MLCask: Efficient Management of Component Evolution in Collaborative Data Analytics Pipelines

Zhaojing Luo*, Sai Ho Yeung*, Meihui Zhang†, Kaiping Zheng*, Gang Chen†, Feiyi Fan§, Qian Lin*, Kee Yuan Ngiam¶, Beng Chin Ooi∗

Abstract—With the ever-increasing adoption of machine learning for data analytics, maintaining a machine learning pipeline is becoming more complex as both the datasets and trained models evolve with time. In a collaborative environment, the changes and updates due to pipeline evolution often cause cumbersome coordination and maintenance work, raising the costs and making it hard to use. Existing solutions, unfortunately, do not address the version evolution problem, especially in a collaborative environment where non-linear version control semantics are necessary to isolate operations made by different user roles. The lack of version control semantics also incurs unnecessary storage consumption and lowers efficiency due to data duplication and repeated data pre-processing, which are avoidable.

In this paper, we identify two main challenges that arise during the deployment of machine learning pipelines, and address them with the design of versioning for an end-to-end analytics system MLCask. The system supports multiple user roles with the ability to perform Git-like branching and merging operations in the context of the machine learning pipelines. We define and accelerate the metric-driven merge operation by pruning the pipeline search tree using reusable history records and pipeline compatibility information. Further, we design and implement the prioritized pipeline search, which gives preference to the pipelines that probably yield better performance. The effectiveness of MLCask is evaluated through an extensive study over several real-world deployment cases. The performance evaluation shows that the proposed merge operation is up to 7.8x faster and saves up to 11.9x storage space than the baseline method that does not utilize history records.

Index Terms—Machine Learning Pipelines, Version Control Semantics, Scientific Data Management, Data Analytics

I. INTRODUCTION

In many real-world machine learning (ML) applications, new data is continuously fed to the ML pipeline. Consequently, iterative updates and retraining of the analytics components become essential, especially for applications that exhibit significant concept drift behavior [12] where the trained model becomes inaccurate as time passes. Consider healthcare applications [2], [18], [23], [42] as an example in which hospital data is fed to data analytics pipelines [14], [22] on a daily basis for various medical diagnosis predictions. The extracted data schema, pre-processing steps, analytics models are highly volatile [7], [20], [43] due to the evolution of the dataset, leading to a series of challenges. First, to ensure quality satisfaction of the analytics models, the pipeline needs to be retrained frequently to adapt to the changes, which costs a lot of storage and time [3], [31], [32], [38]. Second, the lengthy pipeline and computer cluster environment cause the asynchronous pipeline update problem, because different components may be developed and maintained by different users. Third, the demand for retrospective research on models and data from different time periods further complicates the management of massive pipeline versions.

To address the aforementioned challenges, version control semantics [16], [24], [31], [36] need to be introduced to the ML pipeline. Current pipeline management systems either do not explicitly consider the version evolution, or handle versioning by merely archiving different versions into distinctive disk folders so that different versions will not conflict with or overwrite each other. The latter approach not only incurs huge storage and computation overhead, but also fails to describe the logical relationship between different versions.

In this paper, we first elaborate on the common challenges in data analytics applications and formulate version control semantics in the context of ML pipeline management. We then present a design of Git-like end-to-end ML life-cycle management system, called MLCask, and its version control support. MLCask facilitates collaborative component updates in ML pipelines, where components refer to the computational units in the pipeline such as data ingestion methods, pre-processing methods, and models. The key idea of MLCask is to keep track of the evolution of pipeline components together with the inputs, execution context, outputs, and the corresponding performance statistics. By introducing the non-linear version control semantics [16], [24], [36] to the context of ML pipelines, MLCask can achieve full historical information traceability with the support of branching and merging. Further, we propose two methods in MLCask to prune the pipeline search tree and reuse materialized intermediate results to reduce the time needed for the metric-driven merge operation. Lastly, to minimize the cost of the merge operation for divergent ML pipeline versions, we devise multiple strategies...
II. CHALLENGES OF SUPPORTING DATA ANALYTICS APPLICATIONS

In many real-world data analytics applications, not only data volume keeps increasing, but also analytics components undergo frequent updates. A platform that supports intricate activities of data analytics has to address the following two key challenges.

(C1) Frequent retraining. Many real-world data analytics applications require frequent retraining since concept drift is a common phenomenon [12]. For instance, in the computer cluster of NUHS hospital, there are around 800 to 1200 inpatients at any given time and the number of newly admitted patients each day is around 150. Given this dynamic environment, retraining models by using new patient data from time to time is essential for delivering accurate predictions. Currently, the existing workflow needs to rerun every component for each retraining, which is time consuming and resource intensive. Meanwhile, different pipeline versions are archived into separate folders, which leads to huge storage consumption. To overcome the aforementioned resource problems, a mechanism is needed to identify the component that does not need to be rerun for efficient pipeline management. Furthermore, a component’s output maybe just partially different from the output of its previous version; hence, archiving them into separate folders does not resolve the storage redundancy.

(C2) Asynchronous pipeline component update and merge. As expected for collaborative analytics, concurrent updates of a pipeline introduce both consistency and maintenance issues. First, the asynchronous component update by different users may cause potential failure of the entire pipeline when two incompatible updated components are combined. Second, we should consider the fundamental difference between software engineering and building ML pipelines: ML pipeline development is metric-driven, rather than feature-driven. For building ML pipelines, data scientists typically pursue pipeline performance, and different branches are used for iterative trials. They often create different branches for iterative trials to improve individual components of the pipeline. In contrast, software engineers merge two branches because the features developed on the merging branches are needed.

In the context of ML pipeline, simply merging two branches with the latest components does not necessarily produce a pipeline with improved performance, because the performance of the whole pipeline depends on the interaction of different components. Therefore, developing ML pipelines through the collaboration of independent teams that consist of dataset providers (data owners), model developers, and pipeline users is challenging but necessary for better exploitation of individual knowledge and effort. Consequently, we have to address the issue of merging the pipeline updates from different user roles and searching for the best component combination.

Fig. 1. Difficulties in combining updates from different users.
Combining the latest updates from different users is not an ideal solution as it may lead to a sub-optimal pipeline or the asynchronous pipeline update problem \[^{[2]}\] as shown in Fig. 1. Consider the scenario where a user Jane commits a new version of the ML model (indicated by purple) while another user Frank commits a new version of the data pre-processing method (indicated by blue) separately in two branches, i.e., “Jane-dev” and “Frank-dev”. When the other components are fixed, both the new ML model and the new data pre-processing method are better than their old counterparts when they are evaluated separately. However, the performance of the new pipeline that incorporated both updates is generally unknown until it is actually evaluated. Note that the solution space is dependent on the pipeline search space which is usually huge, and an unknown solution space could have multiple local optima. Searching for the optimal combination of updates in a huge search space across the version history is not practical, and thus efficient pruning and optimization are required.

In order to address the aforementioned challenges, version control semantics are incorporated into our end-to-end system MLCask as follows. By leveraging the version history of pipeline components and workspace, skipping unchanged processing steps is realized in Section IV to address (C1), and non-linear version control semantics and merge operation are realized in Sections V and VI to address (C2).

III. SYSTEM ARCHITECTURE OF MLCASK

In this section, we introduce the system architecture of the ML life-cycle management system MLCask, which facilitates collaborative development and maintenance of ML pipelines. MLCask provides version control, stores evaluation results as well as provenance information, and records the dependency of different components of the pipelines. The architecture of MLCask is illustrated in Fig. 2.

![Diagram of MLCask architecture](image)

**Fig. 2.** The architecture of MLCask for supporting collaborative pipeline development with version control semantics.

In general, we abstract an ML life-cycle with two key concepts: component and pipeline.

**Component:** A component refers to any computational units in the ML pipeline, including datasets, pre-processing methods and ML models. In particular, we refer library to either a pre-processing method or an ML model.

A dataset is an encapsulation of data which could either be a set of data files residing in a local/server side, or defined by the combination of database connection configurations and the associated data retrieval queries. A dataset contains a mandatory metafile that describes the encapsulation of data and a series of optional data files.

A library consists of a mandatory metafile and several executables. It can be an executable that performs data pre-processing tasks or deep analytics. In our implementation, we employ Apache SINGA \[^{[27]}, [37]\] , a distributed deep learning system as the backend for training deep learning models. Besides Apache SINGA, MLCask can also readily works with other backend systems such as TensorFlow\[^{[2]}\] or PyTorch\[^{[3]}\] as long as the interface is compatible with the ML pipeline.

**Pipeline:** A pipeline is the minimal unit that represents an ML task. When a pipeline is created with the associated components, the references to the components are recorded in the pipeline metafile. Once a pipeline is fully processed, all its component outputs are archived for potential future reuse, with their references logged into the pipeline metafile.

Considering that a single dataset or library may be used by multiple pipelines, we design a dataset repository and a library repository to store different versions of datasets and libraries respectively, which are shared by all the pipelines in order to reduce storage costs. A pipeline repository is also introduced to record the version updates of all the pipelines.

**Running Example:** To appreciate the discussion in the rest of the paper, without loss of generality, we exemplify an ML pipeline, as shown in Fig. 2 which consists of datasets, data cleansing, feature extraction and a convolutional neural network (CNN) model. This ML pipeline is used to predict whether a patient will be readmitted into the hospital within 30 days after discharge.

IV. VERSION CONTROL SEMANTICS

A. Preliminaries

We use Directed Acyclic Graph (DAG) to formulate an ML pipeline as follows:

**Definition 1 (ML Pipeline).** An ML pipeline \( p \) with components \( f_i \in \mathcal{F} \) is defined by a DAG \( G = (\mathcal{F}, \mathcal{E}) \), where each vertex represents a distinct component of \( p \) and each edge in \( \mathcal{E} \) depicts the successive relationship (i.e., direction of data flow) between its connecting components.

**Definition 2 (Pipeline Data Flow).** For a component \( f \in \mathcal{F} \), let \( \text{succ}(f) \) and \( \text{pre}(f) \) be the set of succeeding and preceding components of \( f \) respectively. Correspondingly, given components \( f_i, f_j \in \mathcal{F} \) and a data flow \( e_{ij} \in \mathcal{E} \) from \( f_i \) to \( f_j \), we have \( f_j \in \text{succ}(f_i) \) and \( f_i \in \text{pre}(f_j) \).

**Definition 3 (Pipeline Component).** A pipeline component \( f_i \) with the type of library can be viewed as a transformation: \( y = \)

1) https://www.tensorflow.org
2) https://pytorch.org/


\[ f_i(x | \theta_i), \] where \( x \) is the input data of \( f_i \), \( \theta_i \) is the component’s parameters, and \( y \) denotes \( f_i \)’s output.

**Definition 4 (Component Compatibility).** A pipeline component \( f_j \) is compatible with its preceding component \( f_i \in \text{pre}(f_j) \) if \( f_j \) can process the output by component \( f_i \) correctly.

**B. Version Control for Pipeline Components**

A **semantic version**\(^4\) in MLCask is represented by an identifier: `branch@schema.increment`, where `branch` represents the Git-like branch semantics, `schema` denotes the output data schema, and `increment` represents the minor incremental changes that do not affect the output data schema.

We use the notation: `<feature_extract, master@0.1>` to denote a component named `feature_extract` and its corresponding semantic version. The representation indicates that the component has received one incremental update and there is no output data schema update yet. For components on its `master` branch, we simplify the representation to the following form: `<feature_extract, 0.1>`. The initial version of a committed library is set to 0.0. Subsequent commits only affect the `increment` domain if `schema` is not changed. In this paper, we assume that the output data schema is the only factor that determines the compatibility between \( f_i \) and \( f_j \). Specifically, if the output data schema of \( \text{pre}(f_i) \) changes, \( f_i \) should perform at least one `increment` update to ensure its compatibility with \( \text{pre}(f_i) \).

For a library component, the update to `schema` is explicitly indicated by the library developer in the library metafile.\(^5\) For a dataset component, we propose that the data provider uses the schema hash function to map the `schema` from data. For data in relational tables, all the column headers are extracted, standardized, sorted, and then concatenated into a single flat vector. Consequently, a unique `schema` can be generated by applying a hash function such as SHA256 on the vector obtained. Note that there are many methods available in the literature on the hash function optimization and this is not the focus of MLCask.

Managing linear version history in ML pipeline has been well studied in literature [31], [32]. However, existing approaches cannot fulfill the gap when non-linear versioning arises, which is common in ML pipelines where multiple disciplines are involved. To tackle this problem, we develop the MLCask system to support non-linear version management in collaborative ML pipelines.

**V. SUPPORTING NON-LINEAR VERSION CONTROL**

We use the pipeline shown in Fig. 3 to illustrate how MLCask achieves `branch` and `merge` operations to support non-linear version history. The example pipeline fetches data from a hospital dataset, followed by data cleansing and feature extraction, and eventually feeds the extracted data into a CNN model to predict how likely a specific patient will be readmitted in 30 days.

**Branch:** In the collaborative environment, committing on the same branch brings in complications in the version history. It is thus desirable to isolate the updates made by different user roles or different purposes. To address this issue, MLCask is designed to support `branch` operations on every pipeline version. As shown in Fig. 3 the `master` branch remains unchanged before the `merge` if all updates are committed to the `dev` branch. By doing so, the isolation of a stable pipeline and development pipeline can be achieved.

**Merge:** The essence of merging a branch to a base branch is to merge the commits (changes) happened on the merging branch to the base branch. By convention, we term the base branch as `HEAD` and the merging branch as `MERGE_HEAD`.

For the simplest case shown in Fig. 3, the `HEAD` does not contain any commits after the common ancestor of `HEAD` and `MERGE_HEAD`, which is constrained by the fast-forward merge. For the fast-forward merge, MLCask duplicates the latest version in `MERGE_HEAD`, changes its branch to `HEAD`, creates a new commit on `HEAD`, and finally sets its parents to both `MERGE_HEAD` and `HEAD`. However, if any commits happen on the `HEAD` after the common ancestor, the resulting conflicts may become an issue. An example is illustrated in Fig. 4 in which the component `CNN` is changed on `HEAD` before the merge.

In terms of the merge operation in this scenario, the naïve strategy is to select the latest components to form the merging result. However, it does not guarantee optimal performance due to the complex coupling among pipeline components. In the worst case, `<CNN, 0.4>` in Fig. 4 is not compatible with `<feature_extract, 1.0>`. As can be seen, the asynchronous pipeline updates may cause the compatibility issue in the context of the non-linear version control semantics.

These observations motivate us to redefine the merge operation for ML pipeline. Our assumption is that in MLCask, different users collaboratively update the pipeline in order to improve the performance, which is measured by a specific metric. To be specific, we propose the **metric-driven merge operation**, which aims to select an ML pipeline with optimal performance based on past commits made on `HEAD` and `MERGE_HEAD` referring to their common ancestor. To this end,
we first define the search space for selecting the optimal ML pipeline and then represent the search space using a pipeline search tree. Since combining the latest versions on different branches cannot guarantee an optimal result, it is intuitive to trace back the history, get all available versions and then try to combine them with other components. For example, in Fig. 4 the component CNN has experienced 5 versions of updates based on its common ancestor, and as a consequence, all these 5 versions will be evaluated by the process of pipeline merge. Here we formalize the definition of “all available versions” with respect to the concept of component search space. Given $f_i$ is a component of pipeline $p$, the search space of $f_i$ on $p$’s branch $b$ is defined by:

$$S_b(f_i) = \{v(f_i|p) | p \in P_b\},$$

where $v(f_i|p)$ is the version of $f_i$ in pipeline $p$, $P_b$ is the set of pipeline versions on the branch $b$. When merging two branches, component search space of $f_i$ can be derived by:

$$S(f_i) = S_{\text{MERGE HEAD}}(f_i) \cup S_{\text{HEAD}}(f_i).$$

For data cleansing component in Fig. 4 its component search space contains two versions, namely:

$$<\text{data\_cleanse, 0.0}> \text{ and } <\text{data\_cleanse, 0.1}>$$

To facilitate the search for the optimal combination of pipeline component updates, we propose to build a pipeline search tree using Algorithm 1 to represent all possible pipelines. In Algorithm 1, $S(f_i)$ denotes the component search space of $f_i$, $N_f$ is the number of pipeline components, and $tree$ is the returned pipeline search tree.

![Fig. 4. MLCask pipeline branching and merging with conflicts.](image)

**Algorithm 1: Pipeline search tree construction.**

```plaintext
1. Input: $S(f_i)$, $N_f$
2. Output: tree
3. tree = TreeNode(component = virtual_root, executed = True);
4. for $i$ ← 0 to $N_f$ do
5.   $fSet$ = $S(f_i)$;
6.   parentNodes = tree.getNodeAtLevel(i);
7.   foreach node ∈ parentNodes do
8.     foreach $f$ ∈ $fSet$ do
9.       node.children.add(TreeNode(component = $f$, executed = False))
10. end
11. end
```

Fig. 5 illustrates an example of a pipeline search tree. Every TreeNode records the reference to a set of child nodes, its corresponding pipeline component, an execution status flag, and the reference to the component’s output. All possible pipelines can be obtained by enumerating all paths from the root to the leaves. The set of all the enumerated pipelines is termed as pre-merge pipeline candidates, and is denoted as $P_{\text{candidate}}$. The merged result can be defined by:

$$p_{\text{merged}} = \arg \max_p \{\text{score}(p) | p \in P_{\text{candidate}}\},$$

where $\text{score}(p)$ denotes the metric score that measures the performance of a pipeline. The form of the score function is dependent on the example pipeline search tree built on version history.

For real-world applications, there are different metrics to evaluate an ML pipeline, e.g., precision, recall, F1-score, AUC score and MSE for model performance, execution time, storage consumption, etc. MLCask generates different optimal pipeline solutions for different metrics so that users could select the most suitable one based on their preference. Alternatively, users may combine multiple metrics into a single one, e.g., precision and recall can be combined as F1-score that is used as a single metric.

**VI. OPTIMIZING MERGE OPERATIONS**

In this section, we present optimizations to improve the efficiency of the merge operations in MLCask. The non-triviality of the merge operation lies in the huge search space...
for the optimal pipeline and how to exclude the incompatible pipelines. For a pipeline with \(N_f\) components, the upper bound of the number of the possible pipeline candidates is given by \(\prod_{j=1}^{N_f} N(S(f_i))\), where \(N(S(f_i))\) denotes the number of elements in set \(S(f_i)\). Therefore, the number of pipeline candidates increases dramatically when the number of past commits increases, which may render the merge operation extremely time-consuming.

Fortunately, among a large number of pipeline candidates, those with incompatible components can be safely excluded. Further, if a component of the pipeline candidate was executed before, it does not need to be executed again since its output has already been saved and thus can be reused. Motivated by these two observations, we propose two tree pruning methods to accelerate the merge operation in MLCask.

A. Pruning Merge Tree using Component Compatibility Information

When the schema of a pipeline component changes, its succeeding components have to be updated accordingly. By leveraging the constraints on component compatibility, we can avoid enumerating the pipelines that are destined to fail in execution.

We continue to use the version history as illustrated in Fig. 4 and its corresponding pipeline search tree in Fig. 5 to exemplify the idea. The succeeding components of feature extraction can be divided into two sets based on compatibility:

- \(\{<\text{CNN}, 0.0>, <\text{CNN}, 0.1>, <\text{CNN}, 0.4>\}\) following \(<\text{feature_extract}, 0.0>\);
- \(\{<\text{CNN}, 0.2>, <\text{CNN}, 0.3>\}\) following \(<\text{feature_extract}, 1.0>\);

In Fig. 5, the nodes in red are not compatible with their parent nodes. By pruning all those nodes, the size of the pre-merge pipeline candidate set can be reduced to half of its original size.

In practice, a compatibility look-up table (LUT) is evaluated based on the pipelines’ version history to support the pruning procedure. Firstly, given a component, all its versions on the HEAD and MERGE_HEAD are enumerated. Secondly, for every version of the given component, find its compatible succeeding component versions. Finally, make the compatible component pairs in 2-tuple and fill the LUT with 2-tuple. Therefore, the number of compatible versions of \(suc(f_j)\) and \(N_{j+1} \in \{1, 2, ..., N(S(f_j)) - 1\}\), because there is at least one compatible version for any pipeline component as MLCask enforces the compatibility constraint when committing a new pipeline. Therefore, the lower bound of the remaining number of pipeline candidates after pruning is: \(\prod_{i=1}^{N_f} N(S(f_i)) \prod_{j=0}^{N_{j+2}} N(S(f_j))\), and the upper bound is given by \(\prod_{i=1}^{N_f} N(S(f_i)) - \prod_{i=1}^{N_{j+2}} N(S(f_j))\).

In terms of using the reusable output, in the worst case (historical pipelines on HEAD and MERGE_HEAD have not been executed), it can be derived that the number of nodes to be executed without leveraging the reusable output is \(\prod_{i=1}^{N_f} N(S(f_i)) \times N_f\). As long as pruning by reusable output
is enforced, every node on the pipeline search tree is strictly executed only once. The level \( j \) of pipeline search tree has \( \prod_{i=1}^{j} N(S(f_i)) \) nodes, thus the number of nodes to be executed is \( \sum_{j=1}^{N} \prod_{i=1}^{j} N(S(f_i)) \).

Algorithm 2: Traversal and execution of the nodes on pipeline search tree with pruning heuristics.

```
1 Input: table, rootNode
2 Output: rootNode
3 Function ExecuteTree (rootNode)
4   if node.children ≠ ∅ then
5     foreach child ∈ node.children do
6       if (node.component, child.component) ∉ table then
7         node.children.remove(child)
8         else
9           walkingPath.push(child)
10          ExecuteTree (child)
11         end
12       end
13     else
14       executeNodeList(walkingPath)
15       foreach node ∈ walkingPath do
16         node.executed ← True
17         node.output ← walking_path.getOutput(component)
18       end
19   end
20 end
```

VII. EVALUATION

A. Evaluated Pipelines

In this section, we evaluate the performance of MLCask in terms of storage consumption and computational time using four real-world ML pipelines, namely, patient readmission prediction (Readmission), Disease Progression Modeling (DPM), Sentiment Analysis (SA), and image classification (Autolearn). These pipelines cover a variety of application domains such as healthcare analytics, natural language processing, and computer vision.

Readmission Pipeline: The Readmission pipeline illustrated in Fig. 3 is built to predict the risk of hospital readmission within 30 days of discharge. It involves three major steps: 1) clean the dataset by filling the missing diagnosis codes; 2) extract readmission samples and their medical features, e.g., diagnoses, procedures, etc; 3) train a deep learning (DL) model to predict the risk of readmission.

DPM Pipeline: The DPM pipeline is constructed to predict the disease progression trajectories of patients diagnosed with chronic kidney disease using the patients’ one-year historical data, including diagnoses and lab test results. It involves four major steps where the first two steps are cleaning the dataset and extracting relevant medical features. In the third step, a Hidden Markov Modeling (HMM) model is designed to process the extracted medical features so that they become unbiased. In the last step, a DL model is built to predict the disease progression trajectory.

SA Pipeline: The SA pipeline performs sentiment analysis on movie reviews. In this pipeline, the first three steps are designed to process the external corpora and pre-trained word embeddings. In the last step, a DL model is trained for the sentiment analysis task.

Autolearn Pipeline: The Autolearn pipeline is built for image classification of digits using Zernike moments as features. In the first three pre-processing steps of this pipeline, Autolearn [17] algorithm is employed to generate and select features automatically. In the last step, an AdaBoost classifier is built for the image classification task.

The pre-processing methods of DPM, SA, and Autolearn pipelines are representative data pre-processing methods and are typically costly to run. Consequently, these three pipelines incur a substantial fraction of the overall run time on the pre-processing steps. For the Readmission pipeline, a substantial fraction of the overall run time is spent on the model training. We include these two types of pipelines to show the effectiveness of MLCask in cases where pre-processing or model training dominates run time respectively.

B. Performance Metrics and Baselines

For each pipeline, we evaluate the system performance under two different scenarios: linear versioning and non-linear versioning. In the evaluation of linear versioning performance, we perform a series of pipeline component updates and pipeline retraining operations to collect the statistics on storage as well as run time, and demonstrate the performance gain due to the proposed version control semantics. For every iteration, we update the pre-processing component at a probability of 0.4 and update the model component at a probability of 0.6. When evaluating the non-linear versioning performance, we first generate two branches, then update components on both branches and merge the two updated branches with the proposed version control semantics.

Baseline for Linear Versioning: To verify the effectiveness of the version control semantics on linear versioning, we use the MLCask without version control semantic support (MLCask w/o VCS) as our baseline. The baseline system has been operational in the NUHS hospital computer cluster since 2017 to support its healthcare analytics. The storage mechanism of the baseline system simulates the system proposed in [31], [82] by archiving different versions into separate folders.

Baselines for Non-linear Versioning: We also design two baselines for the non-linear versioning scenario. MLCask without PCPR baseline enumerates all the possible pipeline combinations, where PC refers to “Pruning using component Compatibility”, PR refers to “Pruning using Reusable output”. MLCask without PR prunes all pipelines with incompatible components and enumerates all remaining pipeline combinations. MLCask generates a pipeline tree and prunes all pipelines with incompatible components, as well as the trained pipeline components. Since baselines are designed by removing features from MLCask, the comparison with two baseline systems MLCask without PCPR and MLCask without PR can serve as an ablation study as well.

The evaluation metrics we employ to measure the performance are cumulative execution time (CET), cumulative storage size (CSS). Execution time refers to the time consumption of running the computational components
in the pipeline while the storage time refers to the time needed for data preparation and transfer. Storage size refers to the total data storage used for training and storing the pipeline components and reusable outputs. Pipeline time refers to the sum of execution time and storage time. The execution time, storage time, storage size, and pipeline time are all accumulated every run during the merge operations for measuring non-linear versioning performance.

All the pipelines run on a server equipped with Intel Core-i7 6700k CPU, Nvidia GeForce GTX 980ti GPU, 16GB RAM and 500GB SSD. MLCask and part of the pipeline components were implemented using Python version 3.6.8. Components written in C++ were complied with GCC version 5.4.0.

C. Performance of Linear Versioning

Fig. 6 shows the total time of linear versioning on all four pipelines. From the figure, we can observe that the total time of the baseline system increases linearly but at a faster rate than MLCask in most cases. The linearity originates from the fact that the baseline system has to start all over in every iteration due to the lack of historical information on reusable outputs. Our proposed MLCask consumes less pipeline time because it skips the executed pipeline components.

Fig. 7 shows the pipeline time composition. As can be seen from the figure, the time spent on model training is comparable for all systems, while the main performance difference lies in the pre-processing. For example, for the proposed MLCask, in the DPM pipeline, iteration 3 and iteration 8 take a longer time. It can be observed from the DPM pipeline of Fig. 6 that the graph segment just before iteration 3 and 8 exhibits steeper slope. This is because, in such cases, the updates happen on or before HMM processing, and HMM processing is time consuming, leading to a large amount of prep-processing time. This can be confirmed by Fig. 7(b) that the pre-processing time at iteration 3 and 8 of MLCask is significantly longer than that of other iterations. Similarly, for iteration 9 of SA and iterations 5 and 9 of Autolearn, the graph segments with steeper slope are attributed to the pre-processing methods, i.e., word embedding and feature generation, respectively. For the Readmission pipeline, since the pre-processing is relatively faster, no obvious steep slope is observed from Fig. 6.

We note that, as shown in Fig. 7(b), the proposed system and the baseline system take nearly the same time to materialize the reusable outputs when executing DPM pipeline. In Fig. 7(d) regarding the Autolearn pipeline, materializing the reusable outputs takes a longer time than materializing the original dataset due to the massive newly generated features in the feature generation procedure. However, since skipping equivalent components saves far more time than starting all over (the cumulative pipeline time of MLCask is much lower), materializing reusable outputs is still an efficient solution. As shown in all the four subfigures of Fig. 7, storage time is an insignificant portion of the pipeline time. For example, for Autolearn pipeline in Fig. 7(d), although materializing the
reusable outputs costs about 55 seconds more when compared to the baseline system, the result in the Autolearn pipeline of Fig. 6 shows that using reusable outputs saves more than 8000 seconds after 10 iterations. The observation suggests that in some extreme cases when the reusable outputs are much larger and pre-processing components are much faster, reusable outputs may take a longer time to materialize and the efficiency of skipping the repeated components may lead to worse performance instead.

In terms of storage space shown in Fig. 8, we observe that the consumption of storage of baseline system increase linearly because the outputs of each iteration are archived to different disk folders. For MLCask, since the repeated components and their outputs will be stored only once, the slope gets higher when an update occurs in the earlier stage of the pipeline. Nevertheless, even when the updates occur in the earlier stages, the slope of MLCask is still relatively lower because it is possible for the different versions of pipeline components and reusable outputs to have overlapping contents. We use our own storage engine, ForkBase [36], to provide efficient chunk level de-duplication, and the storage cost is consequently reduced.

D. Performance of Non-linear Versioning

In this section, we present the experiments on non-linear versioning, i.e., merge operation, to evaluate the performance of the proposed system in terms of cumulative pipeline time, cumulative storage cost, cumulative execution time, and cumulative storage time.

The experimental results in Fig. 9 confirm the effectiveness in pruning the pipeline search tree using component compatibility and reusable outputs. The proposed system dominates the comparison in all test cases as well as all metrics, and MLCask without PR provides minor advantages over MLCask without PCPR.

In order to further analyze the difference among these three systems in terms of cumulative pipeline time, we show the pipeline time composition during merge operation in Fig. 10. As can be seen from the figure, the difference in pipeline time among the three systems are mainly attributed to pre-processing. The reason is that both Pruning using component Compatibility and Pruning using Reusable output happen in the pre-processing components. In terms of model training time, it is nearly the same across the systems. Storage time only constitutes a small fraction of the pipeline time, which is consistent with the observation in Section VII-C.

Comparing MLCask without PR with MLCask without PCPR, MLCask without PR enumerates the possible pipelines and removes the incompatible ones explicitly before the pipeline execution, while MLCask without PCPR materializes the dataset and runs pipeline components from scratch until the compatibility error occurs. Since the schema change happens at a lower probability, only a small subset of the pipeline candidates are removed by pruning using component compatibility. Consequently, the advantage of MLCask without PR over MLCask without PCPR is minor.

Comparing MLCask without PR with MLCask, the problem of MLCask without PR is that it cannot leverage the reusable outputs. Fig. 9 shows that this difference leads to the great advantage of MLCask over MLCask without PR. This is due to the fact that MLCask guarantees that each node on the pipeline search tree is executed only once, while for MLCask without PR, in case there are \(M\) pipeline candidates, the first component in the pipeline will be executed for \(M\) times. Therefore, the cumulative execution time and cumulative pipeline time of MLCask decrease dramatically.

In terms of cumulative storage size and time, Fig. 9(b) and 9(d) show that MLCask outperforms the two baselines significantly. The reasons can be explained as follows. Since every node on the pipeline search tree is equivalent for its
child nodes, siblings of the child nodes can reuse the outputs of their parents. Moreover, these outputs can be stored locally as the child nodes can access the output of their parent node. As a result, MLCask materializes the data, typically the root node’s output, and saves the final optimal pipeline (i.e., the result of merge operation) only once. Consequently, MLCask achieves a huge performance boost on the cumulative storage time and cumulative storage size as well.

E. Prioritized Pipeline Search

Although pruning the pipeline search tree narrows pipeline search space, the number of pipelines that need to be evaluated may still be large. Therefore, we prioritize the pipelines which are more promising to have desirable performance based on the pipeline history. By doing so, the merge operation can return better results given a fixed time budget.

Every time a pipeline candidate is run, the corresponding leaf node on the pipeline search tree is associated with its score. We associate the other nodes on the pipeline search tree with scores as well, following the rule that the score of the parent node is computed using the average of its children (except for the children that have not gotten a score yet). The initial scores are assigned using scores of the trained pipelines on MERGE_HEAD and HEAD.

Assume there are $N$ pipeline candidates (paths from the root node to the leave nodes) in the pipeline search tree. To perform a prioritized pipeline search, we start from the root node and sequentially pick the child nodes that have the highest scores until we reach a leaf node that has not been run yet. This process is repeated for $N$ times so that all the $N$ pipeline candidates are searched in order. Random search, on the contrary, searches all the $N$ pipeline candidates in a random order. For both search methods, we denote the process of searching for all the $N$ pipeline candidates as one trial. We perform 100 trials for both search methods and report the results in Fig. 11.

In this figure, for each application, there are $N$ points for each search method, corresponding to all the $N$ pipeline candidates. For each point, we get the average running end time and score, as well as the variance of the scores over the 100 trials. It is shown in Fig. 11 that the scores obtained from prioritized search are relatively widely distributed, because the pipeline candidates searched first have higher scores while the pipeline candidates searched last have lower scores. On the contrary, the scores from random search are nearly the same for all pipeline candidates because of the randomness. Meanwhile, we observe that the higher score pipeline candidates of prioritized pipeline search have a smaller average end time, which means that the high score pipeline candidates are searched first. The heuristics may be promising to return better pipeline candidates when we "early stop" the merge operation, thus potentially improving the user experience.

F. Distributed Training on Large ML Model

Note that a pipeline could contain any ML model and some models such as DL models require long training time, e.g., ResNet [15] and Xceptionnet [5]. In this case, since MLCask supports any executable as a pipeline component, distributed training can be applied as long as the executable contains the library for distributed training.

In this section, we analyze how much speedup we could achieve if we apply up to 8 GPUs for synchronous distributed training in the same computing node. We take ResNet18 [15] model as an example. The speedup on the model due to distributed training is shown in Fig. 12(a). From this figure, we observe that the training loss decreases faster over training time for more GPUs. This is because more GPUs lead to an increase in sample processing throughput. Consequently, with distributed training for the large ML models in the pipeline, it is possible that the pipeline time can be greatly reduced.

Since pipeline time consists of model training time, pre-processing time, and storage time, analyzing the pipeline time speedup brought about by distributed model training needs to take other components in the pipeline into consideration. We thus formalize the pipeline time speedup due to the distributed model training as follows,

$$\text{Speedup (times)} = \frac{1}{(1 - p) + \frac{k}{p}},$$

where $p$ is the portion of model training time out of the total pipeline time, and $k$ is the speedup of the model training due to distributed training. We show the pipeline time speedup for different combinations of $k$ and $p$ in Fig. 12(b). From the figure, we observe that both increased $k$ and increased $p$ lead to increased pipeline time speedup. As long as $k$ is larger than 1, the pipeline time speedup is larger than 1. Specifically, when the portion of model training time is more than 0.9 and the speedup of the model training equals 8, the pipeline time is less than one fourth of the original pipeline time, which saves a lot of time.
Pipeline time speedup

Such as Maven

Weide et al. propose versioning for end-to-end ML pipelines to MLCask are proposed in [31], [32]. In these two works, pipelines for datasets and model evolution are not supported ML pipelines. Nevertheless, version control semantics of the online management, maintenance, and serving of the pipelines to support ease of use, while Velox [6] focuses enable users to store, index, track, and explore different former, and Estimator. ModelDB [35] and SHERLOCK [34] pipelines by introducing the concepts of DataFrame, Transformation, and Estimator. ModelDB [35] and SHERLOCK [34] enable users to store, index, track, and explore different pipelines to support ease of use, while Velox [6] focuses on online management, maintenance, and serving of the ML pipelines. Nevertheless, version control semantics of the pipelines for datasets and model evolution are not supported by the aforementioned methods.

The pipeline management systems that are most similar to MLCask are proposed in [31], [32]. In these two works, Weide et al. propose versioning for end-to-end ML pipelines to maintain multiple versions of a pipeline. This work archives different versions of data into distinctive disk folders, which may lead to difficulty in tracing the version history and incur huge storage cost. The work in [31] addresses the asynchronous pipeline update problem. However, how to set version number remains undefined. In addition, those two works pay little attention to the collaborative environment, and implicitly assume that all developers work as an entity.

Another line of research works focuses on making use of intermediate results for optimizing the execution of ML pipelines or for diagnosis. MLFlow [40] provides a tracking API for users to log the intermediate results to a specific directory. Derakhshan et al. [8] materialize the intermediate results that have a high likelihood of future reuse and select the optimal subset of them for reuse. In terms of utilizing intermediate results to debug or diagnose the ML pipelines, MISTIQUE [33] efficiently captures, stores, and queries intermediate results for diagnosis using techniques such as quantization, summarization, and data de-duplication, etc., that do not affect the diagnostic results. Zhang et al. [41] diagnose the ML pipeline by using fine-grained lineage, e.g., elements in a matrix or attributes in a record. The above mentioned works emphasize the use of intermediate results as opposed to addressing the non-linear version history problem.

In addition, versioning for ML pipelines is related to workflow management and provenance. Chirigati et al. [4], Koop et al. [19], and Freire et al. [11] discuss workflow provenance in terms of enhancing the reproducibility of scientific workflows, improving user experience [9] and building provenance middle-ware for database. Gharibi et al. [13], Lourenço et al. [21], and Schelter et al. [23] focus on the tracking experiments on ML pipelines. However, non-linear version history is not discussed in these works.

Lastly, KeystoneML [30] is a system that optimizes the end-to-end large-scale machine learning applications for high-throughput training in a distributed setting by using high-level APIs. These optimizations are orthogonal to those used by MLCask, which targets at collaboratively updating a pipeline. AutoML Systems. Automated ML (AutoML) systems focus on automated learning algorithm selection and hyper-parameter tuning. Auto-Sklearn [10] designs meta-learning to automatically take into account past performance on similar datasets to warm-start the Bayesian Optimization (BO) method. TPOT [26] follows a standard Genetic Programming process, i.e., generating an initial population of random pipelines and employing crossover, mutation operators to obtain the final ML pipelines that improve the classification accuracy. Alpine Meadow [29] is an adaptive and interactive AutoML system where users are enabled to embed their domain knowledge. Compared with AutoML systems, MLCask focuses on versioning for the iterative development of ML pipelines instead of only finding a best-so-far pipeline for users. With MLCask, users are able to compare different processing methods or models, perform iterative modifications...
of the pipeline via “trial-and-error” and reuse the pipelines which have been tried before. The search space of ML Cask during the merge consists of the configurations that have been tried before whereas AutoML systems search from a pre-defined large search space.

ML Cask facilitates both feature selection and model selection. With ML Cask, users can try different feature selection methods and analytics models for better performance. All these changes and the corresponding performance will be automatically saved and the best combination of changes will be merged into the master branch via the metric-driven merge operation.

IX. CONCLUSIONS

In this paper, we propose ML Cask to address the key challenges of constructing an end-to-end Git-like ML system for collaborative analytics, in the context of developing or maintaining data analytics applications. Firstly, non-linear pipeline version control is introduced to isolate pipelines for different user roles and various purposes. Secondly, the challenge of asynchronous pipeline update is supported with lineage tracking based on semantic versioning and the ML oriented merge operation. The merge operation we design is guaranteed to find the global optimal solution since it selects from the best score of all feasible pipeline candidates. Thirdly, two pruning methods are proposed to reduce the metric-driven merge operation cost for the pipeline search. For sub-optimal solution under a limited time budget, we present the prioritized pipeline search which provides the trade-off between time complexity and solution quality. Extensive experimental results confirm the superiority of ML Cask in terms of storage cost and computation efficiency. ML Cask has been fully implemented and deployed at a major public hospital.

REFERENCES

[1] B. S. Arab, S. Feng, B. Glavic, S. Lee, X. Niu, and Q. Zeng. GProM: A swiss army knife for your provenance needs. A Quarterly bulletin of the Computer Society of the IEEE Technical Committee on Data Engineering, 2018.

[2] T. Bai, S. Zhang, B. L. Eleston, and S. Vaccete. Interpretable representation learning for healthcare via capturing disease progression through time. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pages 43–51, 2018.

[3] S. Cai, G. Chen, B. C. Ooi, and J. Gao. Model slicing for supporting complex analytics with elastic inference cost and resource constraints. Proceedings of the VLDB Endowment, 13(2):96–99, 2019.

[4] F. Chirigati and J. Freire. Towards integrating workflow and database provenance. In International Provenance and Annotation Workshop, pages 11–23, 2012.

[5] F. Chollet. Xception: Deep learning with depthwise separable convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 1251–1258, 2017.

[6] D. Crankshaw, P. Balaji, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. In Proceedings of the Biennial Conference on Innovative Data Systems Research, 2015.

[7] J. Dai, M. Zhang, G. Chen, J. Fan, K. Y. Ngiam, and B. C. Ooi. Fine-grained concept linking using neural networks in healthcare. In Proceedings of the International Conference on Management of Data, pages 51–66, 2018.

[8] B. Derakhshan, A. Rezaei Mahdijazi, Z. Abedjan, T. Rabbi, and V. Markl. Optimizing machine learning workloads in collaborative environments. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 1701–1716, 2020.

[9] D. Deutch, N. Frost, and A. Gilad. Provenance for non-experts. IEEE Data Engineering Bulletin, pages 3–14, 2018.

[10] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter. Auto-sklearn: efficient and robust automated machine learning. In Automated Machine Learning, pages 113–134. Springer, Cham, 2019.

[11] J. Freire and F. Chirigati. Provenance and the different flavors of computational reproducibility. IEEE Data Engineering Bulletin, page 15, 2018.

[12] J. Gama, I. Zliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia. A survey on concept drift adaptation. ACM computing surveys, pages 1–37, 2014.

[13] G. Gharibi, V. Walunj, R. Alanazi, S. Rehla, and Y. Lee. Automated management of deep learning experiments. In Proceedings of the International Workshop on Data Management for End-to-End Machine Learning, pages 1–4, 2019.

[14] G. Guzzetta, G. Jurman, and C. Furlanello. A machine learning pipeline for quantitative phenotype prediction from genotype data. BMC bioinformatics, pages 1–9, 2010.

[15] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

[16] S. Huang, L. Xu, J. Liu, A. J. Elmore, and A. Parameswaran. OrpheusDB: Bolt-on versioning for relational databases. Proceedings of the VLDB Endowment, pages 1130–1141, 2017.

[17] A. Kaul, S. Maheshwary, and V. Pudi. Autolearn—automated feature generation and selection. In International Conference on data mining, pages 217–226, 2017.

[18] J. A. Killian, B. Wilder, A. Sharma, V. Choudhary, B. Dilkina, and M. Tambe. Learning to prescribe interventions for tuberculosis patients using digital adherence data. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pages 4340–4348, 2019.

[19] D. Koop, C. E. Scheidegger, J. Freire, and C. T. Silva. The provenance of workflow upgrades. In International Provenance and Annotation Workshop, 2010.

[20] C. Lee, Z. Luo, K. Y. Ngiam, M. Zhang, K. Zheng, G. Chen, B. C. Ooi, and W. L. J. Yip. Big healthcare data analytics: Challenges and applications. In Handbook of Large-Scale Distributed Computing in Smart Healthcare, pages 11–41, 2017.

[21] R. Lourenço, J. Freire, and D. Shasha. Debugging machine learning pipelines. In Proceedings of the International Workshop on Data Management for End-to-End Machine Learning, 2019.

[22] Z. Luo, S. Cai, J. Gao, M. Zhang, K. Y. Ngiam, G. Chen, and W. Lee. Adaptive lightweight regularization tool for complex analytics. In International Conference on Data Engineering, pages 485–496, 2018.

[23] F. Ma, J. Gao, Q. Suo, Q. You, J. Zhou, and A. Zhang. Risk prediction on electronic health records with prior medical knowledge. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pages 1910–1919, 2018.

[24] M. Maddox, D. Goehring, A. J. Elmore, S. Madden, A. Parameswaran, and A. Deshpande. Decibel: The relational dataset branching system. Proceedings of the VLDB Endowment, page 624–635, 2016.

[25] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, and et al. MLlib: Machine learning in apache spark. Journal of Machine Learning Research, page 1235–1241, 2016.

[26] R. S. Olson and J. H. Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In Workshop on automatic machine learning, pages 66–74, 2016.

[27] B. C. Ooi, K.-L. Tan, S. Wang, W. Wang, Q. Cai, G. Chen, J. Gao, Z. Luo, A. K. Tung, Y. Wang, et al. Singa: A distributed deep learning platform. In Proceedings of the ACM international conference on Multimedia, pages 685–688, 2015.

[28] S. Schelter, J.-H. Böse, J. Kirschnick, T. Klein, and S. Seufert. Automatically tracking metadata and provenance of machine learning experiments. In Machine Learning Systems workshop at NIPS, 2017.

[29] Z. Shang, E. Zgraggen, B. Buratti, F. Kossmann, P. Eichmann, Y. Chung, C. Binnig, E. Uptal, and T. Kraska. Democratizing data science
through interactive curation of ml pipelines. In Proceedings of the ACM SIGMOD International Conference on Management of Data, 2019.

[30] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. KeystoneML: Optimizing pipelines for large-scale advanced analytics. In international conference on data engineering, pages 535–546, 2017.

[31] T. van der Weide, D. Papadopoulos, O. Smirnov, M. Zielinski, and T. van Kasteren. Versioning for end-to-end machine learning pipelines. In Proceedings of the Workshop on Data Management for End-to-End Machine Learning, 2017.

[32] T. Van der Weide, O. Smirnov, M. Zielinski, D. Papadopoulos, and T. van Kasteren. Versioned machine learning pipelines for batch experimentation. In Machine Learning Systems workshop at NIPS, 2016.

[33] M. Vartak, J. M. F. da Trindade, S. Madden, and M. Zaharia. Mistique: A system to store and query model intermediates for model diagnosis. In Proceedings of the International Conference on Management of Data, pages 1285–1300, 2018.

[34] M. Vartak, P. Ortiz, K. Siegel, H. Subramanyam, S. Madden, and M. Zaharia. Supporting fast iteration in model building. In Machine Learning Systems workshop at NIPS, 2015.

[35] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. ModelDB: A system for machine learning model management. In Proceedings of the Workshop on Human-In-the-Loop Data Analytics, 2016.

[36] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, B. C. Ooi, and P. Ruan. Forkbase: An efficient storage engine for blockchain and forkable applications. Proceedings of the VLDB Endowment, page 1137–1150, 2018.

[37] W. Wang, G. Chen, A. T. T. Dinh, J. Gao, B. C. Ooi, K.-L. Tan, and S. Wang. Singa: Putting deep learning in the hands of multimedia users. In Proceedings of the ACM international conference on Multimedia, pages 25–34, 2015.

[38] W. Wang, M. Zhang, G. Chen, H. Jagadish, B. C. Ooi, and K.-L. Tan. Database meets deep learning: Challenges and opportunities. ACM SIGMOD Record, 45(2):17–22, 2016.

[39] D. Xin, S. Macke, L. Ma, J. Liu, S. Song, and A. Parameswaran. Helix: Holistic optimization for accelerating iterative machine learning. Proceedings of the VLDB Endowment, page 446–460, 2018.

[40] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. IEEE Data Engineering Bulletin, 41(4):39–45, 2018.

[41] Z. Zhang, E. R. Sparks, and M. J. Franklin. Diagnosing machine learning pipelines with fine-grained lineage. In Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing, pages 143–153, 2017.

[42] K. Zheng, S. Cai, H. R. Chua, W. Wang, K. Y. Ngiam, and B. C. Ooi. Tracer: A framework for facilitating accurate and interpretable analytics for high stakes applications. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 1747–1763, 2020.

[43] K. Zheng, J. Gao, K. Y. Ngiam, B. C. Ooi, and W. L. J. Yip. Resolving the bias in electronic medical records. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 2171–2180, 2017.