the edges in this sample workflow: $e_1 = (A, B)$ and $e_2 = (A, C)$. The task execution and data transfer are separated operation steps during workflow execution under the task dependency model. As a result, there is no mechanism available to inform the workflow engine that the currently generated output data from a task can satisfy and be readily transferred to its direct successor tasks. The output data of a task can only be transferred to its successor tasks once the finish status of the current task is reachedd and returned to the workflow executor.

To overcome the limitation of task execution and data transfer separation, we propose a novel task release strategy, referred to as the "data dependency model". This strategy accelerates the task releasing process once the required input data of a task becomes available, which is expressed through the addition of the output data attribute in an edge such as $e_1 = (A, output_1, B)$ and $e_2 = (A, output_2, C)$ in Figure 2. Under this model, the workflow engine is notified to release a child task as soon as its input data becomes available, rather than waiting for the predecessor task's execution status becomes as $finished$. As a result, task release is expedited, and the total execution time of the workflow is thus reduced. This approach tightens the execution of dependent tasks in a workflow and the data transfer between tasks during the lifecycle of a workflow's execution.

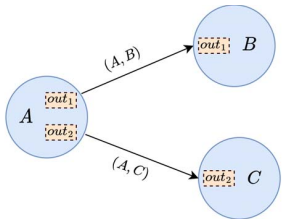


Figure 1: Task Dependency Model.

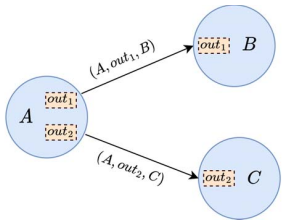


Figure 2: Data Dependency Model.

The detailed implementation of the data dependency model introduced above will be manifested in a later section when we present the workflow executor $Workflow Executor_Beta$ and its associated task executor $TaskExecutor_Beta$.

3. The Architecture for Workflow Engine

A previous paper of the DATAVIEW project by Kashlev et al. [12] proposed a reference architecture for a big data workflow management system, which was implemented in our previous version of DATAVIEW, optimizing the workflow execution process in two aspects: at the workflow planner level and the workflow executor level. Based on this prior work, we designed a new workflow engine architecture as shown in Figure 3 which depicts the three-layer components

Authorized licensed use limited to: Wayne State University. Downloaded on October 29,2026 at 18:26:02 UTC from IEEE Xplore. Restrictions apply.

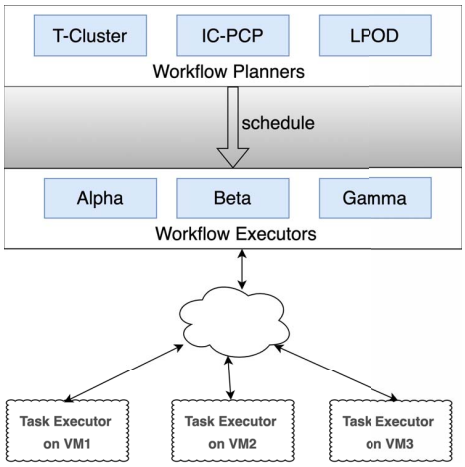


Figure 3: Workflow Engine Architecture in Cloud.

of the generic workflow engine, i.e., workflow planners, a workflow executors, and task executors.

Typically, a workflow executor operates on a master node, while various task executors run on separate worker nodes which are VMs in a cloud execution environment. Workflow planners and executors are separate but collaborative processes, while the task executors are associated with a specific workflow executor. The deployment of a task executor on a VM can be performed statically or dynamically, either through code transfer from a workflow executor or by launching a new VM with a pre-defined image.

More Specially, a workflow planner such as T-Cluster [14], IC-PCP [15] or LPOD [16] takes workflow specific information derived from the workflow graph and a given performance metric under some constrains and generates a workflow schedule. Then a workflow executor takes over the schedule and orchestrates the execution according to the schedule by performing a series of operations, including provisioning and de-provisioning of VMs, and invoking the task execution on a task executor at the scheduled time point at a worker node.

4. Workflow Executor and Task Executors Implementation

The workflow executor is an essential component of the workflow engine module, which dispatches workflow tasks to the best fit VM instances based on a workflow schedule sch generated by a workflow planner and manages the life cycle of each VM instance. A recently implemented new workflow executor for DATAVIEW, named $WorkflowExecutor_Beta$, and its corresponding task executors, named $TaskExecutor_Beta$, are described in this section.

This new functionality incarnated in the new workflow executor allows the executor to release a child task as soon as its input data is available instead of waiting for the predecessor task's execution to complete. This speeds up the task release process and reduces the overall execution time of the workflow. Additionally, the workflow executor monitors the progress of each task's execution and manages the compute resources dynamically by scaling up/down the number of VM instances depending on the workload of the workflow. Overall, the new workflow executor and its corresponding task executors in DATAVIEW provide a reliable and efficient way to execute large-scale workflows in a distributed environment.

In order to release a task to its corresponding VM instance, $WorkflowExecutor_Beta$ uses three abstractions:

- A *Task Schedule* refers an abstract entity that accommodates a workflow task, which includes the task ID, the task name, all direct child tasks, lists of incoming/outgoing data channels, and the IP of the designated VM of this task.
- A *Local Schedule* is created per each provisioned VM to host all the task schedules assigned to run on the VM.
- A *Global Schedule* is a collective object that hosts all local schedules on all VMs, generated by a workflow planner for a workflow.

Workflow executor, $WorkflowExecutor_Beta$, utilizes task schedule objects to dispatch tasks to their designated VM instances in parallel once their input data is available. Each object contains a $numOfIncomingDataUnready$ attribute, indicating whether the current task is ready for release (execution) or not.

The implementation of $WorkflowExecutor_Beta$ is detailed in Algorithm 1. It takes as input the workflow's global schedule $gsch$, which consists of a list of local schedules: $[lsch_1, \cdots, lsch_n]$. Each local schedule $lsch$ is realized by a LocalScheduleRun thread object, denoted as $lschr$, which checks the task status. Each task schedule $tsch$ is handled by a TaskRun object, denoted as tr, which submits each task to the assigned VM instance. Specifically, a thread $lschr_i$ is created for each local schedule (line 2), and a TaskRun object is created for each task included in the local schedule. The TaskRun object is initialized with the number of unready input port data of each task (lines 3-7).

The threads of $WorkflowExecutor_Beta$ are executed in parallel to perform the execution of the ready tasks, as specified in Algorithm1. First, a TaskRun object is created for each local task in each local schedule by initializing the number of unready input ports data of each task, which is a part of the Task Schedule object (lines $3-7$). Subsequently, all threads are executed in parallel to perform the execution of the ready tasks (lines $9-37$). For each thread corresponding to a local schedule and a VM, all tasks are topologically sorted and stored in a variable ts_i (line 10). The readiness of each task to run is checked by verifying the number of unready input ports data for the task. If there is any unready input ports data, the current

Authorized licensed use limited to: Wayne State University. Downloaded on October 29,2026 at 18:26:02 UTC from IEEE Xplore. Restrictions apply.

thread is put on hold (lines $13-17$). However, if the task's input ports data is ready, and the execution code is prepared, then the task is submitted to its corresponding VM instance for execution (lines 12). A socket connection is established between $Workflow Executor_Beta$ and $TaskExecutor_Beta$ to obtain the object input/output streams to the $TaskExecutor_Beta$ (lines $18-19$). The $TaskExecutor_Beta$ executes task t_j on its assigned VM instance $M(t_j)$, which is determined by the mapping function between task and its assigned VM instance, by writing the task specification information to $TaskExecutor_Beta$ (line 20). During the execution period of task t_j, the list of task IDs that represent data movement from input ports to output ports are obtained from $TaskExecutor_Beta$ (line 22). The number of unready input ports data of each task represented in the TaskRun object tr_k is then decremented by 1 for each TaskRun retrieved based on a task ID in the For loop (line 25). Once the child task in tr_k is ready, the thread containing the task run of tr_k is awakened via the $signal()$ method (lines $26-33$), which is lock-protected.

Task executor, $TaskExecutor_Beta$, is responsible for task execution and data movement between VM instances as illustrated in Algorithm2. In the do-while loop, an output stream object and an input stream object are created for communication with the workflow executor $Workflow Executor_Beta$ (lines $2-3$). Next, the task specification information is read from the input stream object (line 4) and the task name is retrieved (line 5). Then, a new task instance t_j is created and executed. During task execution, output data is written to dic (line 6). The $DataThread$ algorithm continuously checks for generated data and moves it to the corresponding VM instance. Simultaneously, each task ID representing a child task of t_j with one input data ready is added to the message queue q (line 7). The $NotationThread$ algorithm continuously checks the queue elements and writes them to $Workflow Executor_Beta$ (line 8).

At last, the task specification information is added into q, which is signaled as the task execution succeeded status. The do-while loop continues to run until the instance is deprovisioned. While the tasks in a workflow have different start times, this arrangement assures each task get instantly executed once it is ready.

5. Evaluation

The evaluation of the $Workflow Executor_Beta$ and $TaskExecutor_Beta$ implementations was carried out on two real-world workflow applications. Our experimental results demonstrate that the data dependency model, which ensures tight integration between task execution and data movement processes, can reduce the total workflow execution time of workflows by an average of 10% and 40%, compared to conventional execution engines based only on the task dependency model.

Algorithm 1: WorkflowExecutor_Beta

```
   Input  : GlobalSchedule gsch = [lsch_1, ..., lsch_n]
   Output : Exit code
 1 foreach lsch_i ∈ gsch do
 2     create a LocalScheduleRun lschr_i for lsch_i;
 3     foreach j ∈ lsch_i.length() do
 4         Taskschedule tsch ← lsch_i[j] ;
 5         create TaskRun tr_j;
 6         tr_j.numDataUnready ←
             tsch.getIncomingDataChannels().size() ;
 7     end
 8 end
 9 forall LocalScheduleRun lschr_i in parallel do
10     ts_i ← all TaskRuns in lschr_i in order ;
11     foreach TaskRun tr_j ∈ ts_i do
12         send the code of t_j to VM, M(t_j);
13         while tr_j.numDataUnready ≠ 0 do
14             lschr_i.mLock.lock() ;
15             lschr_i.mReadyCon.await();
16             lschr_i.mLock.unlock();
17         end
18         out ← output stream to TaskExecutor_Beta;
19         in ← input stream to TaskExecutor_Beta;
20         invoke the TaskExecutor_Beta at VM, M(t_j)
             to execute the task t_j by out.write(taskSpec);
21         while t_j's execution status is not successful do
22             list ← in.read() ;
23             foreach taskID str ∈ list do
24                 get taskRun tr_k from str;
25                 tr_k.numDataUnready − −;
26                 if tr_k.numDataUnready == 0 then
27                     tr_k.ownerLocalScheduleRun.mLock
28                     .lock();
29                     tr_k.ownerLocalScheduleRun.mReadyCon
30                     .signal();
31                     tr_k.ownerLocalScheduleRun.mLock
32                     .unlock();
33                 end
34             end
35         end
36     end
37 end
38 return SUCCESS;
```

A. Experiment Setup

We have implemented two real-world workflows using the DATAVIEW API and evaluated their performance under two different task release models. To support parallel data movement in $TaskExecutor_Beta$, we selected the t2.xlarge instance type on AWS EC2, which has 4 vCPUs, 16GB memory, and moderate network performance. All experiments were conducted with a 2GB JVM heap memory space. We used execution time as a criterion to demonstrate the reported results, which were calculated based on the provenance information collected from the master node.

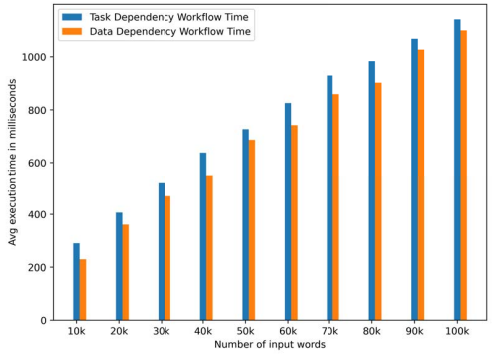
B. Word Count Workflow

The word count workflow, an example application utilizing the Map-Reduce [17] structure, was utilized in our

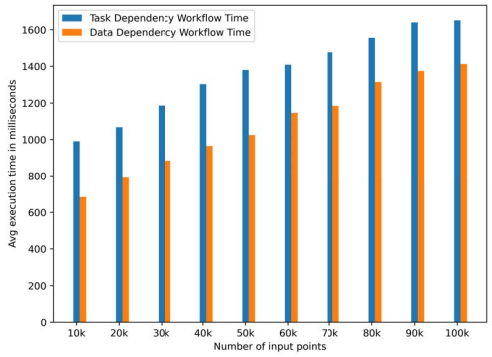
Authorized licensed use limited to: Wayne State University. Downloaded on October 29,2026 at 18:26:02 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2: TaskExecutor_Beta

do
  $out \leftarrow$ Output Stream to $Workflow Executor_Beta$;
  $in \leftarrow$ Input Stream from $Workflow Executor_Beta$;
  $taskSpec \leftarrow in.read()$;
  task $t_j's$ taskName $\leftarrow taskSpec.get$(taskName);
  execute task t_j and write data to dic ;
  $DataThread(dic).start()$;
  $NotationThread(out, q).start()$;
  $q.add(taskSpec)$;
while *True*;



(a) Words Count Workflow.



(b) Distributed K-means Workflow.

Figure 4: Execution time of running different workflows in different task release models with different input datasets.

experiments. We created a concrete workflow comprising 10 tasks, including two Split tasks, two Map tasks, three Shuffle tasks, and three Reduce tasks. We generated $100,000 - 1,000,000$ words randomly as the workflow input. Figure 4a compares the execution time under different workflow schedules and data movement strategies. Notably, the data dependency model yielded a shorter execution time than the task dependency model. Our results showed that the implementation of the data dependency model decreased the average workflow execution time by 10%.

C. The Distributed K-means Workflow

We have also devised a distributed K-means workflow in DATAVIEW by specifying the number of iteration loops beforehand. Each iteration comprises three tasks, namely DisCalculateCentroidStep1, DisCalculateCentroid, and DisReassignCluster. The DisKMeansInitialization task is carried out only once to partition the input data into several clusters, as per user-defined input. For the experiment, we randomly generated points with coordinates x and y, ranging from 100,000 to 1,000,000. The results depicted in Figure 4b highlight that the execution time under the data dependency model is shorter than the task dependency model. Additionally, we observed that the new task release model reduced the average workflow execution time by 40%. It is worth noting that the DisCalculateCentroidStep1 task is more complex than the split task, hence taking a longer time to generate data and yielding greater savings in terms of execution time.

6. Related Work

Workflow systems emerged from resolving specific data analysis problems, resulting in the creation of new tools [18]. Consequently, different optimization technologies are employed in various components, such as resource management, workflow portability, false recovery, and more [19]. Modern workflow management systems have extended the execution environment beyond local infrastructures to include cloud facilities, enabling parallelization throughout the workflow execution process. This parallelization encompasses data parallelization, task submission, and intermediate data transfer [20]. An essential aspect of optimizing workflow execution lies in the workflow schedule component, which ensures the fulfillment of different quality of service requirements. Listed below are several workflow engines examples, supporting diverse forms of parallelism.

Pegasus utilizes Condor's Directed Acyclic Graph Manager (DAGMan) as its foundation, representing workflows as DAGs [21]. It submits individual tasks in the DAG to HTCondor one by one, employing a task queue for task parallelism execution. To optimize the workflow structure, Pegasus clusters tasks with short-running times and converts the workflow into an executable DAGMan workflow. However, this approach may impede certain optimizations related to path-based workflow scheduling processes [16]. In contrast to DATAVIEW [12], Pegasus generates location-specific intermediate files, lacking a memory data transfer mechanism.

Hadoop [22] is implemented for distributed processing of large data sets across cluster of computer by using predefined Map-Reduced models, which is friendly to IO-intensive workflows. However, the Hadoop cannot execute customized workflow structure, which limit the usability for scientific domain researchers. Benefit from the HDFS system

Authorized licensed use limited to: Wayne State University. Downloaded on October 29,2026 at 18:26:02 UTC from IEEE Xplore. Restrictions apply.

in the cluster environment, Hadoop can handle the intermediate data efficiently. Moving the execution environment from cluster to cloud will delegate the performance.

Compared with Hadoop, Spark [23] was implemented to have a in-memory storage of intermediate data by using the Resilient Distributed Datasets (RDDs), which makes data partitioned and processed in a parallel manner. Executing a workflow with Spark engine cannot benefit from RDDs since each task in a workflow is treated as a blackbox. What's more, a workflow schedule cannot directly applied since the input format is in the Spark RDDs.

Recently, microservices have attracted more attention from both academia and industry due to their scalability and flexibility [24]. A serverless infrastructure, represented by Google Cloud Functions [1] or AWS Lambda [2], has emerged as a popular option for executing scientific workflows [25]. However, this architecture relies on intermediate data storage compared to service-based architectures.

7. Conclusions and future work

This paper presents the new design of the workflow engine of the DATAVIEW workflow system, aiming to boost workflow execution performance by minimizing the waiting gap between consecutive workflow tasks. Our experimental results demonstrate the effectiveness and efficiency of design and algorithms. Current SWfMSs still rely on a centralized workflow engine on the master node to manage task execution across multiple worker nodes, leading to extra communication overhead. In the future, we shall consider scheduling sub-workflows directly to worker nodes to reduce this overhead.

Acknowledgement

This work is partially supported by National Science Foundation under grants CNS-1747095 and OAC-1738929.

References

[1] L.-J. L. Zhang, "Quality-driven service and workflow management," *IEEE Transactions on Services Computing*, vol. 4, no. 02, pp. 84–84, 2011.

[2] J.-S. Vöckler, G. Juve, E. Deelman, M. Rynge, and B. Berriman, "Experiences using cloud computing for a scientific workflow application," in *Proceedings of the 2nd international workshop on Scientific cloud computing*. ACM, 2011, pp. 15–24.

[3] M. Abouelhoda, S. A. Issa, and M. Ghanem, "Tavaxy: Integrating Taverna and Galaxy workflows with cloud computing support," *BMC bioinformatics*, vol. 13, no. 1, pp. 1–19, 2012.

[4] M. Atkinson, C. S. Liew, M. Galea, P. Martin, A. Krause, A. Mouat, O. Corcho, and D. Snelling, "Data-intensive architecture for scientific knowledge discovery," *Distributed and Parallel Databases*, vol. 30, no. 5-6, pp. 307–324, 2012.

[5] D. A. Brown, P. R. Brady, A. Dietz, J. Cao, B. Johnson, and J. McNabb, "A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis," *Workflows for e-Science: Scientific workflows for grids*, pp. 39–59, 2007.

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

[2] https://aws.amazon.com/lambda/

[6] I. Ahmed, S. Lu, C. Bai, and F. A. Bhuyan, "Diagnosis recommendation using machine learning scientific workflows," in *Proceedings of the 2018 IEEE International Congress on Big Data*. IEEE, 2018, pp. 82–90.

[7] D.-s. Lee, Y.-T. Chen, and S.-L. Chao, "Universal workflow of artificial intelligence for energy saving," *Energy Reports*, vol. 8, pp. 1602–1633, 2022.

[8] E. Deelman, K. Vahi, M. Rynge, R. Mayani, R. F. da Silva, G. Papadimitriou, and M. Livny, "The evolution of the Pegasus workflow management software," *Computing in Science & Engineering*, vol. 21, no. 4, pp. 22–36, 2019.

[9] K. Lee, N. W. Paton, R. Sakellariou, E. Deelman, A. A. Fernandes, and G. Mehta, "Adaptive workflow processing and execution in pegasus," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 16, pp. 1965–1981, 2009.

[10] J. Liu, S. Lu, and D. Che, "A survey of modern scientific workflow scheduling algorithms and systems in the era of big data," in *Proceedings of the 2020 IEEE International Conference on Services Computing*. IEEE, 2020, pp. 132–141.

[11] J. Yu and R. Buyya, "A taxonomy of scientific workflow systems for grid computing," *ACM Sigmod Record*, vol. 34, no. 3, pp. 44–49, 2005.

[12] A. Kashlev, S. Lu, and A. Mohan, "Big data workflows: A reference architecture and the DATAVIEW system," *Services Transactions on Big Data*, vol. 4, no. 1, pp. 1–19, 2017.

[13] J. Liu, Z. Xiao, S. Lu, D. Che, M. Dong, and C. Bai, "Infrastructure-level support for gpu-enabled deep learning in DATAVIEW," *Future Generation Computer Systems*, vol. 141, pp. 723–737, 2023.

[14] A. Mohan, M. Ebrahimi, S. Lu, and A. Kotov, "A nosql data model for scalable big data workflow execution," in *Proceedings of the 2016 IEEE International Congress on Big Data*. IEEE, 2016, pp. 52–59.

[15] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 158–169, 2013.

[16] C. Bai, S. Lu, I. Ahmed, D. Che, and A. Mohan, "LPOD: A local path based optimized scheduling algorithm for deadline-constrained big data workflows in the cloud," in *Proceedings of the 2019 IEEE International Congress on Big Data*. IEEE, 2019, pp. 35–44.

[17] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[18] R. F. Da Silva, H. Casanova, K. Chard, I. Altintas, R. M. Badia, B. Balis, T. Coleman, F. Coppens, F. Di Natale, B. Enders *et al.*, "A community roadmap for scientific workflows research and development," in *Proceedings of the 2021 IEEE Workshop on Workflows in Support of Large-Scale Science*. IEEE, 2021, pp. 81–90.

[19] L. Wratten, A. Wilm, and J. Göke, "Reproducible, scalable, and shareable analysis pipelines with bioinformatics workflow managers," *Nature methods*, vol. 18, no. 10, pp. 1161–1168, 2021.

[20] L. Belcastro, F. Marozzo, and D. Talia, "Programming models and systems for big data analysis," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 34, no. 6, pp. 632–652, 2019.

[21] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger, "Workflow management in condor," *Workflows for e-Science: Scientific Workflows for Grids*, pp. 357–375, 2007.

[22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th symposium on mass storage systems and technologies*. IEEE, 2010, pp. 1–10.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica *et al.*, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[24] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Computing Surveys*, vol. 54, no. 10s, pp. 1–34, 2022.

[25] A. Kumari, M. K. Patra, B. Sahoo, and R. K. Behera, "Resource optimization in performance modeling for serverless application," *International Journal of Information Technology*, vol. 14, no. 6, pp. 2867–2875, 2022.

Authorized licensed use limited to: Wayne State University. Downloaded on October 29,2026 at 18:26:02 UTC from IEEE Xplore. Restrictions apply.