HPE ML Data Management: Deployment Guide

Learn about the basics of HPE ML Data Management (MLDM) and how to install the platform within a Kubernetes cluster.

Contents

Terms and Definitions

Term	Definition
HPE ML Data Management	HPE ML Data Management (MLDM) is built upon the open source Pachyderm platform. For the latest documentation, visit the HPE ML Data Management documentation.
Pachyderm	The open source Pachyderm platform

Introduction to MLDM

MLDM is a data science platform that provides data-driven pipelines with version control and autoscaling. It is container-native, allowing developers to use the languages and libraries that are best suited to their needs, and runs across all major cloud providers and on-premises installations.

The platform is built on Kubernetes and integrates with standard tools for CI/CD, logging, authentication, and data APIs, making it scalable and incredibly flexible. MLDM's data-driven pipelines allow you to automatically trigger data processing based on changes in your data, and the platform's autoscaling capabilities ensure that resource utilization is optimized, maximizing developer efficiency.

Key Features

The following are the key features of MLDM that make it a powerful data processing platform.

Data-driven Pipelines

- Automatically trigger pipelines based on changes in the data.
- Orchestrate batch or real-time data pipelines.
- Only process dependent changes in the data.
- Reproducibility and data lineage across all pipelines.

Version Control

- Track every change to your data automatically.
- Works with any file type.
- Supports collaboration through a git-like structure of commits.

Autoscaling and Deduplication

- Auto scale pipelines based on resource demand.
- Automatically parallelize large data sets.
- Automatically deduplicate data across the entire platform.

Flexibility and Infrastructure Agnosticism

- Use existing cloud or on-premises infrastructure.
- Process any data type, size, or scale in batch or real-time pipelines.
- Container-native architecture allows for developer autonomy.
- Integrates with existing tools and services, including CI/CD, logging, authentication, and data APIs.

Basic Concepts

Pachyderm File System

The Pachyderm File System (PFS) is the backbone of the MLDM data platform, providing a secure, scalable, and efficient way to store and manage large amounts of data. It is a version-controlled data management system that enables users to store any type of data in any format and scale, from a single file to a directory of files. The PFS is built on top of Postgres and an object store, ensuring that your data is secure, consistent, and easily accessible. With PFS, users can version their data and work collaboratively with their teams, using branches and commits to manage and track changes over time.

Repositories (Repo)

MLDM repositories are version controlled, meaning that they keep track of changes to the data stored within them. Each repository can contain any type of data, including individual files or directories of files, and can handle data of any scale.

Branches

Branches in MLDM are like those in Git. They are pointers to commits that move along a growing chain of commits. This allows you to work with different versions of your data within the same repository.

Commits

A commit in MLDM is created automatically whenever data is added to or deleted from a repository. Each commit preserves the state of all files in the repository at the time of the commit, similar to a snapshot. Each commit is uniquely identifiable by a UUID and is immutable, meaning that the source data can never change.

Pachyderm Pipeline System

The Pachyderm Pipeline System (PPS) is a core component of the MLDM platform, designed to run robust data pipelines in a scalable and reproducible manner. With PPS, you can define, execute, and monitor complex data transformations using code that is run in Docker containers. The output of each pipeline is version-controlled in an MLDM data repository, providing a complete, auditable history of all processing steps. In this way, PPS provides a flexible, data-driven solution for managing your data processing needs, while keeping data and processing results secure, reproducible, and scalable.

Pipelines

MLDM pipelines are used to transform data from MLDM repositories. The output data is versioned in a MLDM data repository, and the code for the transformation is run in Docker containers. Pipelines are triggered by new commits to a branch, making them data driven.

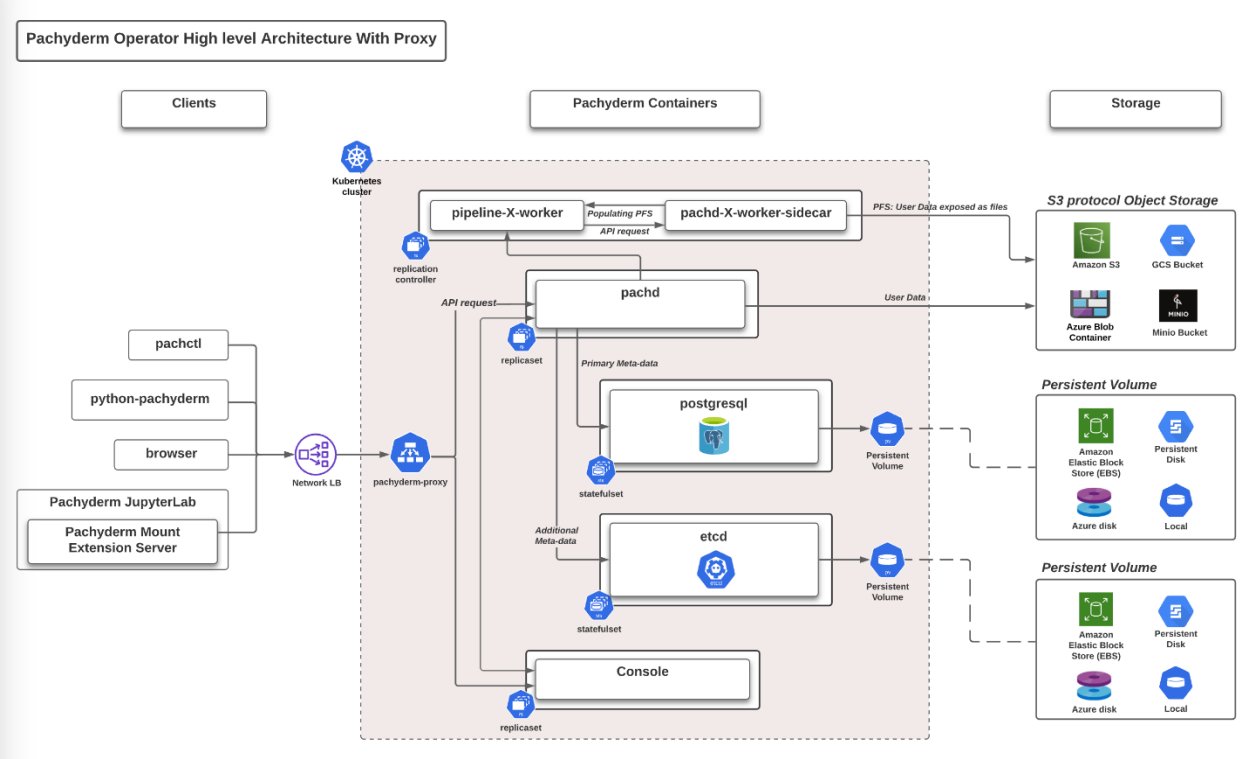
Jobs

A job in MLDM is the execution of a pipeline with a new commit. The data is distributed and parallelized computation is performed across a cluster. Each job is uniquely identified, making it possible to reproduce the results of a specific job.

Datum

A datum in MLDM is a unit of computation for a job. It is used to distribute the processing workloads and to define how data can be split for parallel processing.

High-Level Architecture Diagram



How MLDM Works

MLDM is deployed within a Kubernetes cluster to manage and version your data using projects, input repositories, pipelines, datums and output repositories. A project can house many repositories and pipelines, and when a pipeline runs a data transformation job it chunks your inputs into datums for processing.

The number of datums is determined by the glob pattern defined in your pipeline specification; if the shape of your glob pattern encompasses all inputs, it will process one datum; if the shape of your glob pattern encompasses each input individually, it will process one datum per file in the input, and so on.

The end result of your data transformation should always be saved to `/pfs/out`. The contents of `/pfs/out` are automatically made accessible from the pipeline's output repository by the same name. So, all files saved to `/pfs/out` for a pipeline named foo are accessible from the foo output repository.

Pipelines combine to create DAGs, and a DAG can be comprised of just one pipeline.

How to Interact with MLDM

You can interact with your MLDM cluster using the PachCTL CLI or through Console, a GUI.

- **PachCTL** is great for users already experienced with using a CLI.

- **Console** is great for beginners and helps with visualizing relationships between projects, repos, and pipelines.

The following are some quick links to help you perform operations in MLDM using both the CLI and Console.

- Project Operations
- Pipeline Operations
- Branch Operations
- Datum Operations
- Provenance Operations

Before You Start

Before you can deploy MLDM, you will need to perform the following actions:

1. Install Kubectl
2. Install Helm
3. Deploy Kubernetes
4. Deploy two Kubernetes persistent volumes for MLDM metadata storage
5. Deploy an object store using a storage provider like MinIO, EMC's ECS, or SwiftStack to provide s3-compatible access to your data storage.
6. Install PachCTL and PachCTL Auto-completion.

Note: This guide assumes that you are deploying MLDM on-prem; for all of the most up-to-date installation methods, see our online documentation.

Install on Kubernetes

1. Install NVIDIA GPU Operator via Helm

The NVIDIA GPU Operator enables Kubernetes administrators to manage GPU nodes like CPU nodes within the cluster by relying on a standard OS image for both node types. The GPU Operator then provisions any software components needed for the GPUs. Verson 1.9.1 or later is required.

1. Download the NVIDIA GPU Operator, then run the following command:

```
helm repo add nvaie https://helm.ngc.nvidia.com/nvaie \
  --username='$oauthtoken' --password='<password>' \
  && helm repo update
```

2. Install:

```
helm install --wait gpu-operator nvaie/gpu-operator-<M>-<m> -n gpu-operator \
  --set driver.repository=nvcr.io/nvidia \
  --set driver.image=driver \
  --set driver.version=<driver-version> \
  --set driver.licensingConfig.configMapName=""
```

For a complete installation guide, see the official NVIDIA GPU Operator installation instructions.

2. Install MLDM via Helm

You can obtain MLDM and its updates via Helm.

```
helm repo add pachyderm https://helm.pachyderm.com
helm repo update
```

3. Create Enterprise Key Secret

Before we begin configuring our Helm `values.yaml` file, let's create a secret key for our enterprise license. This will allow us to enable enterprise-only features like authentication/authorization.

There are a few ways to do this, but the easiest would be to:

1. Create a file named `pachyderm-enterprise-key.json`.
2. Copy & paste the following template, adding your license key:

```
{
  "kind": "Secret",
  "apiVersion": "v1",
  "metadata": {
    "name": "pachyderm-enterprise-key",
    "creationTimestamp": null
  },
  "data": {
    "enterprise-license-key": "<replace-with-key>"
  }
}
```

Then, you can add the secret to PachCTL:

```
pachctl create secret -f pachyderm-enterprise-key.json
```

We'll use the secret's name, `pachyderm-enterprise-key`, later in our `values.yaml` configuration file.

4. Configure Helm Values

The Helm `values.yaml` file defines your deployment, storage, and enterprise feature settings. You can view and copy a full example Helm chart from GitHub or ArtifactHub for reference when configuring your Helm Chart Values (HCVs).

If you already have MLDM deployed and wish to see the already existing user-provided values, you can run the following:

```
helm get values pachyderm > values.yaml
```

Add Storage Classes to Helm Values

Update your Helm values file to include the storage classes you are going to use:

```
etcd:
  storageClass: MyStorageClass
  size: 10Gi
postgresql:
  persistence:
    storageClass: MyStorageClass
    size: 10Gi
```

Size & Configure Object Store

1. Determine the endpoint of your object store, for example `minio-server:9000`.
2. Choose a unique name for the bucket you will dedicate to MLDM.
3. Create a new access key ID and secret key for MLDM to use when accessing the object store.
4. Update the MLDM Helm values file with the endpoint, bucket name, access key ID, and secret key.

```
pachd:
  storage:
    backend: minio
    minio:
      endpoint: minio-server:9000
      bucket: pachyderm-bucket
      id: pachyderm-access-key
      secret: pachyderm-secret-key
      secure: false
```

Add Enterprise License Key Secret Name

Add the secret name created in a previous step to your Helm `values.yaml` file at the following location:

```
pachd:
  enterpriseLicenseKeySecretName: "pachyderm-enterprise-key"
```

Configure Authentication & Authorization

To set up Authentication, you must have an active Enterprise License and create a corresponding key secret (completed in previous sections). Then you must set up both TLS and an OIDC connector such as Auth0 or Okta. Once set up, you are ready to add the OIDC configuration to your Helm `values.yaml` file. The following is an Auth0 example:

```
oidc:
  upstreamIDPs:
  - type: oidc
    id: auth0
    name: Auth0
    config:
        issuer: https://<auth0.app.domain.url>/
        clientID: FbTzaVdFCB9TbX07pXqxBwofuEOux004
        clientSecret: 1kbxtx22DLGSULrjJgV-TaaUs1qPLK5yTOsrmwwVNXP9U
        redirectURI: https://<proxy.host.value.com>/dex/callback
        insecureEnableGroups: true
        insecureSkipEmailVerified: true
        insecureSkipIssuerCallbackDomainCheck: false
```

After deploying MLDM, you can log in as the root user and begin to add users to certain resource types such as Projects and Repos.

```
pachctl auth set project <project-name> <role-name> user:<username@email.com>
```

5. Deploy

Run the following command:

```
helm install pachyderm -f values.yaml pachyderm/pachyderm --version
<your_chart_version>
```

You can check the status of your deployment by running the following kubectl command:

```
Kubectl get pods
```

6. Connect & Login

After the `pachd` pod is up and ready, you can connect, check the version, and log in using the following PachCTL commands:

```
pachctl connect grpcs://<your-proxy.host-value>:443
pachctl version
pachctl auth login
```

7. Build a GPU-enabled Pipeline

Now you can define pipeline specifications and deploy pipelines to your MLDM cluster. The following example deploys a GPU-enabled pipeline from our Market Sentiment Analysis example in GitHub. You can review the relevant PPS sections in more detail at the Resource Limits and Resource Requests sections of our documentation.

1. Create an `example.json` file and populate it with the following:

```
{
  "pipeline": {
    "name": "train_model"
  },
  "description": "Fine tune a BERT model for sentiment analysis on financial
data.",
  "input": {
    "cross": [
      {
        "pfs": {
          "repo": "dataset",
          "glob": "/"
        }
      },
      {
        "pfs": {
          "repo": "language_model",
          "glob": "/"
        }
      }
    ]
  },
  "transform": {
    "cmd": [
      "python", "finbert_training.py", "--lm_path", "/pfs/language_model/",
"--cl_path", "/pfs/out", "--cl_data_path", "/pfs/dataset/"
    ],
    "image": "pachyderm/market_sentiment:dev0.25"
  },
  "resourceLimits": {
    "gpu": {
      "type": "nvidia.com/gpu",
      "number": 1
    }
  },
  "resourceRequests": {
    "memory": "4G",
    "cpu": 1
  }
}
```

2. Run `pachctl create pipeline -f example.json`.
3. List your pipelines in the terminal using `pachctl list pipelines` or view the Console.