Dynamic Tracking, MLOps, and Workflow Integration: Enabling Transparent Reproducibility in Machine Learning

Hamza Safri[*] [†],[§] George Papadimitriou[‡], Ewa Deelman[‡],

[*] Berger-levrault, Toulouse, France
[†] Grenoble University, Grenoble, France
[‡] University of Southern California, Los Angeles, USA
[§] INRIA, Grenoble, France
E-mail: hamza.safri@{univ-grenoble-alpes.fr, berger-levrault.com, inria.fr}, {georgpap, deelman}@isi.edu

Abstract*—Workflow management systems (WMS) provide a robust solution for automating and ensuring the reproducibility of scientific and engineering experiments. However, reproducing machine learning (ML) experiments requires replicating every aspect of the process, including code implementation, workflow execution, data, and the execution environment. Traditionally, tracking these components is done manually, if at all, before execution. In this work, we propose an approach for on-demand and dynamic tracking of ML workflows. Our approach extends the ML workflow automatically and introduces steps for tracking, organizing, and versioning all elements, such as code, data, the main workflow steps and the execution environment for each job. This tracking approach includes two modes: a custom mode, where user-tagged elements will be tracked by the WMS, and an automatic mode, where the WMS automatically tracks and organizes all necessary elements. We implemented this solution by extending the Pegasus-WMS system and tested it on two types of workflows: traditional scientific ML pipelines and Federated Learning (FL) applications. Our findings demonstrate that this tracking approach does not interfere with the normal execution of user-designed workflows and the execution time. Additionally, we show how this approach can integrate a WMS with versioned data in remote storage (such as S3 or Google Drive) and ML lifecycle solutions like MLflow, ensuring reproducibility and transparency of the computational experiments.

***Index Terms*—Workflow Management System, Tracking, Versionning, MLops, Machine learning**

I. Introduction

As ML continues to emerge as a driver of innovation, automating ML computations using WMS becomes crucial. Automation provides the opportunity to accelerate and optimize every step of the process, reduce errors, and alleviate practitioners having to perform repetitive manual tasks.

An ML workflow is a sequence of organized and interconnected steps that include data preparation, model selection and training, performance evaluation, and model deployment. All these require continuous maintenance in a production environment. These workflows are applied in various industrial applications, as well as in research, to execute and automate training and evaluation tasks, fine-tuning, and promising the reproducibility of experiments.

However, the models, which are the main artifacts of the ML workflows, can be affected by factors such as the input data being used, the hardware and the version of the software (environment), and the training parameters. As a result, to guarantee reproducible outcomes, a WMS needs to monitor and track closely the ML workflows. Apart from monitoring and ensuring the correct execution of the workflow steps, a WMS needs to collect information about the input and output data of the tasks, collect information about the execution environment, and correlate this information with the resulting models and their training parameters, generating a fully reproducible workflow signature for each ML model.

Most workflow management solutions [1] [2] do not meet the needs of ML workflows, which poses a significant challenge for data scientists. Even MLOps, which seeks to bridge this gap by integrating DevOps principles into the ML lifecycle [3], is not sufficient. Additionally, even with existing solutions in this area, we observe limitations in the interface, such as the case of DVC [4]. The most commonly used tool in data versioning, lacks sufficient programmability because of incomplete APIs and configurations required for data tracking.

Integrating MLOps into scientific ML workflows, enhances the capabilities of WMS, enabling comprehensive tracking of various experiments beyond ML workflows. This adaptation also organizes and structures the outputs of workflow experiments, facilitating result interpretation. Controlling versioning becomes paramount, enabling the collection of comprehensive versioning information and enhancing the human exploitation of these insights.

Therefore, in this paper, we present our vision for integrating MLOps into a WMS. The paper makes the following contributions: (i) creation of a distributed and automated approach to data versioning tailored for WMS based on tags; (ii) integration of the MLflow ML model lifecycle management solutions with a WMS; and (iii) the management of experiments through the ability to create strong links between workflow instances, the utilized data, and the experiment results while ensuring transparent reproducibility based on metadata.

The paper is organized into several sections. It begins by discussing the current state-of-the-art in ML workflows and MLOps. Following this, the paper delves into the description of our approach, starting with tracking in ML workflows and progressing toward experiment tracking. Subsequent sections detail experiments aimed at assessing the efficiency of our approach with two types of ML: traditional ML and federated ML applications. The paper concludes with a summary of the findings and highlights potential avenues for future research.

II. Related work

A WMS is crucial for managing and optimizing complex ML workflows. It automates tasks, reduces human and execution errors, ensures efficient resource use, and provides real-time monitoring, all of which are essential for effective ML model development and deployment.

In the ML domain, WMSs have become essential for automating ML pipelines. Krawczuk et al. [5] explored how different data management configurations impact scientific ML workflows using Pegasus-WMS, showcasing its effectiveness across diverse datasets. Meanwhile, Krawczuk et al. [6] utilized Pegasus-WMS to implement deep learning algorithms for extracting disaster-related information from social media posts, enhancing situational awareness for first responders through their CrisisFlow workflow. Additionally, Nguyen et al. [7] proposed integrating ML modules into the Kepler tool to simplify ML workflow manipulation, advocating for broader integration to enhance usability.

With the objective of automating various stages of ML, Nvidia introduced the NVIDIA FLARE framework, which specialized in FL but requires complex manual configuration [8]. Carreira et al. proposed ”Cirrus,” a serverless framework streamlining end-to-end management of data center resources for ML workflows [9]. Despite positive aspects, WMSs face limitations in fine-tuning model parameters, handling multiple rounds, managing models at each step, and ensuring reproducibility. The MLOps approach has evolved to address these challenges, focusing on deploying, managing, and sustaining ML models in real-world production settings [3].

Dominik Kreuzberger et al.'s paper [3] defines MLOps as a paradigm focusing on best practices for end-to-end ML product lifecycle management. It outlines roles, interactions, and technical aspects bridging development and operations, emphasizing CI/CD automation, versioning, collaboration, continuous training and evaluation, metadata tracking, monitoring, and feedback loops. Tools such as MLflow [10], Kubeflow [11], DVC [4], Airflow [2], SageMaker [12], and AzureML [13] are highlighted. Upon analyzing the end-to-end MLOps architecture and workflow presented in various papers, it becomes apparent that for scientific ML workflows utilizing a WMS to ensure seamless execution, that system needs to have the capacity for model tracking and registry, ML metadata, data, and workflow versioning. These components collectively contribute to covering all aspects of the ML pipeline, from execution to reproducibility, and results structuring.

In the realm of model tracking and versioning, various solutions have been developed to streamline model and data management. Vartak et al. propose ModelDB [14], while Zaharia et al. introduce MLflow [10]; both frameworks facilitate automatic tracking and unified storage of models in popular ML environments. MLflow is comprehensive, supporting experiment tracking, project packaging, and model deployment, whereas ModelDB focuses on model versioning and management. Additionally, Schelter et al. present a system for automating metadata and provenance tracking in ML experiments [15], integrated with popular frameworks to manage metadata and lineage effectively. In data versioning, the term ”data” broadly includes all files, transformations, and inputs throughout the workflow lifecycle. Solutions like Woodman et al.'s integration of provenance with service and workflow versioning [16] and Miao et al.'s unified provenance and metadata management system [17] address these challenges. Data Version Control (DVC) stands out for enabling reproducibility and managing ML experiment complexities [4], offering features like command-line tools, dataset versioning, Git integration, and remote storage. However, integrating DVC with WMS is problematic due to its command-line-centric configuration and lack of APIs, highlighting the need for better compatibility and interoperability to streamline ML workflow management. Despite the availability of numerous MLOps solutions, there is a significant gap in integrating these tools for data scientists, hindering the creation of a unified and flexible machine-learning workflow. The absence of well-established connections between these solutions makes workflow construction intricate and less customizable. This integration challenge is exacerbated by the crucial need for smooth integration between experiments, models, metrics, and data stored in different repositories, impacting the efficiency of data scientists and the overall effectiveness of the ML process.

In this paper, we propose an approach based on tagging workflow elements to indicate information needed to be captured to ensure reproducibility of ML workflows. We leverage the Pegasus-WMS tool as our WMS, integrating its interface with MLflow for model and experiment management. Additionally, we develop a distributed data versioning approach tailored to the WMS, facilitating on-demand creation of versions for various data and files. This is crucial for accurately reproducing experiments and organizing data and models.

III. From scientific ML workflows to Tracked scientific ML workflows

A. ML Pipeline

Before describing our approach, it is important to define the 5 steps of a typical ML pipeline (Fig. 1). **(1) Data Acquisition:** Gathering relevant data from diverse sources while ensuring integrity, compliance, and permissions; **(2) Data Processing:** Preprocessing raw data through cleaning, integration, and feature engineering to prepare it for analysis; **(3) Model Building and Training:** Selecting, training, and validating ML models based on the prepared data to optimize performance; **(4) Evaluation and Interpretation:** Assessing model performance using metrics and interpreting predictions to gain insights into underlying data relationships; **(5) Model Packaging and Deployment:** Integrating and deploying trained models into real-world applications to support decision-making or discovery, ensuring usability and scalability.

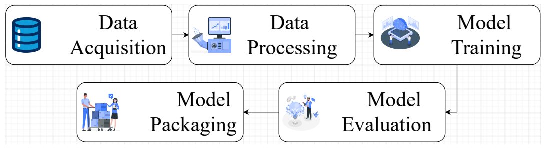


Fig. 1: The five stages of a typical ML pipeline.

The data pipeline undergoes several transformations, starting from collection, through processing, to feature extraction. Fine hyperparameter tuning adds complexity by increasing file transformations, trained models, model parameters, and metrics. Although using a WMS streamlines execution, managing and organizing the outputs remains a concern. As the pipeline becomes more complex with additional parameter combinations, understanding, and managing the outputs and assessing the produced artifacts becomes more challenging for users.

B. Tracked scientific ML workflows

ML workflows leveraging a WMS such as Pegasus-WMS [1], Apache Airflow [2], or Kubeflow [11] employ Directed Acyclic Graphs (DAGs) to orchestrate tasks. These DAGs encode the dependencies between stages, enhancing automation and reproducibility. Reproducibility in ML ensures consistent results across experiments, achieved by tracking relevant aspects like input data, training and test data, hyperparameters, model architecture, and evaluation metrics. This transparency fosters trust in research and applications by enabling repeatability of analyses.

Fig. 2 illustrates a comparison between the current implementation of ML workflows and the desired Tracked ML workflows. In the existing ML workflow implementation, high-level jobs for ML steps (III-A) are present, with tracking or versioning operations embedded within these jobs or manually added by users, leading to increased complexity. Conversely, the desired implementation of automatically Tracked ML workflows, as depicted in the right section of the figure, dynamically creates new jobs based on user-defined needs within the WMS. These jobs are dedicated to file/data tracking, generating metadata files for each job containing key information like checksum, storage type, and modification details. These metadata files are first combined into a single file, organized by the workflow's unique identifier (ID). This file is then added to a central metadata file at the WMS's central node, organized first by workflow name and then by workflow ID, and includes metadata from all previously tracked workflows submitted from that central node.

The data tracking jobs are executed in parallel with the main jobs defined by the user in the workflow, aiming to minimize their impact on the overall workflow execution time.

In the next sections we present how this vision works and our efforts to implement it and integrate it with the Pegasus WMS [1]. Additionally we discuss the versioning feature using a Version Control System (VCS) and how it ties in with the experiments tracking system.

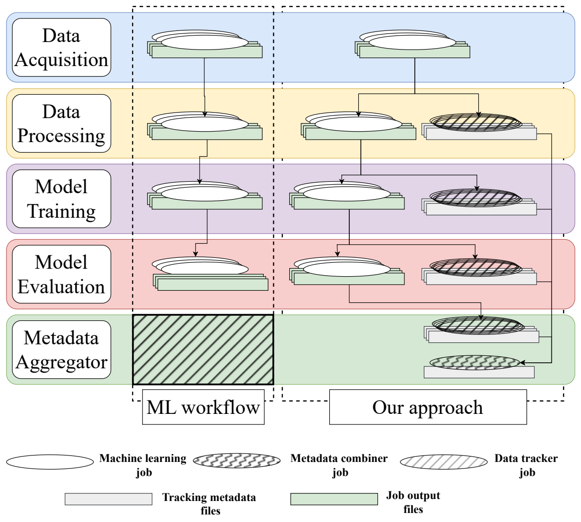


Fig. 2: Our approach extends the ML workflow with additional tracking jobs to ensure reproducibility of results.

C. Pegasus-WMS Python API

Pegasus-WMS is an open-source software framework that streamlines the execution of scientific workflows across distributed computing environments by automating computational tasks, data movement, and resource allocation. Its main concepts include: **(1) Transformations:** individual computational tasks specifying executable programs or scripts with their input and output files; **(2) Sites:** computing resources where tasks are executed, such as campus or local machines, each with specific properties; **(3) Replicas:** copies of data files used by tasks, tracked in a catalog to manage data locations and movement efficiently.

These components ensure portable, scalable and reliable workflow execution. The Pegasus-WMS API, accessible via Python, R, or Java, allows developers and scientists to programmatically define, submit, monitor, and manage workflows, accelerating scientific discovery.

D. Data Versioning Stage

The Pegasus Python API facilitates close collaboration between replicas and files. The File object serves as a representation of files within the workflow, while the Replica Catalog keeps tabs on their storage locations. Developers utilize the File object to find the locations of file, which may be replicated in the environment. Furthermore, the File object acts as the foundation for file and ML data tracking. This operation enables users to tag files for tracking, extending beyond managing locations, ensuring data availability, staging data, and ensuring data integrity. By incorporating tags in the Pegasus workflow generator using the Python API, users can effortlessly track various file types within workflows. This approach introduces different types of tags during workflow implementation, aiding in structuring versioning and facilitating workflow analysis.

[1]https://swarmourr.github.io/pegasus/index.html

In this implementation, we define (based on the workflow structure) three customizable boolean tags for tracking data within workflows:

- **input_track:** Assigned to files considered by the user as input data for the workflow, such as input data before processing.
- **output_track:** Used to track files generated by the workflow, whether intermediate or final results. For instance, in ML, this could include the trained model or metrics files.
- **wf_track:** This tag is employed for versioning the executed instance of the workflow.

These tags can be used together or independently to version the desired products. By combining the three tags, the WMS provides complete versioning of the workflow, which guarantees the reproducbility of the ML workflow artifacts.

These tags extend the definitions of workflows, files, and jobs in the Pegasus Python API, providing a comprehensive mechanism for tracking various files used by these components of Pegasus. These tags dynamically introduce new jobs for versioning files without altering the original job order defined by the user in the abstract workflow. Each versioning job generates a metadata file capturing details described in Table I. This metadata file is structured to represent each instance, identified by a unique ID, within the workflow experimentation.

TABLE I: Workflow and file information.

Field	**Description**
Workflow Name	Name of the workflow Experimentation
Workflow ID	Unique identifier for a run in the workflow experimentation workflow
File Name	Name of the file versioned by the job
Bucket Storage URL	Storage URL in a bucket system
Google Drive Remote URL	Remote storage URL for Google Drive
Last Modification	Timestamp indicating last modification time
Path	File path in the local filesystem
Size	Size of the file in bytes
Timestamp	Timestamp when the file was last modified
Type	Type of the file (e.g., Inputs, Outputs, wf)
Version	Version identifier associated with the file
env	Information about the execution environment
mlflow_url	Direct url to the MLflow run associated with the job

Upon workflow completion, a job aggregates all metadata files, consolidating them in the central node of the workflow submission system. This central metadata file can be managed using versioning tools like Git or GitHub for traceability and reproducibility.

Remote storage mirrors the metadata file structure, simplifying experiment reproduction. Each experiment is represented by a folder containing subfolders for individual runs, including workflow, input data, and output data. A new command in Pegasus enables users to copy all files from remote storage, including the workflow, facilitating consistent results and reproducibility. This approach ensures that experiments can be rerun with the same data and identified by the same run ID, mitigating risks of version mismatch or data modification.

E. Model tracking Version

To complete the tracking of the ML workflow we need to introduce a solution for managing the lifecycle of ML projects. The data versioning approach outlined in section III-D provides extensive information regarding the data utilized in experimentation, which essentially represents the execution of a workflow. Additionally, these experiments yield a plethora of metrics, figures, and parameters. Therefore, to ensure seamless connectivity between experimental data and results, it is imperative to log not only these metrics but also the path of data on remote storage that was generated by data tracking jobs.

To ensure seamless integration between data versioning tasks and ML lifecycle management, a direct link is essential. This link necessitates a shared unique identifier among all involved parties, alongside an efficient WMS. In this scenario, the WMS identifies workflows through two key elements, accessible to all actors: the workflow's name and its execution ID (Experiments) of the workflow instance. Consequently, in experiments management, we arrange experiments based on workflows at the primary level and their respective IDs at the secondary level. MLFlow [10] can help us organize these data in easy-to-manage, user-friendly dashboards. Making the WMS MLflow-aware will enable the complete tracking of ML workflows. In alignment with the previous discussion on data tracking, users must designate the jobs that will utilize the tracking solution. Users can assign two distinct tags:

– **job_auto**: In this scenario, the parameters for identifying experiments will be automatically extracted from the workflow generator, predominantly relying on job names for identification purposes. For that, Pegasus-WMS, as described in Fig. 3, creates an experiment based on the name of the workflow. Once created, each time this workflow is executed, a parent run is created in MLflow. The name of this parent run is a concatenation of the date and time, plus the name of the workflow. This parent run contains all the child runs corresponding to the tracked jobs by MLflow.

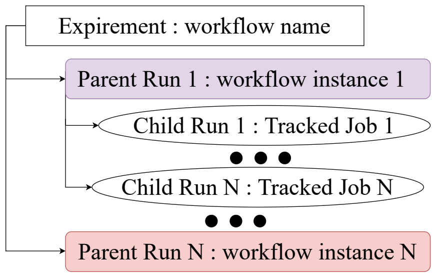
– **job_custom**: Conversely, users have the flexibility to define custom names for each job run and experiment, facilitating later extraction of results.

Within the workflow, each job's output is configured by the user to be logged in the experimentation tracking solution. To ensure a comprehensive overview of the experiments within the tracking solution, default parameters will be automatically included to extract relevant data if tracking is enabled. For each run, the workflow ID and the paths for all utilized files, along with the metadata file generated by the tracking jobs, will be appended as metadata and artifacts in the ML project management solution. This workflow ID establishes connections between experiments, data, and workflows, enhancing project organization and traceability. It can also be used to query and

TABLE II: ML workflow tracking capabilities as implemented using Pegasus-WMS.

Tracking Type	**Dynamic**		**Custom**			
Elements & Tags	Basic Tracking	Full Tracking	Input Tracking	Output Tracking	WF Tracking	Transformation Tracking
Workflow description	×	✓	×	×	✓	×
Workflow instance/version	×	✓	✓	✓	✓	×
Logical file names	✓	✓	✓	✓	✓	×
Physical Input files	✓	✓	✓	×	×	×
Physical Output files	✓	✓	×	✓	×	×
Physical Intermediate files	✓	✓	✓	✓	×	×
Logical Transformations	×	✓	×	×	×	✓
Physical Transformations	×	✓	×	×	×	✓
Codes	×	✓	×	×	×	✓
Declaration location	Workflow Declaration		File Declaration		Workflow Declaration	Transformations Declaration

locate the workflow in the MLflow interface, providing access to all metadata about the execution, the metrics logged by the user, and direct links to the data.



Expirement : workflow name
Parent Run 1 : workflow instance 1
Child Run 1 : Tracked Job 1
Child Run N : Tracked Job N
Parent Run N : workflow instance N

Fig. 3: Pegasus-WMS MLflow structure for ML experimentation

F. Code Versioning

To ensure comprehensive tracking of workflow instances and maximize the reproducibility of ML experiments, tracking the code used in each experiment is a crucial element in the tracking process. For this purpose, and following the same approach as data and workflows, we have introduced a new tag for tracking transformations. This tag, named ***Track_trans***, facilitates automatically pushing the codes in the workflow to the Git server and building metadata descriptions about the scripts. This metadata primarily includes the commit ID, the direct link to the committed file, the local path, and the checksum for verification. The generated metadata is then concatenated with information about the tracked data in the workflow instance and stored in a central metadata file.

G. Dynamic Tracking

For automatic tracking of workflow where the user needs effortless tracking, we introduced two tracking options:

– **full_track:** Like the name indicates, this tag ensures that the WMS tracks all files (input and output) used in each job, as well as the transformations and workflows used. This tag also creates the experiments on MLflow and organizes them as described in Fig. 3. This tag is equivalent to combining all the previous tags, which means ensuring the reproducibility of experimentation if needed.
– **basic_track:** In this tag, reproducibility is of minor importance because it only provides the possibility to track and version the input and output files, meaning only versioning the data.

At this stage, we present our vision for tracking ML workflows. In this approach, the WMS serves as the cornerstone for tracking, facilitating various components such as data tracking, code tracking, and model tracking. This tracking is ensured by the tags or metadata added directly to workflows generated. Table II summarizes the ML workflow tracking capabilities as implemented using the API and the functionality offered by Pegasus-WMS. This approach enables users to easily correlate between these elements. Furthermore, thanks to this tracking approach, Pegasus-WMS can offer the capability to reproduce workflows by retrieving all elements based on the workflow instance ID and the central metadata file. This allows for the retrieval of all necessary components from remote storage and the rerunning of the same workflow with the same data and code as initially tracked.

H. Implementation

After discussing the data tracking approach presented in this work in Section III-D and the ML experimentation management in Section III-E, this section delves into the implementation of an approach aimed at bridging the gap between WMS, exemplified by the Pegasus-WMS, data versioning represented by our approach, and ML experimentation management facilitated by the MLflow open-source solution, with which the WMS will interface.

In this implementation, we extend the Python API of Pegasus-WMS by introducing a new component that incorporates the ML workflow tracking outlined in the previous sections of this work. Fig.4 illustrates the three main phases of the Pegasus-WMS to execute a workflow. **(1) Create:** the user uses the Pegasus API to generate an abstract workflow description, outlining tasks, input/output files, and dependencies, without specific file formats; **(2) Plan:** Pegasus converts the abstract description into a concrete one, adding specific execution details and generating a formatted workflow DAG; **(3) Run:** Pegasus delegates the DAG to HTCondor for job queue management and execution supervision.

In this implementation, as shown in Fig. 4, the Pegasus-data component integrates into the Pegasus API during the creation phase to add tracking jobs based on the user's abstract workflows. When generating the workflow,

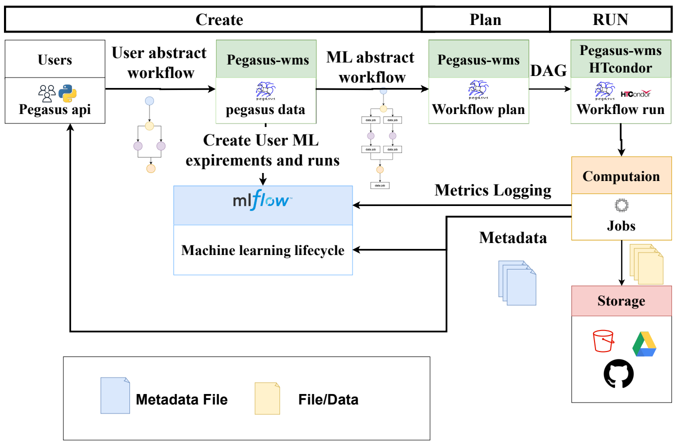


Fig. 4: System design.

it analyzes the tracking and experiment tags defined by the user, leading to the creation of MLflow experiments and runs with unique run IDs. These IDs are passed to the execution environment as environment variables (e.g., *MLFLOW_RUN_ID*, *MLFLOW_EXPERIMENT_NAME*, *MLFLOW_TRACKING_URI*, *MLFLOW_CREDENTIALS*), enabling users to authenticate and log information to MLflow by inserting the metrics tracking code into the transformation using the MLflow Python API.

Simultaneously, the Pegasus-data component adds data tracking jobs with similar input or output data and with the same execution environment parameters as the original tracked job to the abstract workflow. These jobs run concurrently with the main jobs, collecting data information and transmitting it to remote storage (buckets or Google Drive). If MLflow is enabled by the user, these jobs log workflow information to MLflow, including the workflow execution ID, remote storage URL, and metadata files, ensuring the relationship between workflow execution, data versioning, and experiment outputs is maintained.

All connection information and configurations are stored in a central local configuration file and the main Pegasus database, located in the central node of Pegasus-WMS, and passed as environment variables as needed.

IV. Experiments

In this paper, we analyze metadata tracking based on two perspectives of ML workflows:

- **Traditional ML:** We will compare the performance of Pegasus-WMS with and without tracking for three application workflows. In this paper, we utilize the "Lung Segmentation Workflow" and "Galaxy classification" implemented described in the paper [5].
- **Federated Learning (FL)** Representing distributed ML, this implementation allows us to evaluate tracking within a distributed edge computing paradigm, providing insights into its effectiveness across diverse ML methodologies.

In these workflows, dynamic tracking is enabled during generation using the tracking component. Specifically, we utilize the "**full**" tracking type, which dynamically tracks the entire workflow without any other tracking types. This includes versioning the generated workflows, tracking the files/data used on Google Drive, automatically pushing transformations to a branch on GitHub under the workflow's name, and preparing the experiments required in the MLflow platform. Alongside ensuring accurate workflow execution, these experiments provide insights into parameters such as execution time and analysis of generated jobs.

A. Description of Workflows

Lung Segmentation Workflow

Context: Lung segmentation is a crucial component of the broader field of medical image analysis, playing a pivotal role in the diagnosis and treatment of pulmonary diseases. Accurate and precise delineation of lung structures from radiological images, such as X-rays, CT scans, or MRI scans, is paramount for a multitude of clinical applications. In recent years, the integration of deep learning techniques into the realm of medical image analysis has revolutionized the way we approach lung segmentation tasks.

Workflow Overview: The Lung Segmentation Workflow (Fig. 5) uses the *Chest X-ray Masks and Labels* dataset (800 high-resolution X-ray images and masks, 5.4 GB) available on Kaggle. The dataset is split into training, validation, and test sets before the workflow starts. Each set consists of original lung images (3000x2933 pixels each, 6.3 MB in size) and their associated masks (same resolution, 30 KB in size). The *Pre-process and Augment Images* job resizes images (lungs and masks) to 256x256 pixels and normalizes lung X-rays. Additionally, for each pair of lung images and masks in the training dataset, two new pairs are generated through image augmentation (e.g., rotations, flips). Next, the train and validation data are passed to the *UNet HPO* job, where Optuna [18] explores different learning rates. The *Train UNet* job fine-tunes the UNet model with the recommended learning rate on the concatenated train and validation set, and saves the weights into a file. The *Inference on Unet* job uses the trained model to generate masks for the test X-ray images. The final step of the workflow, the *Evaluation* job generates a PDF file with the scores for relevant performance metrics and prints examples of lung segmentation images produced by the model.

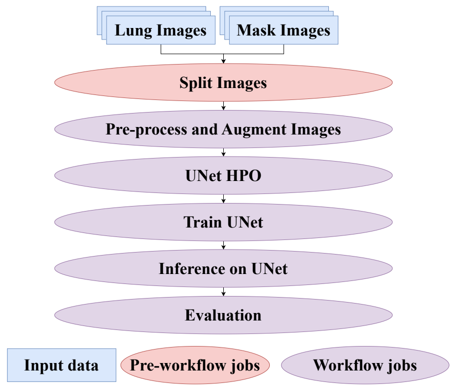


Fig. 5: Lung segmentation workflow.

Galaxy Classification Workflow

Context: Automated galaxy morphology classification is a critical step in understanding the formation and evolution of galaxies. It allows astronomers to systematically categorize and analyze galaxies on an unprecedented scale, enabling the identification of trends and patterns in the galaxy population. The Sloan Digital Sky Survey (SDSS) [19] has gathered over 600 terabytes of image and spectral data over its mission lifetime and it has become an important resource in studying the sky. This staggering volume of data highlights the need for automated classification techniques like deep learning to efficiently process and categorize galaxies.

Workflow Overview: The Galaxy Workflow (Fig. 6) utilizes the Galaxy Zoo 2 dataset[2] that consists of 61,578 RGB images, each of size 424x424x3 pixels (1.9 GB of compressed data). The first stage of the workflow (*Dataset Generation and Split*) filters out galaxies based on their feature scores. This reduced dataset of 28,790 images is split into training, validation, and test sets. These datasets are passed to *Pre-process Images* jobs where several data transformations (e.g., crop, downscale, whitening) are applied. To address the problem of class imbalance in the dataset *Augment Images* jobs generate additional instances of underrepresented galaxy types. Next, *VGG16 HPO* job utilizes the Optuna [18] to find a good set of hyperparameters (e.g., learning rate, numbers of transferred layers). Using the chosen hyperparameters and the training images, the *Train VGG16* job trains the model. The weights of the trained model are saved to a checkpoint file. Finally, the *Inference and Evaluation* job runs predictions on the test set, generates statistics and plots that provide insights into the quality of the trained model. The implementation of this workflow is based on a recent publication [20].

FL Workflow

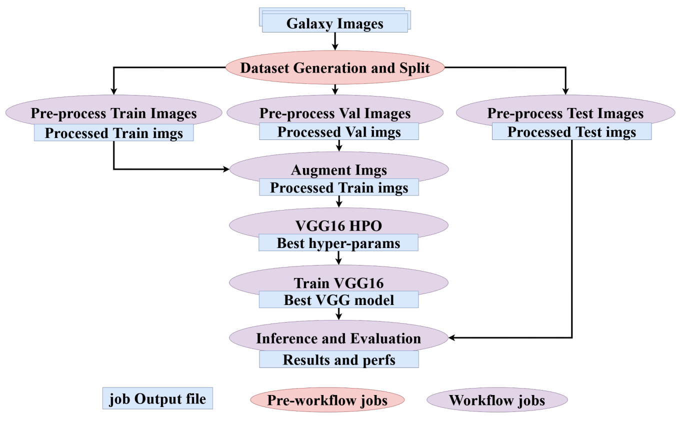


Fig. 6: Galaxy classification workflow.

Context: Federated Learning (FL), introduced by Google in 2017 [21], addresses privacy concerns by enabling local training on individual devices or servers. Each device or server computes model updates using its local data, which are then aggregated centrally to update the global model. This iterative process enhances edge computing capabilities while ensuring data privacy and security.

[2]https://www.kaggle.com/c/galaxy-zoo-the-galaxy-challenge

Workflow Overview: This paper explores an implementation of FL to address the complexity of managing distributed rounds and numerous participants during training or evaluations.Fig. 7 shows a single round of FL, from model initialization to global performance evaluation. Initially, an ***init_model*** job creates the initial model. Three clients are selected for local training using this model and their data, facilitated by the ***local training client**** job. These local models are aggregated into a global model via the ***Global model Aggregation*** job. The global model is then evaluated by randomly selected clients. The ***Model evaluation client**** job produces local evaluation files, which are used by ***Perf_evaluation*** jobs to generate a global evaluation based on averaged metrics.

Two implementations are proposed: a unified workflow for all rounds in one workflow, and dynamic round generation using Pegasus-WMS's sub-workflow feature. For a three-round FL setup, the MNIST dataset is split among 10 clients, with 5 chosen for training. Each client uses a neural network with two hidden layers (100 neurons each, ReLU activation) and a softmax-activated output layer for classification.

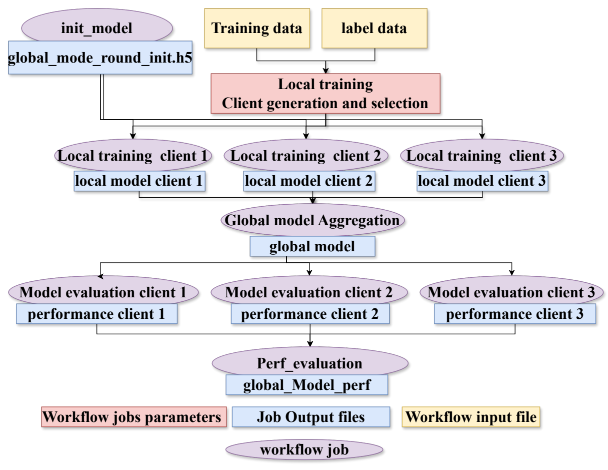


Fig. 7: A workflow for one round of federated learning.

B. *Results analysis*

1) ***Workflow generation****:* Before exploring the workflow execution details, it is essential to evaluate the time taken to generate the workflow structure with and without tracking. Table III represent the execution time for generating tracked and untracked workflows, with and without experimentation tracking using MLflow, as described in the preceding section. The tracking approach modifies the abstract workflow defined by the user by adding new jobs for tracking various files/data utilized in the workflow. for the first workflow, there was no discernible difference between the tracked and untracked workflows. However, upon generating workflows for FL, a noticeable and significant difference in generation time emerged between the two types of workflows. This disparity becomes particularly pertinent when additional jobs are added through the inclusion of new FL rounds. To discern the origin of this difference, we conducted identical experiments without MLflow experimentation tracking. As depicted

in Table III, workflow generation without MLflow experimentation tracking markedly reduces execution time, nearly aligning with untracked workflows. As explained in Section III-E, for each workflow, experimentation with the name of its parent workflow is created, with parent runs identified by timestamps and workflow names containing child runs for individual jobs. In these experiments, managed MLflow (a hosted and maintained service provided by DagHub for managing the machine learning lifecycle) is utilized. Due to the high volume of requests for the MLflow API from the Pegasus-API, the platform queues requests for a brief period of time before continuing the creation process, thereby increasing the workflow generation time and accounting for the significant difference observed in the represented workflows in Table III . Although generating workflows that include tracking jobs takes more time, this additional time is still small compared to the actual workflow execution time, as we will describe in section IV-B3

TABLE III: Workflow generation time in seconds.

Workflow	Not Tracked	Tracked with MLflow	Tracked without MLflow
Lung Segmentation	0.06	0.06	0.06
FL one wf 1 round	0.01	45.26	0.01
FL one wf 3 rounds	0.01	121.39	0.01
Galaxy Classification	2.46	93.43	4.14

2) ***Workflow Jobs analysis:*** In the preceding section (IV-B1), we established that the quantity of jobs significantly influences the generation of the workflow. Here, we conduct an in-depth analysis of the workflow's jobs. Fig. 8 illustrates the total number of jobs per workflow across four implementations. In the case of the lung segmentation and Galaxy Classification, as expected, the total number of jobs for both tracked and untracked models is relatively insignificant compared to the "FL one wf" and "Fl sub wf" cases, where there is a difference in the job count ("FL sub wf" is an FL workflow structured as a nested or hierarchical workflow). This variance likely contributes to the generation time taken by the Pegasus-API to create the workflow. Concerning tracked workflows, Fig. 9 offers a breakdown of total jobs for each tracked workflow. In an untracked workflow, there are two types of jobs: main jobs defined by the user and auxiliary jobs added by the WMS (for example, data staging and registration). In tracked workflows, in addition to these, we have new data-tracking jobs.

Fig. 9 illustrates that the number of tracking jobs increases with the number of main jobs. Notably, the 'lung segmentation' features only 7 main jobs, resulting in the creation of only 8 runs in MLflow. On the other hand, for FL one wf, FL sub wf and Galaxy classification the need to create more than 25 runs at once can lead to blocking during run creation. Moreover, the number of tracking jobs typically doubles the number of user abstract workflow jobs, owing to the addition of data tracking jobs for input and output files, transformations (code) versioning, combining distributed tracking metadata,

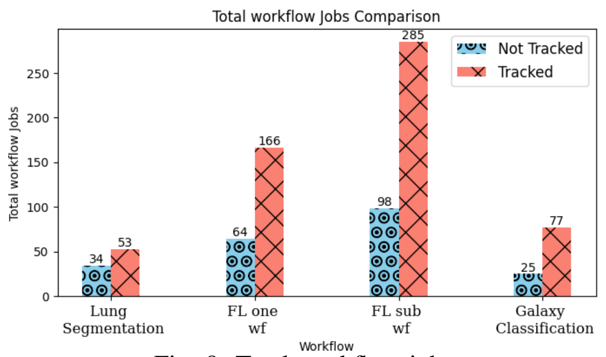


Fig. 8: Total workflow jobs.

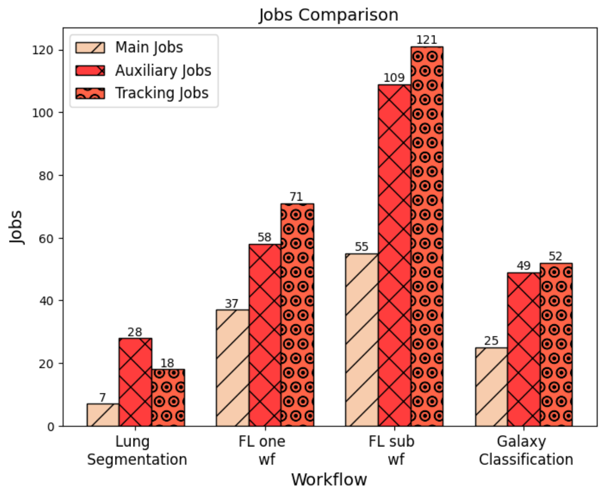


Fig. 9: Tracked workflow jobs.

and copying metadata to the submit node to maintain version consistency. Consequently, the greater the number of jobs, the more data-tracking jobs are generated. Additionally, platforms that block or queue requests can significantly impact workflow generation time. Despite having identical FL configurations as "FL one wf", the "FL sub wf" exhibits more jobs for tracking and main workflows. This disparity arises from the fact that sub-workflows entail additional jobs, particularly for creating workflows and submitting workflows for the subsequent FL round. This increased dynamism enhances post-analysis capabilities, and data jobs and experiment runs are generated only when necessary, contributing to a more efficient workflow structure.

3) ***Workflow Execution:*** After analyzing the generation time and the number of jobs, we explore their impact on the execution time. Table III displays execution times for four workflow implementations, comparing tracked and untracked workflows. For the first two, the execution time remains under 150 seconds (2.5 minutes), but for the third, it exceeds 600 seconds (10 minutes) and for the last one, it exceeds 240 seconds (4 minutes). This discrepancy is due to the sub-workflow structure, where each round necessitates generating workflows anew, incurring MLflow delays described in Section IV-B1. Furthermore, repeated workflow generation and submission increase job execution time, notably for data tracking for transformations, workflow, metadata combining, and metadata

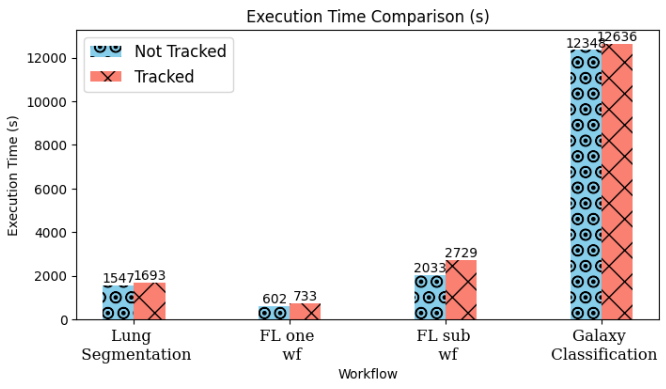


Fig. 10: Execution time by workflow.

copying. In the case of 3 rounds of FL, these jobs execute once for "FL one wf" but at least three times for "FL sub wf", with three workflows generated. Despite higher job volume, the parallel approach described in Section III-B demonstrates that the tracking approach does not significantly impact execution time, with an acceptable gap between tracked and untracked workflows compared to the benefits of tracking and versioning offered by this approach.

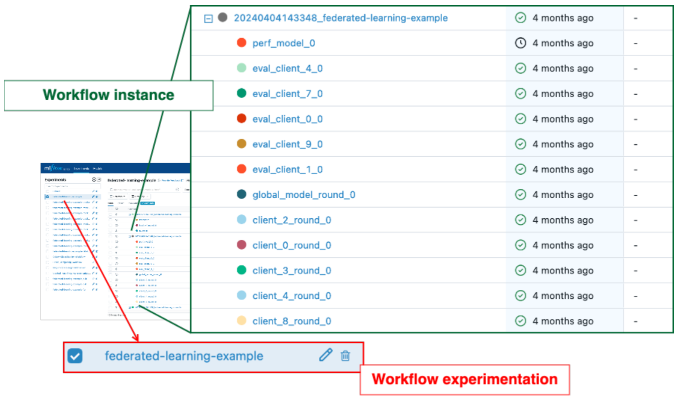
C. Results analysis: application level

After analyzing the performance of the workflows from generation to execution, this section provides an overview of the organization of version files on different platforms, including MLflow, Google Drive, and GitHub.

*1) **MLflow: Experimets Tracking**:* [3] As illustrated in Fig. 11, each workflow corresponds to experimentation containing multiple runs named as date_workflow_name, representing instances of workflows. Each run contains child runs representing the jobs, with the same name as the workflow. By default, each child run includes two elements, as shown in Fig. 12: the first one, at the tag level, includes a direct link to the remote storage (such as Buckets or Google Drive), in addition to the workflow's ID. This information maps the experimentations, workflows, and results. To provide a detailed view of jobs, metadata files describing the used files/data are logged in the MLflow artifact, offering precise insight into inputs and outputs for specific jobs.

*2) **Git: workflow transformations versioning**:* [4]

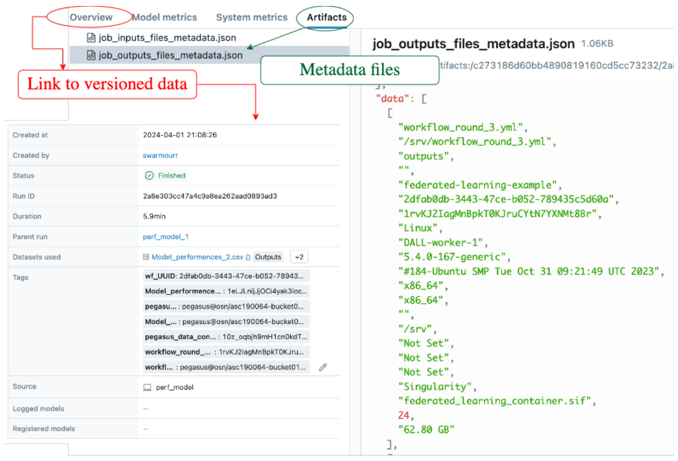
To manage the versioning of code used in workflows and their metadata, the tracking jobs utilize two branches. Firstly, an automatic branch is created by the tracking jobs for each workflow (Fig. 13), containing only the transformations (code) provided by the user. This tracking is added to the metadata of the workflow instance, including the commit ID for reproducibility if needed. If the branch does not exist, it is created automatically. Secondly, the main branch (Fig. 14) is used to version the metadata file of all workflows. This metadata can be pulled whenever necessary to track new workflow instances and copied to the local submit node after



20240404143348_federated-learning-example	4 months ago	-
perf_model_0	4 months ago	-
eval_client_4_0	4 months ago	-
eval_client_7_0	4 months ago	-
eval_client_0_0	4 months ago	-
eval_client_9_0	4 months ago	-
eval_client_1_0	4 months ago	-
global_model_round_0	4 months ago	-
client_2_round_0	4 months ago	-
client_0_round_0	4 months ago	-
client_3_round_0	4 months ago	-
client_4_round_0	4 months ago	-
client_8_round_0	4 months ago	-

federated-learning-example

Fig. 11: MLflow organisation[3].



Created at	2024-04-01 21:08:26
Created by	swarmourr
Status	Finished
Run ID	2a8e303cc47a4c9a8ea262aad0893ad3
Duration	5.9min
Parent run	perf_model_1
Datasets used	Model_performences_2.csv () Outputs +2
Tags	wf_UUID: 2dfab0db-3443-47ce-b052-78943...
	Model_performence... : 1eLJLnijJjOCi4yak3ioc...
	pegasu... : pegasus@osn/asc190064-bucket0...
	Model_... : pegasus@osn/asc190064-bucket0...
	pegasus_data_con... : 10z_oqbjh9mH1cn0kdT...
	workflow_round_... : 1rvKJ2IagMnBpkT0KJru...
	workfl... : pegasus@osn/asc190064-bucket01...
Source	perf_model
Logged models	—
Registered models	—

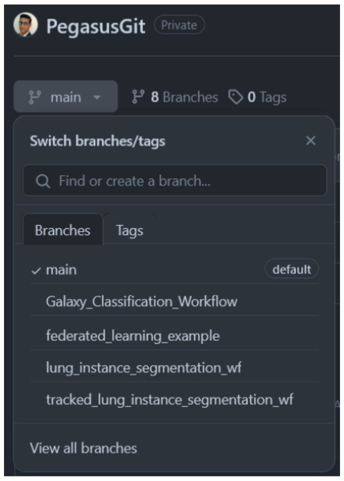
job_outputs_files_metadata.json 1.06KB

tifacts:/c273186d60bb4890819160cd5cc73232/2a

```
],
"data": [
  [
    "workflow_round_3.yml",
    "/srv/workflow_round_3.yml",
    "outputs",
    "",
    "federated-learning-example",
    "2dfab0db-3443-47ce-b052-789435c5d60a",
    "1rvKJ2IagMnBpkT0KJruCYtN7YXNMt8Br",
    "Linux",
    "DALL-worker-1",
    "5.4.0-167-generic",
    "#184-Ubuntu SMP Tue Oct 31 09:21:49 UTC 2023",
    "x86_64",
    "x86_64",
    "",
    "/srv",
    "Not Set",
    "Not Set",
    "Not Set",
    "Singularity",
    "federated_learning_container.sif",
    24,
    "62.80 GB"
  ],
```

Fig. 12: Linking MLflow with data versioning for consistency and reproducibility in ML workflows.

each submission, ensuring synchronization between the Git and local repositories.



main
Galaxy_Classification_Workflow
federated_learning_example
lung_instance_segmentation_wf
tracked_lung_instance_segmentation_wf

Fig. 13: Workflow scripts versioning organization [4].

*3) **Data storage : data versioning**:* [5] To version the data used in each workflow in these experiments, Google Drive is utilized, employing a similar organizational structure as MLflow. Each workflow has its specific folder, as shown in

[3]https://dagshub.com/swarmourr/FL-WF.mlflow/
[4]https://github.com/swarmourr/PegasusGit

[5]Remote Storage Google Drive

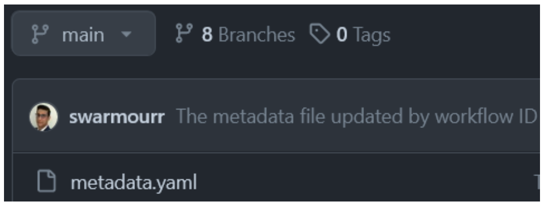
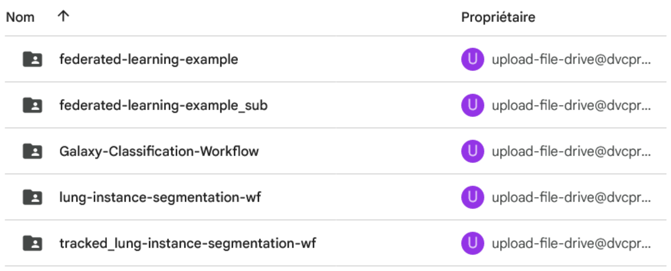

main
8 Branches
0 Tags
swarmourr The metadata file updated by workflow ID
metadata.yaml

Fig. 14: Main GitHub repository tracking branch.

Fig. 15. Within these directories, subdirectories are organized by the ID of the workflows, with additional directories for used files and workflows categorized as input/output/WF. This organization facilitates easy access and management of data associated with each workflow instance. This implementation maintains the link between the data, experiment tracking results, code used, and the workflows employed, ensuring seamless connectivity across all elements. It simplifies the reproduction of data experiments by requiring only the workflow metadata file, which can pull data, code, and workflows from different distant storage locations. This approach guarantees consistency by utilizing the same data, code, and workflows across experiments.



Nom ↑	Propriétaire
federated-learning-example	upload-file-drive@dvcpr...
federated-learning-example_sub	upload-file-drive@dvcpr...
Galaxy-Classification-Workflow	upload-file-drive@dvcpr...
lung-instance-segmentation-wf	upload-file-drive@dvcpr...
tracked_lung-instance-segmentation-wf	upload-file-drive@dvcpr...

Fig. 15: Data organisation in Google Drive[5]

V. Conclusion

In this work, we addressed the issue of tracking and versioning in ML workflows. Despite the use of WMS, reproducing experiments, especially in ML workflows, remains challenging without considering the elements used at runtime, such as code, data, and execution environment, for generating consistent results.

We presented an approach to dynamically track data, code, and workflows. We generate dedicated jobs for tracking and versioning upon request. These jobs run in parallel with the main job to create human-readable metadata. Workflow metadata provides insights into the data, code, and executed workflow during an experiment. Additionally, they are automatically versioned using Git, stored in Google Drive or S3, and linked to experiment results via MLflow, identifiable everywhere with the workflow instance ID.

Our approach demonstrated good results, showing that workflow execution time is unaffected thanks to parallel job execution. The workflow generation time depends on MLflow API access policies, which vary depending on query management policies and concurrency limits. Furthermore, the number of jobs added to the original workflow is approximately twice the number of jobs in the original workflow, plus three jobs for code versioning, metadata combination, and copying to the local submission node for Git-local synchronization.

Future work will focus on refining our tracking approach for improved implementation efficiency and integrating Pegasus-WMS with other tracking solutions. Additionally, we plan to enhance a dedicated command for reproducing tracked workflows. This work represents the beginning of significant efforts in workflow management to streamline and automate tracking, fostering transparency and ensuring reproducibility in WMS.

Acknowledgment

This work is funded by DOE award #DE-SC0022328 and NSF award #2138286 and #2106147.

References

[1] E. Deelman *et al.*, "The evolution of the pegasus workflow management software," *Computing in Science Engineering*, vol. 21, no. 4, pp. 22–36, 2019.
[2] "Apache Airflow Documentation," https://airflow.apache.org/docs/.
[3] D. Kreuzberger *et al.*, "Machine learning operations (mlops): Overview, definition, and architecture," *IEEE Access*, 2023.
[4] "Data Version Control," https://dvc.org.
[5] P. Krawczuk *et al.*, "A performance characterization of scientific machine learning workflows," in *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2021, pp. 58–65.
[6] ——, "Crisisflow: multimodal representation learning workflow for crisis computing," in *2021 IEEE 17th International Conference on eScience (eScience)*. IEEE, 2021, pp. 264–266.
[7] M. H. Nguyen *et al.*, "Integrated machine learning in the kepler scientific workflow system," *Procedia Computer Science*, vol. 80, pp. 2443–2448, 2016.
[8] H. R. Roth *et al.*, "Nvidia flare: Federated learning from simulation to real-world," *arXiv preprint arXiv:2210.13291*, 2022.
[9] J. a. Carreira *et al.*, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 13–24.
[10] M. Zaharia *et al.*, "Accelerating the machine learning lifecycle with mlflow." *IEEE Data Eng. Bull.*, vol. 41, no. 4, pp. 39–45, 2018.
[11] E. Bisong *et al.*, "Kubeflow and kubeflow pipelines," *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*, pp. 671–685, 2019.
[12] A. V. Joshi *et al.*, "Amazon's machine learning toolkit: Sagemaker," *Machine learning and artificial intelligence*, pp. 233–243, 2020.
[13] S. Singh *et al.*, "Analysis and implementation of microsoft azure machine learning studio services with respect to machine learning algorithms," in *Modern Electronics Devices and Communication Systems: Select Proceedings of MEDCOM 2021*. Springer, 2023, pp. 91–106.
[14] M. Vartak *et al.*, "Modeldb: a system for machine learning model management," in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, 2016, pp. 1–3.
[15] S. Schelter *et al.*, "Automatically tracking metadata and provenance of machine learning experiments," 2017.
[16] S. Woodman *et al.*, "Achieving reproducibility by combining provenance with service and workflow versioning," in *Proceedings of the 6th workshop on Workflows in support of large-scale science*, 2011, pp. 127–136.
[17] H. Miao *et al.*, "Provdb: Lifecycle management of collaborative analysis workflows," in *Proceedings of the 2nd Workshop on Human-in-the-Loop Data Analytics*, 2017, pp. 1–6.
[18] T. Akiba *et al.*, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019.
[19] S. M. Kent, "Sloan digital sky survey," *Astrophysics and Space Science*, vol. 217, pp. 27–30, 1994.
[20] X.-P. Zhu *et al.*, "Galaxy morphology classification with deep convolutional neural networks," *Astrophysics and Space Science*, 2019.
[21] B. McMahan *et al.*, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*. PMLR, 2017, pp. 1273–1282.